EPJ Web of Conferences **214**, 01014 (2019) https://doi.org/10.1051/epjconf/201921401014
CHEP 2018

The design of a distributed key-value store for petascale hot storage in data acquisition systems

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 for high energy physics experiments readout terabytes of data per second from a large number of electronic components. They are thus inherently distributed systems and require fast online data selection, otherwise requirements for permanent storage would be enormous. Still, incoming data need to be buffered while waiting for this selection to happen. Each minute of an experiment can produce hundreds of terabytes that cannot be lost before a selection decision is made. In this context, we present the design of *DAQDB* (Data Acquisition Database) — a distributed key-value store for high-bandwidth, generic data storage in event-driven systems. DAQDB offers not only high-capacity and low-latency buffer for fast data selection, but also opens a new approach in high-bandwidth data acquisition by decoupling the lifetime of the data analysis processes from the changing event rate due to the duty cycle of the data source. This is achievable by the option to extend its capacity even up to hundreds of petabytes to store hours of an experiment's data. Our initial performance evaluation shows that DAQDB is a promising alternative to generic database solutions for the high luminosity upgrades of the LHC at CERN.

1 Introduction

Data acquisition (DAQ) systems are responsible for collecting, transforming and recording data representing physical signals. Their source can be a wide variety of instruments like sensors, antennas, or telescopes. In many cases, it is not feasible to store all the data for the required capacity. For this reason, an online filtering system selects the relevant pieces of information according to the experiment's goal and only relevant events are permanently saved. Examples of such large-scale systems are the detectors designed to study particle collisions at the Large Hadron Collider (LHC) at CERN. They count millions of different sensors producing data.

The online filtering system typically consists of a fast hardware trigger and a software-based high-level event selection running on a large computing farm. Its size is dictated by

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

the complexity of the selection algorithms and the rate of the incoming data. Transport and buffering layer between detector readout nodes and assembly/selection nodes are typically governed by experiment-dedicated software frameworks. The buffers at the readout nodes can typically store up to few seconds of data due to the high rates of the experiments. This is due to the capacity constraints and high costs of DRAMs. Other storage media, such as solid state drives (SSDs), cannot be considered because their performance and endurance would not meet the requirements of high-bandwidth experiments. As result, data-readout and data-selection subsystems are tightly coupled. The pressure on the readout buffers will expand even more with the next LHC upgrades, for which the data rates will continue to increase reaching 5 TB/s [1]. Furthermore, LHC experiments require to store selected events in the online data acquisition system for several days for security reasons: the data acquisition must reliably store dozens of petabytes. In summary, there is a need for a generic solution to handle data in high-bandwidth DAQ systems at petascale capacity.

The remainder of this paper is organized as follows. We first overview the current state of the art in Section 2 and present an overview of a traditional approach to data acquisition at the LHC experiments. We next propose and discuss the advantages of a new approach — logical event building with hot storage. Our solution, a distributed key-value store for high-bandwidth DAQ systems, DAQDB, and initial performance are introduced in Section 4. We conclude our work in Section 5.

2 Relevant work

High-bandwidth DAQ systems have storage requirements which can exceed capacities and possibilities of available solutions [1, 2].

Relational databases [3] are limited by the information volume: when the amount of data increases, the query execution time can become slow [4]. The NoSQL data stores overcome this problem by limiting operations and taking advantage of horizontal scaling. Nowadays, more than 200 different NoSQL stores exist [5] and they are mainly categorized into key-value, wide column, document, and graph stores. In the Key-Value Store (KVS), data are represented as pairs of key and value, where the key is unique. The pair is stored in key-based look-up structures [6]. They are the best candidate for the event-driven DAQ systems. Redis [7], a widespread in-memory KVS, is used as database, cache and message broker and is exploited by many companies [8]. It offers good performance but suffers from the capacity limits of DRAM memories like the current readout buffers in DAQ.

Despite a large number of NoSQL stores, to the best of our knowledge, the existing systems do not satisfy the requirements of event-driven DAQ systems with enough performance.

3 Event building

Detectors in physics experiments usually consist of numerous different sensors. They measure different quantities of the same physical studied phenomenon. Assembling the different sensors' data corresponding to the same physical event, e.g., a specific particle collision, is essential to analyze meaningful information. This process is called event building, and it is part of the DAQ system.

3.1 Physical event building

The physical event building process aggregates information in a single, contiguous storage area, usually in RAM or in a file on a storage device. The data are fetched over a DAQ network from readout nodes.

The ATLAS experiment at the LHC implements physical event building in a two-step process: fragments are transferred from the detector to the readout system (ROS) via point-to-point optical links connected to FPGA-based PCIe cards. The data are buffered in the ROS and made available for the High-Level Trigger (HLT). The HLT processing unit collects and processes fragments of a given event. A data collection manager (DCM) dynamically distributes the available resources and orchestrates all the data flow process [9].

Another LHC experiment, CMS, implements physical event building in a similar strategy, the event builder assembles the fragments in the RAM, and a central entity supervises the allocation of available building resources to the events. The acquisition of all the data, from the detector to the selection of the interesting events are coupled. The DAQ system can buffer data up to 90 seconds before the HLT selection, and it can store only selected events for few days [10].

3.2 Logical event building with hot storage

The physical event building is the traditional approach, where data fragments are fetched explicitly over a network from temporary buffers at the readout nodes to a single physical location. Here, we propose a new approach: *logical event building with hot storage*. The fragments are stored in a large distributed key-value store. The event building process is done internally in the KVS: the processing units can retrieve or delete all the fragments belonging to one event with a single query. The entire data exchange is handled internally by the KVS: the participating nodes are not aware of the network topology.

Each object, event data fragment, is associated with a key. The readout system inserts the data into the KVS with a simple operation, *put(key, object)*. The consumer, e.g., a filtering node, can request a specific object using the associated key, *get(key)*. In the KVS, the key and the object are treated as an opaque array of bytes. The KVS is distributed: a hash function is applied to the key to generate an identifier, which is used to determine the responsible node. This approach has already shown its potential, for instance, Amazon uses it for their core web services [11].

3.2.1 Motivation

The proposed DAQ architecture brings several advantages. The readout systems and the filtering nodes use the KVS interface to insert and retrieve data without being aware of the network topology. Moreover, the logical event building with hot storage can increase the data taking efficiency by decoupling real-time data acquisition from event selection. During the experiments, there are *inter-fill* periods where data are not delivered to the DAQ system. During this time and at the end of the data taking, the processing units are underutilized or even left idle. The KVS can be used as a large buffer to store events. It would allow to have less strict processing timeouts and/or a better sensor calibration, which implies a purer selection of events.

3.2.2 Challenges

There are many challenges to overcome, but the main ones are the total *request rate* and *capacity* of the distributed KVS. The KVS must satisfy the requirements of the different experiments [1, 2] collecting terabytes of data per second and store hundreds of petabytes. The total number of readout nodes also changes across different experiments, but generally is in the order of hundreds of servers [1]. The rate of put requests can reach up to 1 MHz, with the value size in the range of 1 kB to 10 kB depending on the subdetector. On the other side of

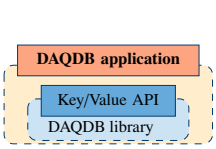

Figure 1. Single-node DAQDB application.

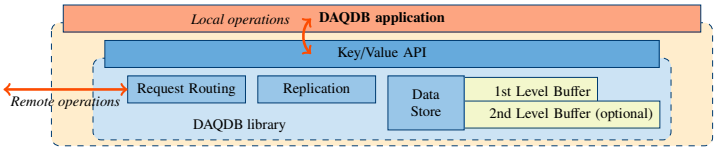

Figure 2. High-level design of DAQDB.

the DAQ system, the filtering farm, there can be hundreds of thousands of clients accessing the KVS for all or partial event data in parallel.

The *key* identifies a specific fragment. 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.

A *processing unit* needs to retrieve some or all the fragments of a specific event from the KVS. In the existing DAQ systems, as described in Section 3.1, the supervisor assigns an event that has not yet been processed to one of them. The KVS should provide the ability to directly select an unprocessed event.

The KVS should store data for hours and provide a *reliable service*: it should not lose unprocessed data event in case of failures. This can be realized saving data on persistent memory on each node and replicating the information over multiple machines.

4 DAQDB

From all the KVS projects available, as described in Section 2, none was designed with the challenges from the previous section in mind. Here, we propose DAQDB [12] — a distributed key-value store for high-bandwidth data acquisition systems. The key design choices are made to optimize the data flow of a DAQ system and using the best potential of emerging technologies.

4.1 High-level design

DAQDB is a dynamic user-space library, see Figure 1. Its high-level internal design is presented in Figure 2. The library can be used in various operation modes, for which some of the features described below are optional. The details behind that and the associated deployment models are discussed in Section 4.2.

Key/Value API The library exposes API that provides typical operations for data stores, like `Get(key, options)`, `Put(key, value, options)`, `Update(key, value, options)` and `Remove(key)`. The key is used to route the requests to the appropriate DAQDB nodes which are handled by the library internally. The key, in contrast to the generic strings typically used, is composed of multiple sub-keys: operations on a range of keys are more efficient avoiding costly parsing of strings. In the range operations, the user can define lower and upper boundaries for each sub-key. This is important for filtering nodes that request a subset of data from specific detectors.

The structure of the key can be defined in the configuration of DAQDB, and one of the sub-keys needs to be specified as *primary key*. This represents a unique identifier for a set of key-value pairs, e.g., an entire physics event. There is a set of attributes assigned to each primary key, e.g., replication scheme or distributed lock. These attributes are kept in metadata attached to each key in the data store.

Data store DAQDB offers two buffer levels as presented in Figure 2. The first level provides high bandwidth and terascale capacity, whereas the second one extends the storage capacities to petascale at the cost of lower bandwidth. For each key, it is possible to decide which buffer to use, this helps in addressing the traditional challenge of storage systems, where performances are traded for capacity. This feature is crucial for DAQ systems where only a small part of the data is used for fast data rejection. In this case, the incoming data are stored in the high-bandwidth buffer, whereas the high-capacity buffer stores pre-accepted events that are waiting for the final selection. Nevertheless, performance and capacity requirements for both buffers, in the context of the LHC upgrades, are still challenging and can be difficult to meet by technologies available today. The technological advances that DAQDB is building on are discussed in Section 4.3.

Distributed lock The nodes performing data selection need to access events, not yet processed, in typical DAQ systems. In most of the cases, a central supervisor manages the assignment of events, becoming a single point of failure. DAQDB is designed to build fully distributed systems implementing a distributed lock mechanism. DAQDB provides a `GetAny(options)` command that retrieves any primary key, e.g., an event, from the data store not yet locked. This mechanism is implemented with a global attribute, `LOCKED`. In this way, the DAQDB client is capable of acquiring exclusive access to the data of an event, not yet processed. Additionally, `options` can be used to control from which buffer level to retrieve the data, it allows to distinguish between events awaiting fast filtering or event building. If needed, the `LOCKED` attribute can be also controlled with the `Update(options)` call. This algorithm assumes that all data associated with a given primary key eventually arrive and must be accessible before data processing starts.

4.2 Deployment models

As explained previously, DAQDB is a library that can be embedded into applications in different operation modes.

Figure 3 shows the base model, in which readout applications use DAQDB with one or two levels of buffering. Here, the same nodes are used both for detector readout and storage. Fewer servers can be used, however, they need to be capable of running both workloads in parallel. Incoming data are inserted into the data store locally reducing the network load. The data processing applications access the information using the DAQDB library without any local buffering capabilities. Their requests are routed internally to appropriate DAQDB nodes on the readout side.

Figure 4 shows a different model, where an independent group of servers provides the storage functionality. Here, both readout and data processing applications embed DAQDB without any level of buffering. All their requests issued via the DAQDB interface are routed internally to storage nodes, which run standalone *thin-server* applications that provide storage capabilities. With this approach, DAQDB can optimally distribute the requests across the available pool of servers to improve the performances. However, in the system, more servers are needed and there is an increase in the network load.

4.3 Technology insights

The implementation of DAQDB is based on technological advantages given by the newest hardware and software capabilities to approach the challenges defined in Section 3.2.2.

Storage media DAQDB uses two-level buffering scheme designed to meet bandwidth and capacity requirements. The 1st level buffer is meant to act as a high-bandwidth buffer with

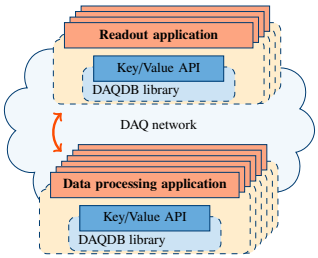

Figure 3. Base deployment model.

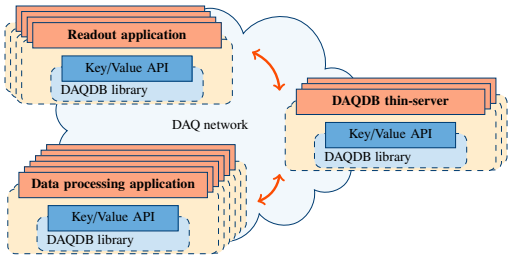

Figure 4. Deployment model with detached storage.

a terascale capacity. The underlying technology is byte-addressable and high-capacity *persistent memory* [13, 14]. It overcomes the limitations of DRAM mentioned in Section 1, providing the required performance. The current generation of Intel Optane DC Persistent Memory offers up to 3 TB of memory capacity per CPU socket. Assuming input data flow of 100 Gbit/s [1], a couple of minutes buffer per single CPU becomes feasible. To fully exploit the capabilities of persistent memory, it is necessary to use new programming paradigms. DAQDB is optimized by design for persistent memory and uses PMDK libraries [15] as interface.
The 2nd level buffer is optional and can be used to extend storage capabilities and/or decouple data readout from data selection. The medium behind is non-volatile storage media like SSDs accessed via PCIe (NVMe). From the software perspective, they are accessed directly from user-space with the software interface offered by SPDK libraries for performance reasons [16]. Deriving from available PCI lanes and disks capacities, DAQDB can offer a high-bandwidth petascale buffering solution. In case the 2nd level buffer is used, which can potentially handle hours of data, the data need to be persistently stored to prevent significant loss. Disk writes are considered persistent, while all the metadata are kept in the memory medium, thus usage of persistent memory in contrast to volatile DRAM is crucial. DAQDB is capable of delivering data and metadata persistency.

Data structure The number of writes is equal to or greater than the number of reads, so the data structure, built on top of the two levels of buffering, is optimized for write workloads. Furthermore, on the readout side, the system remains synchronized to a detector producing data at high rates requiring fast buffering capability, as described in Section 3.2.2. From the functionality perspective, there is the need for range query operations. For this reason, DAQDB uses a modified adaptive radix tree (ART) [17] structure. This structure provides O(k) time for accessing data at the cost of used capacity over regular binary trees. Capacity constraints driven from regular Radix Tries are addressed by ART, in contrast in DAQDB we have modified the ART algorithm to address even more savings in data structures volume by additional tree compression on branches with single child structure, that are common to large systems with constrained network configuration, and complex descriptive key structure.

Data partitioning If an operation has to be performed on a remote node, DAQDB internally handles routing requests. DAQDB implements a *zero-hop* distributed hash table (DHT) [11, 18]: each node maintains enough routing information to route a request directly to the target node. In this way, the latency is reduced by avoiding routing through multiple nodes, typical of many DHT systems, such as Chord [19].If the storage nodes are the same as detector readout nodes, data locality is generally desired to reduce the network load. This is achievable by replacing hash tables with user-configurable look-up tables.

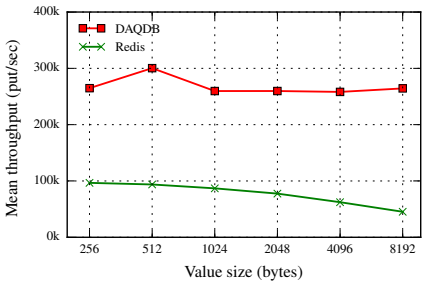

Figure 5. Operations per second of a single readout thread using DAQDB and Redis for different value sizes.

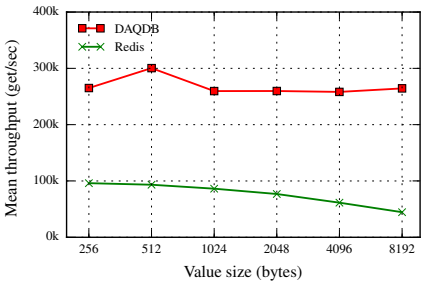

Figure 6. Operations per second of a single filtering thread using DAQDB and Redis for different value sizes.

Data transport In any distributed system, the transport layer is often a major performance constraint. To achieve high-performance, it is needed to use the newest technologies that reduce CPU latency and overhead, as well as reduce the network overhead. RDMA, in particular, provides mechanisms for accessing remote memory, including persistent memory, and therefore allows zero-copy operations on distributed data sets. It is well-known that current RDMA implementations do not scale well up to large clusters. The reason is that RDMA-capable NICs require cache for each connection state [20]. Considering the scale of the system described in Section 3.2.2 with hundreds of thousands of clients issuing requests to the KVS for event data in parallel, it becomes an essential aspect of DAQDB design. To address this, DAQDB uses eRPC [20] as its transport layer which provides scalability to a large number of nodes and CPU cores.

4.4 Single node performance

The first proof point for DAQDB is the initial performance evaluation in a simple scenario on a single node. There is one thread emulating a readout application, which executes put requests with a predefined value size. A second thread emulates a data filtering application. It retrieves the data inserted earlier by the readout thread and immediately deletes them from the store. The duration of a single test is thirty seconds. The goal is to prove that the initial implementation of the design presented in this paper provides expected levels of performance. In this version, DAQDB uses 24-bit radix trie, which ultimately will change to the adaptive radix trie described earlier. The S2600WF board with two Intel Xeon Gold 6140 CPU eighteen-core CPUs is used for this evaluation. Persistent memory is emulated with ramdisk (i.e., /dev/shm) using 80 GB out of 96 GB of available memory.

Figure 5 and Figure 6 show that DAQDB performs better than a well known single-node KVS, Redis [7], used as a reference. It is evaluated in similar scenario with parallel put and get operations using *redis-benchmark* with the same ramdisk emulation of persistent memory [21]. Volatile allocations are used for both data structure and values in contrast to DAQDB which keeps data structure persistent. On the other hand, it must be also taken into account that in this scenario DAQDB is an embedded data store in contrast to Redis which requires inter-process communication between the store and the benchmark application. Still, DAQDB maintains high mean throughput just below 300 KOPS in the entire range being around three times more than Redis. The minimal value of 241 KOPS and maximum standard deviation of 7.6 KOPS prove also stable performance over time. Although this performance evaluation is preliminary, it is already apparent that a dedicated KVS for DAQ systems is a promising approach.

5 Conclusion

In this paper, we present how emerging technologies like persistent memory, NVMe SSDs, scalable networking, and scalable data structures make it now possible to design a key-value store for high-bandwidth data acquisition systems. On the one hand, the requirements of the high luminosity upgrades of the LHC can be met by a proper combination of those technologies. On the other hand, the use of KVS is a novel approach for data acquisition and opens up new perspectives for DAQ system designers. Firstly, DAQDB hides away many physical aspects usually needed for experiment's data assembly abstracting it with a logical interface. Secondly, it allows to substantially increase the temporary buffer for the incoming data to reduce the tight coupling of the data readout and data processing subsystems. Our initial performance evaluation proves that DAQDB takes a valid path. The future work will focus on proving DAQDB's scalability in large DAQ farms.

References

[1] The ATLAS Collaboration, Tech. rep., CERN, Geneva (2018), `http://cds.cern.ch/record/2285584`

[2] J. Andre, U. Behrens, J. Branson, P. Brummer, O. Chaze, S. Cittolin, C. Contescu, B. Craigs, G. Darlea, C. Deldicque et al., *The CMS Data Acquisition-Architectures for the Phase-2 Upgrade*, in *J Phys Conf Ser* (IOP Publishing, 2017)

[3] E.F. Codd, Communications of the ACM **13**, 377 (1970)

[4] N. Leavitt, Computer **43** (2010)

[5] A. Davoudian, L. Chen, M. Liu, ACM Computing Surveys **51**, 1 (2018)

[6] R. Cattell, ACM SIGMOD Record **39**, 12 (2011)

[7] *Redis*, `https://redis.io/`

[8] *Who's using Redis?*, `https://redis.io/topics/whos-using-redis`

[9] M.E. Pozo Astigarraga, Tech. rep., ATL-COM-DAQ-2017-149 (2017)

[10] J. Andre, U. Behrens, J. Branson, P. Brummer, O. Chaze, S. Cittolin, C. Contescu, B. Craigs, G. Darlea, C. Deldicque et al., *Performance of the CMS Event Builder*, in *J Phys Conf Ser* (IOP Publishing, 2017), 3, p. 032020

[11] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, SIGOPS Oper. Syst. Rev. **41**, 205 (2007)

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

[13] A. Rudoff, Login: The Usenix Magazine **42**, 34 (2017)

[14] V.J. Marathe, M. Seltzer, S. Byan, T. Harris, *Persistent memcached: Bringing legacy code to byte-addressable persistent memory*, in *9th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 17)* (2017)

[15] *pmem.io: PMDK*, `http://pmem.io/pmdk/`

[16] *Storage Performance Development Kit*, `http://www.spdk.io/`

[17] 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

[18] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, I. Raicu, *ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table*, in *IPDPS'13* (2013), pp. 775–787

[19] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, H. Balakrishnan, IEEE/ACM Transactions on Networking (TON) **11**, 17 (2003)

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

[21] *Redis, enhanced to use PMDK's libpmemobj*, `https://github.com/pmem/redis/`