

A Conditions Data Management System for HEP Experiments

P J Laycock¹, D Dykstra², A Formica³, G Govi², A Pfeiffer¹, S Roe¹, R Sipos⁴

¹ CERN, ² Fermi National Accelerator Lab, ³ CEA Saclay, ⁴ Eotvos Lorand University

E-mail: paul.james.laycock@cern.ch

Abstract. Conditions data infrastructure for both ATLAS and CMS have to deal with the management of several Terabytes of data. Distributed computing access to this data requires particular care and attention to manage request-rates of up to several tens of kHz. Thanks to the large overlap in use cases and requirements, ATLAS and CMS have worked towards a common solution for conditions data management with the aim of using this design for data-taking in Run 3. In the meantime other experiments, including NA62, have expressed an interest in this cross-experiment initiative. For experiments with a smaller payload volume and complexity, there is particular interest in simplifying the payload storage. The conditions data management model is implemented in a small set of relational database tables. A prototype access toolkit consisting of an intermediate web server has been implemented, using standard technologies available in the Java community. Access is provided through a set of REST services for which the API has been described in a generic way using standard Open API specifications, implemented in Swagger. Such a solution allows the automatic generation of client code and server stubs and further allows changes in the backend technology transparently. An important advantage of using a REST API for conditions access is the possibility of caching identical URLs, addressing one of the biggest challenges that large distributed computing solutions impose on conditions data access, avoiding direct DB access by means of standard web proxy solutions.

1. Conditions in HEP experiments

Broadly speaking, conditions data is defined as the non-event data required by data-processing software to correctly simulate, digitise or reconstruct the raw detector event data. The non-event data required to maintain, operate and optimise detectors can be broken down into the following categories:

- (i) Configuration
- (ii) Detector Control System (DCS) [copied from Process Visualization and Control System (PVSS)]
- (iii) Monitoring
- (iv) Calibrations, alignments

Conditions data thus largely consists of (iv), together with the subset of (i) and (ii) that are required for data processing. Other non-event data may also be required for data processing, for example machine parameters, and thus this list only attempts to give an indication of the scope.

In practice, any non-event data from any source that are required for optimal data processing can be considered as conditions data. In general, conditions data vary with time but with a granularity much coarser than the event, ranging from one year to one run, down to a granularity of the order of one minute for a small subset.

1.1. Workflows

Given the definition of conditions data, the workflows that need to be supported are predominantly those of offline data processing. However, both CMS and ATLAS high level (software) triggers use the same software framework as offline data reconstruction and must also be supported, as must any workflow that requires conditions data.

A non-exhaustive list of important workflows follows:

- (i) Subsystem calibration: conditions determination and testing (including the ability to access a copy of the conditions stored locally); uploading conditions to the production database.
- (ii) Prompt data processing, with conditions updates adhering to strict protocols
- (iii) Offline data processing, including Monte Carlo simulation, using pre-determined conditions.

1.2. Data volumes, read and write rates

Conditions data tend to scale in a controlled way during the lifetime of an experiment, typically producing data volumes of Gigabyte to Terabyte scale. It is worth noting that on ATLAS, where the same database instance was used for conditions data as well as the DCS and trigger data, the DCS and trigger data dominated the offline and online database instances, respectively. It is therefore strongly recommended by this working group to factorise conditions data from other use cases, even where these are related. Typically write-rates for conditions data must support of the order of 1 Hz (to ensure good support for several independent systems writing conditions data every minute), with the majority of conditions data being updated much less frequently than this. On the other hand, read-rates up to several kHz must be supported for distributed computing workflows, where thousands of jobs needing the same conditions data may start up at the same time. Conditions data are typically written once and read frequently.

2. Conditions Database Archetype

The archetypal solution to a conditions database management system (CDMS) is shown below. The conditions data payloads are stored in a master database and are accessed using a client-server design through a REST interface. All of the experiments gave feedback that achieving a high degree of separation between client and server was very desirable. Due to the read-rate requirements, caching is extremely important and good experience was seen when using web-proxy caches, e.g. the Squid cache shown here. Some key design principles are detailed in the following.

2.1. Payload technology

Experiments will inevitably choose their favourite payload technology. This working group recommends placing most emphasis on homogeneity and long-term maintenance when making this choice. Inhomogeneity and home-grown solutions all place additional burden on projects that typically lack the resources to support this after the initial build and commissioning phase of an experiment. CMS has very good experience of removing choice and only supporting boost-serialised C++ objects, with all classes belonging to one package in the CMSSW framework. Such a strategy lends itself more readily to long-term maintenance and minimises hurdles to data preservation. It is noted that there are payload formats used more in industry which would lend themselves more to higher-level functionality without the need of the software framework,

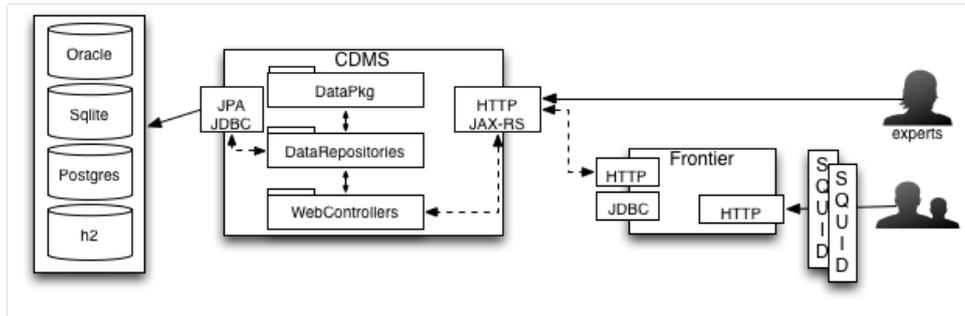


Figure 1. The conditions database archetype described in the text.

but, while this is attractive, the choice of format tends to be driven by software framework developers.

2.2. Database backend

One of the key features of the design is that it is agnostic to particular choices of database backend. This was also one of the key features of COOL, but due to the lack of caching in COOL, the queries themselves had to become more complicated. Thus on ATLAS, which uses an Oracle back-end, a significant amount of Oracle DBA effort was required to tune the queries and make them performant. It is therefore important to realise that real flexibility with respect to choices in database backends only comes when the system as a whole is simplified.

2.3. Client-side requirements

The client layer should be as simple as possible and should be as agnostic of the rest of the architecture as it is possible to be in order to improve maintainability. The database insertion tools in particular benefit from adopting a simple e.g. REST interface. The client layer needs to take care of payload deserialisation, as the remaining architectural components will deal with serialised objects. Experience also shows that clients should be able to manage multiple proxies and servers to provide robustness against server failures.

2.4. Caching layer

CMS and ATLAS require an intermediate layer between the client and server to provide caching capabilities. Considering the distributed use case, where thousands of jobs start at the same time and require the same conditions data, this is a clear requirement and one that can be well met using web-proxy technologies. An alternative solution would be to use a distributed filesystem with good caching capabilities, and the ALICE experiment is gaining experience using cvmfs. The simplicity of this solution makes it very attractive and it has thus been adopted as a strategy by the NA62 experiment. This in turn suggests that, as a design requirement, it should be possible to represent a conditions database on a filesystem. The primary challenge is to make the filesystem mapping use the cvmfs caching layers efficiently. Several experiments also have experience using SQLite replicas which are attractive for workflows where the exact subset of conditions is known in advance (some MC workflows), but a performant caching layer is preferable for general use cases.

2.5. Data Model

The data model for conditions data management is an area where the experiments have converged on something like a best practice. The model is shown in the figure below. A global

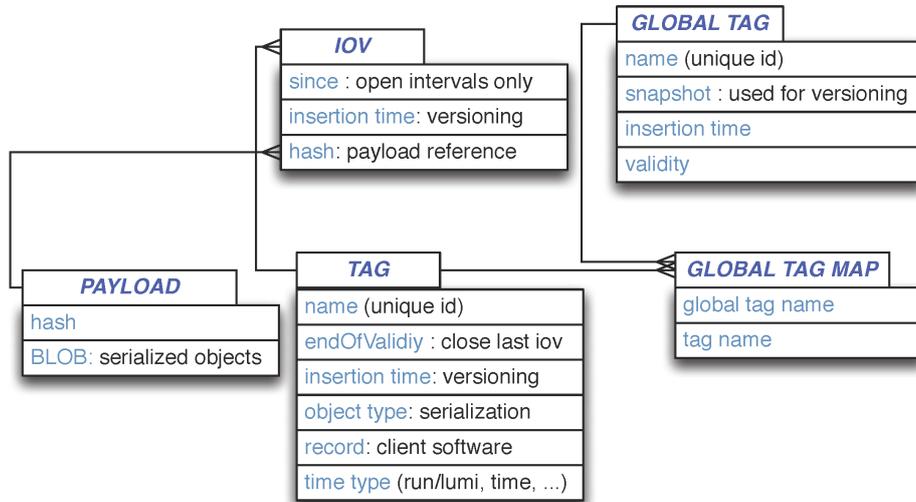


Figure 2. The data model for conditions data management described in the text.

tag is the top-level configuration of all conditions data. For a given system and a given interval of validity, a global tag will resolve to one, and only one, conditions data payload. The Global Tag resolves to a particular system Tag via the Global Tag Map table. A system Tag consists of many intervals of validity or entries in the IOV table. Finally, each entry in the IOV table maps to a payload via its unique hash key in the Payload table. A relational database is a good choice for this design.

This design has several key features. Firstly, conditions data payloads are uniquely identified by a hash which is the sole reference to any given conditions data payload. The payload data has been separated from the data management metadata and could in principle be placed in a separate storage system. This could also be important for data preservation, as the entire metadata component will occupy a trivial data volume and could exist in e.g. an SQLite file, while the payload storage could be handled separately. Secondly, IOVs are resolved independently of payloads and are also cacheable. Efficient caching is a key design requirement for any conditions database that must support high rate data access.

2.6. A git-based approach

For workflows which are completely offline and asynchronous with respect to data-taking, LHCb has adopted a different approach. Using git as the versioning system, conditions are placed in a directory structure, one for each type of condition. A file is used to map timestamps to payload files, and a simple format is used to allow a level of indirection to improve performance. With the algorithm used, payload look-up on timestamp is linear within the file, so large numbers of IOVs cause performance issues; the file format allows a timestamp to point either directly to a payload file or to a directory, thus allowing partitioning of the lookup. Versioning is then taken care of by creating a git tag, equivalent to a global tag.

3. Future and roadmap

The Conditions Data Management System described here is expected to meet the needs of CMS and ATLAS into the HL-LHC era [1]. Data volumes are not expected to exceed a Terabyte per year, and the rate of requests (determined by the computing resources of the experiments)

are expected to peak at tens of kHz. In other words, assuming the solutions presented here work in Run 3, they are also expected to work beyond that. The most important issue for those experiments then will be maintenance and operation in the face of evolving hardware and infrastructure, which makes consolidation to a simple and modular design, such as that presented here, crucial for the experiments' continued success. Full scale tests of this design are expected in the coming years, but based on experience with similar systems (the current CMS approach) the outlook is positive. Nevertheless, there are still open questions that need to be addressed. The functionality to produce a filesystem-based replica of a conditions database, that can be accessed transparently to the client, needs to be prototyped. The NA62 experiment will look at this problem in reverse, moving from a filesystem-based approach to a conditions database. This requires first modifying the filesystem-based infrastructure so that it can later be migrated to a database. Meanwhile, LHCb and ALICE will move to only applying calibrations promptly on a time-scale of Run 3, effectively making them conditions-free for offline workflows. This is completely driven by the data rates, the data-volume output to offline workflows must be significantly reduced and that can only be achieved if the only data output is already fully reconstructed. While this clearly leaves those experiments exposed to potential data quality losses, the benefit in terms of data processing is equally obvious.

3.1. Conditions for analysis

Despite the plans for LHCb and ALICE to be effectively conditions-free, there will nevertheless be higher-level calibrations applied to the physics data, and this will require some management. Belle II expects that this will be part of the analysis model from the outset, and there is wider interest in the analysis community in using systems like those described here to deliver analysis conditions data. Given the plans of Belle II, and the growing interest in the analysis community, feedback on the suitability of these solutions is expected within the next few years.

4. Conclusion

Following a decade of experience with the LHC experiments, conditions data management is an important component of HEP software and one that is often considered relatively late in the software design cycle. While the data volumes are easily accommodated by several database technologies, the workflows can be demanding. In particular, access rates at the level of tens of kHz is critical (and non-trivial) to support, and the importance of a caching system like Frontier [2] or CVMFS [3] for CMS and ATLAS cannot be overstated. Several experiments have converged on similar solutions for their plans for solving conditions data management problems. All of the experiments want a high degree of separation between the client and server side of the problem, with a client that is relatively simple (in contrast to the original COOL solution). An intermediate layer with caching capability is needed to support high rate requests and web-proxies have performed very well. REST interfaces and industry standard components, together with a relational data model to represent the global tag concept, can support the wide variety of workflows used in HEP while being sufficiently modular to evolve easily. For purely offline workflows, cvmfs is attractive and could be produced from the master database. This would resemble the git-based approach used by LHCb, where the IOV structure is encapsulated by a simple file. Analysis conditions data management could also benefit from using one of these solutions.

References

- [1] DB Futures Workshop: <https://indico.cern.ch/event/615499/timetable/?view=standard>
- [2] Frontier homepage: <http://frontier.cern.ch/>
- [3] CVMFS homepage: <https://cernvm.cern.ch/portal/filesystem>