Let's get our hands dirty: a comprehensive evaluation of DAQDB, key-value store for petascale hot storage

Adam Abed Abud[1], *Danilo* Cicalese[1], *Grzegorz* Jereczek[2,*], *Fabrice* Le Goff[1], *Giovanna* Lehmann Miotto[1], *Jeremy* Love[3], *Maciej* Maciejewski[2], *Remigius K* Mommsen[4], *Jakub* Radtke[2], *Jakub* Schmiegel[2], and *Malgorzata* Szychowska[2]

[1]European Laboratory for Particle Physics, CERN, Geneva 23, CH-1211, Switzerland
[2]Intel Technology Poland, ul. Slowackiego 173, 80-298 Gdansk, Poland
[3]Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, USA
[4]Fermi National Accelerator Laboratory, Batavia, IL 60510, USA

Abstract. Data acquisition systems are a key component for successful data taking in any experiment. The DAQ is a complex distributed computing system and coordinates all operations, from the data selection stage of interesting events to storage elements. For the High Luminosity upgrade of the Large Hadron Collider, the experiments at CERN need to meet challenging requirements to record data with a much higher occupancy in the detectors. The DAQ system will receive and deliver data with a significantly increased trigger rate, one million events per second, and capacity, terabytes of data per second. An effective way to meet these requirements is to decouple real-time data acquisition from event selection. Data fragments can be temporarily stored in a large distributed key-value store. Fragments belonging to the same event can be then queried on demand, by the data selection processes. Implementing such a model relies on a proper combination of emerging technologies, such as persistent memory, NVMe SSDs, scalable networking, and data structures, as well as high performance, scalable software. In this paper, we present *DAQDB* (Data Acquisition Database) — an open source implementation of this design that was presented earlier, with an extensive evaluation of this approach, from the single node to the distributed performance. Furthermore, we complement our study with a description of the challenges faced and the lessons learned while integrating DAQDB with the existing software framework of the ATLAS experiment.

1 Introduction

Decoupling real-time data acquisition from event selection is a promising approach for future data acquistion (DAQ) systems of large-scale experiments like the Large Hadron Collider (LHC) at CERN. In [1], facing the new requirements of the High Luminosity upgrade of the LHC, we analyzed this approach in detail and proposed a possible design using a large distributed key-value store (KVS) as a generic high-bandwidth data storage system. In this paper we present the first performance evaluation of our open-source implementation of this design — Data Acquisition Database (DAQDB) [2].

*e-mail: grzegorz.jereczek@intel.com

DAQDB builds on emerging technologies like persistent memory, NVMe SSDs, scalable networking, and scalable data structure. It becomes feasible to substantially increase the temporary buffer for the incoming data and to reduce the tight coupling of the data readout and data processing subsystems in a typical DAQ system. In effect, processing timeouts can be less strict, sensor calibration can be improved implying a purer selection of events, and the use of computing resources can be optimized by designing for the average instead of the peak data rate. These factors can be particularly important for the High Luminosity upgrades of LHC, for which the DAQ systems will need to digest a few terabytes of data each second. Also, DAQDB hides away many physical aspects usually needed for experiment's data assembly abstracting it with a logical interface.

The remainder of this paper is organized as follows. We first summarize the key concepts behind logical event building with DAQDB in Section 2. An extensive performance evaluation in different scenarios is then given in Section 3. Next we briefly describe the TDAQ architecture of the ATLAS experiment with the first attempts of integrating DAQDB in Section 4. We conclude our work in Section 5.

2 Logical event building with DAQDB

In the High-Luminosity upgrade of the LHC, large-scale experiments such as ATLAS or CMS are considering a logical event building approach contrary to the standard physical event building in which fragments are transferred over a network from the readout nodes to a single physical location. Logical event building relies on saving all the fragments in a distributed storage buffer such as a large key-value store. The event building process is then taken care internally by the KVS by only using metadata operations. For example, the processing nodes can request or delete fragments belonging to a particular event by executing a query to the KVS.

DAQDB is used as a distributed key-value store that implements the logical event building approach. In practice, this means associating each fragment with a unique identifier (*key*). Its structure has to be configurable and large enough to support specifics of DAQ systems: it has to identify the event, the subdetector and the run of the experiment. Considering 64 bits for event ID, and 16 bits for subdetector and run IDs, the total length of the key is in the order of 100 bits. The value of the fragment can then be added into the database by implementing a *put(key, value)* operation. Similarly, a *get(key)* operation can be used to retrieve the value associated with a specific fragment. In this way, a consuming object such as a filtering node can query the KVS to retrieve the values of the fragment. Finally, in order to scale the system the KVS is distributed across several nodes. DAQDB uses a hashing function on the fragment key to identify the location and the node where the data entry is stored.

3 DAQDB evaluation

We evaluate DAQDB using *minidaq* application that is part of the DAQDB repository [2]. Minidaq is built on top of DAQDB and runs a configurable number of *readout* and *collector* threads that emulate the operation of a typical DAQ system. The readout threads insert event data into the DAQDB store, while the collector threads request all event's data and remove them without any processing. DAQDB is either embedded locally into the minidaq application or runs as a server on an independent node accessible over the network. This simplified data flow focuses on data I/O operations. The following section gives a detailed description of the evaluation configurations.

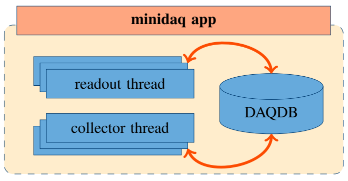

Figure 1. Single-node configuration.

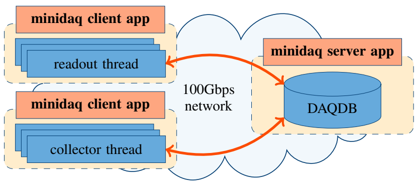

Figure 2. Multi-node configuration.

3.1 Evaluation configurations

Figure 1 and Figure 2 show two configurations that are being evaluated. The application containing the actual DAQDB store runs always on the same server platform being S2600WF board with two Intel® Xeon® Platinum 8280L 28-core CPUs running at 2.70 GHz. It is equipped with 3 TB of Intel® Optane™ persistent memory (PMem) 100 series and 96 GB of DDDR4 memory per each CPU node, but DAQDB is evaluated using only one CPU to avoid cross CPU effects. Eventually, an independent DAQDB server instance is to be run on each CPU. DAQDB uses 64-bit adaptive radix tree (ART) [3] with 8b per ART node, with DAQ-specific optimizations [1]. Both event data and data structure are kept on PMem. The former is kept volatile, while the latter is persisted.

Two additional nodes are used for evaluations using a remote DAQDB instance. In this case minidaq client nodes are the same server platforms as the DAQDB server's, but equipped with Intel® Xeon® Gold 6252 or 6140 CPUs running at 2.10 GHz or 2.30 GHz respectively. eRPC [4] in raw Ethernet mode with OFED 4.7 is used as transport layer for DAQDB. Network cards are Mellanox ConnectX-5. In this configuration DAQDB server runs 14 request handler threads serving remote minidaq clients. We use Centos 7.7 with kernel 4.19 and CERN LCG environment 95 [5] with GCC 8 in all cases.

DAQDB is configured with a 64-bit key: 40b event ID, 8b detector ID, 16b component ID. Each minidaq readout thread inserts a single fragment with a unique event ID. Detector and component IDs are fixed, which means that an entire event consists of just a single event fragment. Although this is not a typical event configuration, the goal of this evaluation is to assess base performance numbers of DAQDB. Single test time is either five minutes or a total of 50M iterations. Throughput samples are measured in 5 ms intervals

3.2 Multi-core scalability

Figure 3 shows how DAQDB scales with the increasing number of readout and collector threads for different event fragment sizes. DAQDB store is first filled with data by readout threads, followed by filtering threads draining the store event by event with incremental event IDs. Local write requests saturate at approximately 2 MOPS with good multi-core scalability and requiring less than 15 client threads to reach maximum performance for smaller fragment sizes. Bandwidth-wise DAQDB saturates at 10 GB/s with 10 kB fragments with similar amount of threads. Local read requests show similar characteristics reaching the maximum of 7 MOPS and 20 GB/s respectively. We compare these numbers with the basic measurements of PMem [6] reporting 39.4 GB/s of read and 13.9 GB/s of write throughput. Those numbers represent sequential operations in volatile mode of operation. Since DAQDB uses mixed-size operations on non-volatile data structure and short-lived volatile values, reaching more than 50 % of this baseline is considered satisfactory. The readout system of the ATLAS experiment will require 6 TB/s input bandwidth after the Phase-II upgrade. A DAQDB-based hot storage

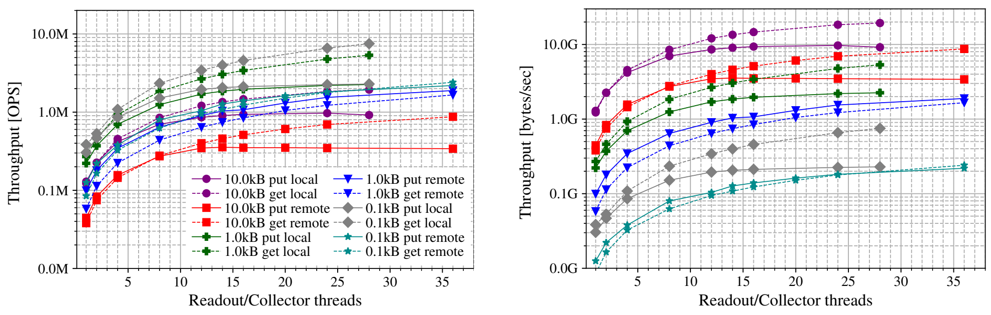

Figure 3. DAQDB throughput (operations per second and bytes per second) for different number of client threads.

Function	Time spent	
	@100 B	@10 kB
→ DaqDB::KVStore::Alloc	77 %	20 %
↳ DaqDB::TreeImpl::findValueInNode	— 71 %	— 13 %
↳ pmemobj_mutex_lock	——— 33 %	——— < 5 %
↳ pmemobj_mutex_unlock	——— 28 %	——— < 5 %
→ pmem_memcpy_nodrain	< 5 %	60 %
→ DaqDB::KVStore::Put	5 %	13 %

Table 1. Code profiling summary for 100 B and 10 kB fragment sizes for a minidaq readout thread with embedded DAQDB.

solution would require a reasonable amount of 300 dual-CPU servers, assuming 10 GB/s per-CPU DAQDB bandwidth and 10 kB fragment size. Those numbers look promising, but they neglect some DAQDB features that are still under development, like data structure cleanup and offload to second-line buffer.

Performance drop is observed when switching DAQDB into remote mode. More client threads are required in order to reach the same throughput as in the local mode. With 10 kB fragment size the same level is not even reached, saturating below 4 GB/s on write. The reasons are two-fold. Firstly, DAQDB supports now only fully synchronous operations without any batching, which means that for every request each thread waits until a response is received before proceeding with its workflow. Thus, round-trip network latency is added to every request, so more threads are needed to saturate the bandwidth offered by DAQDB. Secondly, eRPC operates with 1000 B MTU, so additional segmentation latency and CPU power affect the characteristics for 1 kB and 10 kB fragment sizes. Another interesting effect is that with remote DAQDB write performance is close to read performance, contrary to the local mode of operation. This indicates that network latency is the dominating factor and becomes the bottleneck, not the memory access to the data store on the server side itself. This effect does not occur for 10 kB fragments though, when memory bandwidth and network latency are both significant factors.

The effects of persistent memory bandwidth and access latency on the DAQDB performance are confirmed by the profiling summary in Table 1. With small fragments, most time is spent while traversing the tree structure looking for the requested key and allocating new nodes within. The latter requires locks to prevent multiple threads from modifying the structure concurrently, which becomes the limiting factor. On the contrary, for larger fragments,

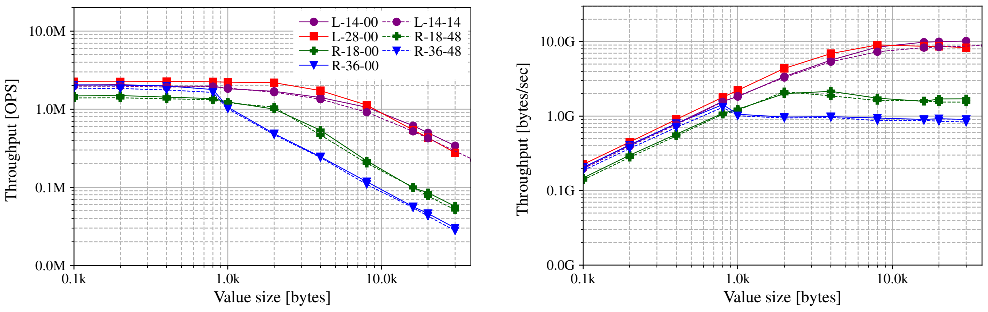

Figure 4. DAQDB throughput (operations per second and bytes per second) for different value sizes.

it is the event data copy to DAQDB directly dependent on the persistent memory bandwidth that defines the upper bound on DAQDB performance.

3.3 Dependence on value size

In the second evaluation we are analyzing how DAQDB performance depends on event fragment size in more detail. We also study if there is any performance impact when both the readout and filtering threads are running in parallel with the latter using the GetAny feature [1] to request the next available event ID from the store. The performance characteristics are presented in Figure 4. The legend is coded with the code $X - nRO - nCL$. X denotes the DAQDB mode of operation being either L meaning locally embedded DAQDB store or R for a remote DAQDB server with network access. nRO defines the number of readout threads accessing DAQDB, whereas nCL is the number of collector threads working in parallel with the readout threads and using GetAny feature for next event retrieval. Both figures show the total throughput achieved by the readout threads only. Data collection throughput is not plotted as it is close to the readout throughput. The latter sets also the upper bound when both worker types are running in parallel with GetAny.

With local DAQDB access, the number of operations per second start to decline from its maximum CPU-bound 2 MOPS with fragments of 20 kB and larger. This is when the memory becomes the limiting factor, reaching the saturation throughput of nearly 10 GB/ sec with 20 kB data fragments. This confirms the observations already described in the previous section. Most important here to observe is the fact that enabling parallel data collection threads does not significantly affect the overall DAQDB performance.

In the remote mode, due to the reasons described in the previous section, more threads are required to reach the maximum number of operations per second. It also starts to decline with lower fragment size of approximately 1 kB because of the extra overhead of data segmentation in order to match eRPC's MTU. Still, the reported eRPC single-thread throughput for large messages is more than 9 GB/s [4]. Such level are not reached with DAQDB in this evaluation. Interestingly, in our case, using less readout threads leads to better performance for large messages above MTU. This indicates that the required packet fragmentation for event data fragments larger than MTU becomes a new point for multi-thread contention. As a consequence the overall DAQDB is severely limited by this transport layer effect. It requires further investigation and optimization at the DAQDB-eRPC interface boundary.

4 Integration with ATLAS TDAQ framework

The trigger and data acquisition (TDAQ) system of the ATLAS experiment uses a two stage selection based on a first level trigger (L1) and a High Level Trigger (HLT). The L1 trigger uses custom made electronics and FPGAs to reduce the input rate from 40 MHz to approximately 100 kHz within a constant latency of 2.5 µs. Even if the L1 trigger makes a quick and rough selection of the most interesting collision data events, it is nonetheless able to considerably reduce the data rate and to identify *Regions of Interest* (RoI). This information represents hints of where potentially interesting events are located across the detector.

The RoI information is then used by HLT to perform a more detailed selection. HLT uses a commodity computing farm with more than 2000 nodes to perform a software based selection. The algorithms used in this stage are more powerful compared to the L1 algorithms. The processing time of each collision data takes on average 100 ms. Finally, accepted data events are sent to a temporary storage solution at a rate of 1.5 GB/s. Figure 5 represents a functional diagram of the ATLAS TDAQ system used during the 2015-2019 (Run 2) data taking. The corresponding event and data rates at each stage are also highlighted.

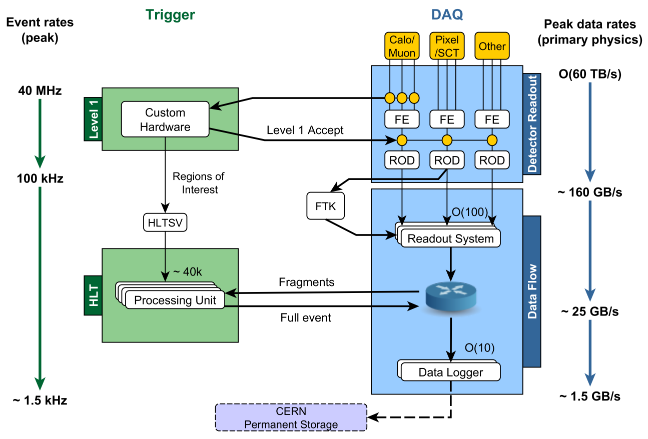

Figure 5. Diagram of the ATLAS Trigger and Data Acquisition system in Run 2 [7].

It is worth mentioning that in the current ATLAS architecture data fragments are stored in the DRAM of the readout nodes and made available for the high level trigger. A dedicated application is responsible for dynamically orchestrating the dataflow process by assigning and transferring the relevant fragments to the processing units in the computing farm. If an event is accepted by the HLT, it is built into a single location and passed on for persistent storage.

The initial approach to integrate DAQDB into the ATLAS TDAQ framework involved the swROD application. This is a software tool designed to interface the readout system (Readout detector or ROD) with the ATLAS High-Level Trigger. The swROD is responsible for building and formatting fragments, as well as executing detector specific operations such as fragment validation or monitoring. A considerable amount of time was invested in integrating the swROD data input to the DAQDB storage back-end and a successful prototype was obtained. Using ATLAS applications is the desired solution because it allows to integrate

DAQDB with a real DAQ system. However, a synchronization element was missing in the emulation environment, such as the one used in minidaq for the initial measurements. This is required in order to prevent the reader threads (HLT) from running ahead of the data writers (swROD).

We noticed that with this approach it was too complex to perform basic performance and scaling tests and as a consequence another method is being considered. It involves setting up a DAQ emulator by developing standalone *put* and *get* applications that would act as readout and event filtering nodes. This flexible approach allows to benchmark and validate the system with applications that have been extensively tested. As an example, the performance of the standalone *put* and *get* applications for a single node was evaluated and the results were compatible with the ones obtained with minidaq. In the future, tests with a complete dataflow system are foreseen in order to assess the performance of DAQDB with the ATLAS TDAQ framework.

5 Conclusion

In this paper we presented the first performance characteristics of a key-value store designed specifically for DAQ based on the emerging persistent memory technology, DAQDB. Single-node evaluation confirms its capabilities to meet the challenging requirements of the High Luminosity upgrade of the LHC reaching 1 MOPS with 10 kB data fragments on write operations. Although these performance levels are not yet reached in distributed mode of operation, our results confirm that DAQDB is functionally capable of providing next unprocessed event retrieval for the filtering farm with negligible impact on the readout performance. The future work will focus on optimizing DAQDB transport layer with asychronous mode of operation, NVMe-based second-level buffer, range scan operations, and proving DAQDB's scalability in large DAQ farms with typical event fragment distributions. In addition, further tests and integration with the ATLAS trigger and data acquisition framework are also foreseen in the future. This will allow to have a more accurate understanding of the performance of DAQDB in a real benchmarking scenario.

References

[1] D. Cicalese, G. Jereczek, F. Le Goff, G. Lehmann Miotto, J. Love, M. Maciejewski, R. Mommsen, J. Radtke, J. Schmiegel, M. Szychowska, EPJ Web Conf. **214**, 01014 (2019)

[2] *Data AcQuisition DataBase*, `https://github.com/daq-db/daqdb`

[3] V. Leis, A. Kemper, T. Neumann, *The adaptive radix tree: ARTful indexing for main-memory databases*, in *ICDE'13* (IEEE, 2013), pp. 38–49, ISBN 978-1-4673-4910-9

[4] A. Kalia, M. Kaminsky, D.G. Andersen, preprint arXiv:1806.00680 (2018)

[5] *LCG Info: Releases, Packages & Platforms*, `http://lcginfo.cern.ch/`

[6] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y.J. Soh, Z. Wang, Y. Xu, S.R. Dulloor et al., arXiv preprint arXiv:1903.05714 (2019)

[7] ATLAS Collaboration, *DAQ approved plots* (2017), `https://twiki.cern.ch/twiki/bin/view/AtlasPublic/ApprovedPlotsDAQ`