Belle II Conditions Database

M Ritter[1], L Wood[2], T Kuhr[1], M Bracko[3], T Elsethagen[2], K Fox[2], J Hall[2], C Pulvermacher[4] B Raju[2], M Schram[2], E Stephan[2],

[1] Ludwig-Maximilians University Munich, Excellence Cluster Universe, Boltzmannstr. 2, 85748 Garching, Germany
[2] Pacific Northwest National Laboratory, 902 Battelle Boulevard, Richland, WA, USA
[3]Jožef Stefan Institute, Jamova 39, 1000 Ljubljana, Slovenia
[4]High Energy Accelerator Research Organization (KEK), 1-1 Oho, Tsukuba 305-0801, Ibaraki, Japan

E-mail: Martin.Ritter@lmu.de, Lynn.Wood@pnnl.gov

Abstract. The Belle II experiment at KEK is preparing for taking first collision data in early 2018. For the success of the experiment it is essential to have information about varying conditions available to systems worldwide in a fast and efficient manner that is straightforward for both the user and maintainer. The Belle II Conditions Database was designed to make maintenance as easy as possible. To this end, a HTTP REST service was developed with industry-standard tools such as Swagger for the API interface development, Payara for the Java EE application server, and the Hazelcast in-memory data grid for support of scalable caching as well as transparent distribution of the service across multiple sites.

On the client side, the online and offline software has to be able to obtain conditions data from the Belle II Conditions Database in a robust and reliable way under very different situations. As such the client side interface to the Belle II Conditions Database has been designed with a variety of access mechanisms which allow the software to be used with and without an internet connection. Different methods to access the payload information are implemented to allow for a high level of customization per site and to simplify testing of new payloads locally. Changes to the conditions data are usually handled transparently but users can actively check whether an object has changed or register callback functions to be called whenever a conditions data object is updated. In addition a command line user interface has been developed to simplify inspection and modification of the database contents.

1. Introduction

The Belle II detector [1] and the SuperKEKB accelerator are currently under construction at the KEK laboratory in Tsukuba, Japan. The aim of this next generation B factory experiment is to collect 50 times more data than its predecessor Belle [2] and to use this data to search for new physics in a variety of B meson, charm hadron, or τ lepton decays with unprecedented precision. This requires detailed information on varying calibration and detector conditions to be available when analyzing the data.

The Belle II software Framework (basf2) [3] is a C++/Python framework to process the events recorded by the Belle II detector. Events are grouped in so-called runs which mark a data-taking period with stable operating conditions. Recorded events are processed one by one using a sequential set of algorithms called "modules." As all events are independent from each

other we can benefit from current architectures with multiple cores by processing different events in parallel.

The Belle II Conditions Database is designed as a representational state transfer (REST) service. Communication is performed by standard HTTP using XML and JSON data. The database manages conditions data on run granularity. Binary objects called payloads which are defined for a certain interval of validity (IoV) are grouped into so-called global tags. The conditions database is by design agnostic to the contents of the payloads and only identifies them by name and revision number. The integrity of all payloads is verified using a checksum of the full content.

One requirement for the conditions client interface is that it needs to also work inside the data acquisition network which does not have a network connection to the outside. To facilitate this the client can also use a local, file-based backend which reads the list of payloads from an index file and the payload contents from files in a given directory. Command line tools written in Python allow easy downloading of all payloads and IoVs defined in a global tag to be used locally or uploading a locally prepared folder to be available for all users.

2. Server side

2.1. Implementation

The current configuration of the Belle II conditions database back-end is shown in Figure 1. The left and right sides represent two servers, one of which supports the REST API and database, while the other provides access to the payload data files. Each component in the figure above is implemented using Docker [4], a software framework that supports virtualization of user-space instances as opposed to full virtual machines, which reduces resource requirements, provides auto-restart functionality for each component separately, and provides a basis for scalability by instantiating multiple identical containers.

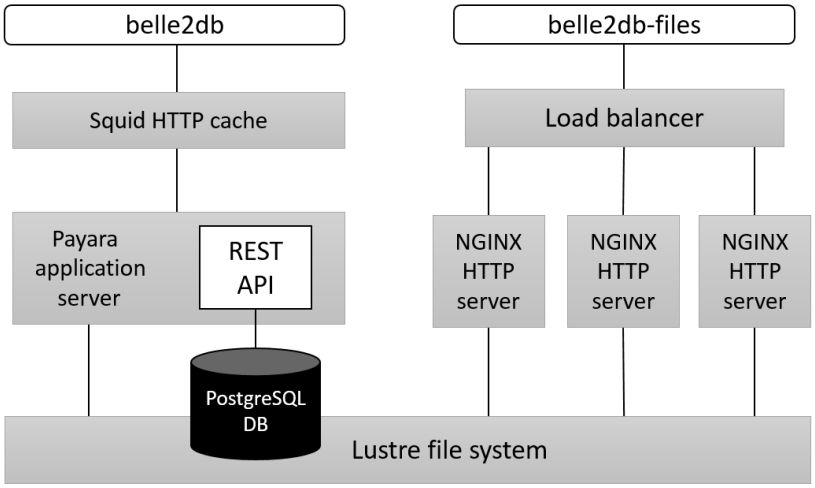

Figure 1. Current implementation of the conditions database back-end. Database and payload file access are handled separately by separate services.

Communication with the database uses standard HTTP using XML or JSON data. The choice of a standardized REST API makes the client coding independent of the actual database

implementation details, and allows for easy caching support using a Squid [5] HTTP cache proxy server to reduce load on the API and database. The database itself is PostgreSQL, but the schema and procedures have been kept generic so other database applications could be used if necessary. To keep the database small, the payloads only consist of references to files on the separate payload file server. The payload files are currently assumed to be ROOT objects by the client, although there is nothing in the database design that limits the data type. The REST API interface is built from two commercially-available applications. Payara Micro [6] is a Java EE application server optimized for production operations in a containerized infrastructure such as Docker. Hazelcast [7] is a Java-based in-memory data grid. Data can be distributed evenly among separate nodes of a computer cluster, which provides horizontal scaling in both available storage space and processing power. This provides multiple benefits, including caching of frequently-used data in memory, transparent scalability, and load-balancing to reduce the query load on the database.

The file server currently consists of three NGINX [8] HTTP servers handled by a load balancer to distribute traffic evenly across the servers. The back-end file system is based on Lustre [9], an open-source parallel file system for high-performance computing environments. The Lustre file system is shared between all back-end instances.

2.2. Performance

The back-end has been evaluated with both direct access by Grid-based Belle II Monte Carlo campaigns and directed testing by the Belle II database group. The directed testing was done using Gatling [10], an open-source load stress testing tool for HTTP servers which allows custom test design through scripting. Testing of the database back-end was implemented by monitoring the usage patterns during Monte Carlo operations and then writing separate tests for the REST API and file server. The use of scripting allowed for much higher stress loads than was readily available from Grid-based testing. Recent performance results are shown in Figure 2. The top graph shows the database server responding successfully to 80 requests per second, while the bottom graph shows the payload file server responding to 180 file requests per second, with up to 10,000 open connections being handled simultaneously by the load-balanced HTTP servers. These tests correspond to nearly half of the payload bandwidth for the expected full-scale Belle II grid-based analysis cloud of 100,000 active nodes.

2.3. Future Improvements

Several improvements to the Belle II Conditions Database are in progress: Hazelcast supports cache clustering between remote sites, and will be evaluated as a means of supporting more localized database access worldwide. Placing portions of the cache at three or more sites will provide faster response times as well as backup capability for site or network outages. The PostgreSQL database is still a single-site instance and not currently scalable. Several options of supporting a distributed database are being investigated, including OpenStack Trove [11] and CockroachDB [12]. The replicated databases would be sited in tandem with the Hazelcast cache cluster sites. Authentication is currently not implemented, but is planned for services that would modify the database. The possibility of leveraging the X.509 authentication already present in the Belle II Grid computing interface is being investigated.

3. Client side

The choice of a standardized REST API makes the client implementation independent of the actual database implementation details and allows for a simple and flexible implementation of clients in different programming languages.

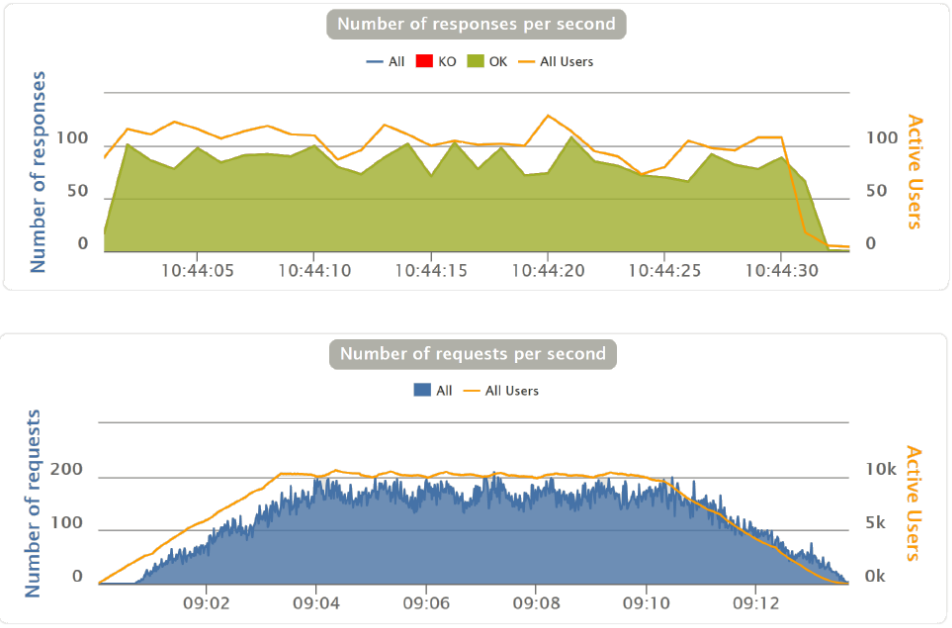

Figure 2. Performance results. Top graph shows Gatling stress testing of the database server; bottom graph show testing of the payload file server.

3.1. Access of Conditions Objects

The software framework assumes that payload contents are serialized ROOT [13] objects. User access to conditions objects is provided using two interface classes, one for single objects called `DBObjPtr` and one class for arrays of objects called `DBArray`. These class references payload objects, so called `DBEntry` objects in a global store, the `DBStore`. Multiple instances of the interface class all point to the same object. Access to the conditions objects is available in C++ and in Python where the class interface has been designed to be as close as possible to the already existing interface for event level data. Users familiar with the event level storage should have few problems accessing conditions data.

The interface classes always point to correct payload object for the current run, updates are transparent to the user. If the user needs to be aware when the object changes they can either manually check for changes or register a callback function to be notified on change. Figure 3 visualizes the relations between all the entities.

The objects in the global store are updated at the beginning of each run if needed. To allow for conditions which change inside of one run the client supports a so-called intra run granularity: Payload objects which inherit from an abstract base class `IntraRunDependency` can contain multiple objects for different parts of the run. The correct one will be selected transparently depending on certain criteria like event number or timestamp.

3.2. Creation of Conditions Data

To create payload objects we also provide two interface classes to simplify the procedure of preparing the objects, serializing them and committing them to the database. Users can just instantiate one of the creation classes, add objects to them and commit them to the configured database with a user supplied IoV. This includes support for intra run dependency. The possibility for a local file based database allows for easy preparation and validation of payloads as is needed during the calibration of the detector.

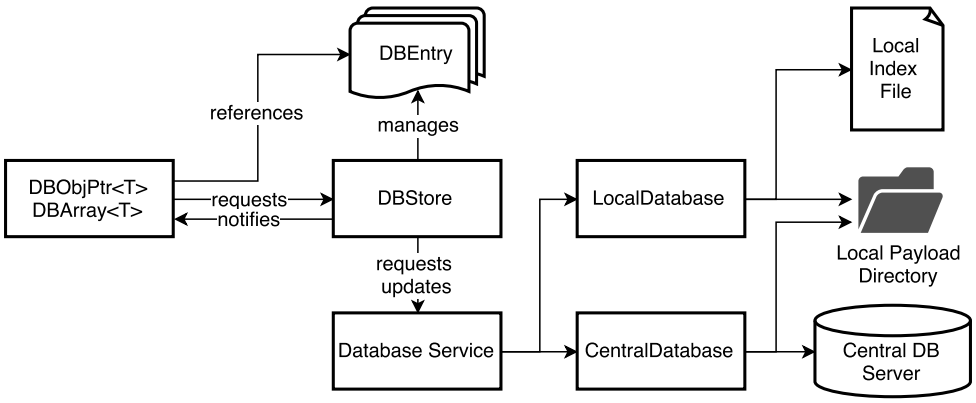

Figure 3. Relationships for the Conditions Database Interface. The user only interacts with the `DBObjPtr` and `DBArray` classes, everything else is handled transparently and can be configured independently.

Once all payloads are verified they can be uploaded to the central database using a standalone command line client developed in python. In addition this command line client allows inspection, modification and download of parts of the database contents.

3.3. Storage backends

Payloads in the central database are represented by logical filename, revision number and content checksum. While the central server provides a way to download all payloads directly via HTTP this is not the only possible distribution system. As the payloads are usually a large number of small files it might be beneficial to investigate alternative ways to distribute them. The database client can easily accommodate different storage implementations for the payloads by defining a cascade of storage backends to look for the actual payload content. After downloading the metadata on all needed payloads it will check all configured backends one by one for the needed files. If no storage backend contains the required payload it will automatically try to directly download the payload from the central server.

Currently only file based storage, for example shared filesystems or Cern-VM FS [14] are implemented. However it would be trivial to extend this to other technologies, for example key-value based storage systems like MICA [15]. Also hybrid systems where multiple payload objects are consolidated into larger archives similar to git packfiles [16] could be considered as well. This allows for a highly flexible distribution of the needed payload files which can be adapted as the content of the conditions database evolves.

The fallback solution to simply download missing payloads over HTTP makes it trivial to use the software without the need for explicit payload storage configuration.

4. Summary

Access to conditions data is a crucial part of the data processing. We have implemented a high level interface to the Belle II Conditions Database using a REST API. Access for the users is kept as simple as possible by requiring them just to instantiate one interface class to gain access to the correct data for any given event.

Distribution of the actual payload contents is highly flexible and can be expanded or configured easily. Any missing payloads will automatically be downloaded via HTTP so no special setup is required.

Acknowledgements
Portions of this work were carried out for the U.S. Department of Energy under Contract DE AC05 76RL01830 PNNL-SA-121968.

References

[1] Abe T *et al.* 2010 Belle II Technical Design Report *Preprint* arXiv:1011.0352
[2] Abashian A *et al.* 2000 The Belle Detector *Nucl. Instrum. Meth. A* **479** 117
[3] A Moll 2011 The Software Framework of the Belle II Experiment *J. Phys.: Conf. Series* **331** 032024
[4] Docker Platform, "Docker" [software], available from http://www.docker.com/ [accessed 2017-10-26]
[5] Squid Caching Proxy, "Squid" [software], available from http://www.squid-cache.org/ [accessed 2017-10-26]
[6] Payara Micro Server, "Payara Micro" [software], available from http://www.payara.fish/payara_micro [accessed 2017-10-26]
[7] Hazelcast In-Memory Data Grid, "Hazelcast" [software], available from http://www.hazelcast.com [accessed 2017-10-26]
[8] NGINX HTTP Server, "NGINX" [software], available from http://www.nginx.com [accessed 2017-10-26]
[9] Lustre parallel HPC file system, "Lustre" [software], available from http://www.lustre.org [accessed 2017-10-26]
[10] Gatling Load and Performance Test Tool, "Gatling" [software], available from http://www.gatling.io [accessed 2017-10-26]
[11] OpenStack Trove Database as a Service, "OpenStack Trove" [software], available from http://www.openstack.org/software/releases/ocata/components/trove [accessed 2017-10-26]
[12] CockroachDB Cloud-Based Database, "CockroachDB" [software], available from http://www.cockroachlabs.com [accessed 2017-10-26]
[13] Brun R and Rademakers F 1997 ROOT - An Object Oriented Data Analysis Framework, *Nucl. Instrum. Meth. A* **389** 81. See also "ROOT [software], Release v6.08.06, doi:10.5281/zenodo.848819
[14] P Buncic, C Aguado Sanchez, J Blomer, L Franco, A Harutyunian, P Mato and Y Yao 2010 CernVM a virtual software appliance for LHC applications *J. Phys.: Conf. Series* **219** 042003
[15] H Lim, D Han, D Andersen and M Kaminsky 2014 *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle: USENIX Association) pp 429–444
[16] Git Project, "Git" [software], available from https://www.git-scm.com [accessed 2017-10-23]