EPJ Web of Conferences **214**, 04037 (2019) https://doi.org/10.1051/epjconf/201921404037
CHEP 2018

A Git-based Conditions Database backend for LHCb

Marco Clemencic[1,*] on behalf of the LHCb Collaboration

[1]CERN

Abstract. LHCb has been using the CERN/IT developed Conditions Database library COOL for several years, during LHC Run 1 and Run 2. With the opportunity window of the second long shutdown of LHC, in preparation for Run 3 and the upgraded LHCb detector, we decided to investigate alternatives to COOL as Conditions Database backend. In particular, given our conditions and detector description data model, we investigated the possibility of reusing the internal Git repository database as a conditions storage, and we adopted it since 2017 data taking. The adoption of Git gave us improved performance, smaller storage size and simplified maintenance and deployment.
In this paper we describe the implementation of our Git Conditions Database and the way it simplified our detector description and conditions development workflow.

1 Introduction

LHCb is one of the four main experiments on the Large Hadron Collider (LHC) at CERN. As for any other High Energy Physics experiment, to correctly analyze the data collected by the detector, the software used by LHCb relies on the so called Conditions Database to keep track of the state of the detector during data taking periods.

For more than twelve years LHCb used the CERN/IT developed Conditions Database library COOL[1] to manage its Conditions Database content in SQLite[2] files, but, in the context of the modernization of its software, LHCb decided to investigate alternative solutions.

1.1 The Conditions Database concept

Conditions are usually defined as "time varying non-event data" that describe the state of the detector for any given time. They live in a three-dimensional space, where they are identified by positions on the three axes (see figure 1)

- *source*: identifier of the information (e.g. alignment of a specific detector, calibration parameter, ...)
- *time*: moment in time the condition value is valid for (e.g. GPS time of an event, run number, luminosity section, ...)
- *version*: iteration on the measured or computed value of the condition, for cases when data like alignments for a detector are recomputed with a new algorithm for the same times.

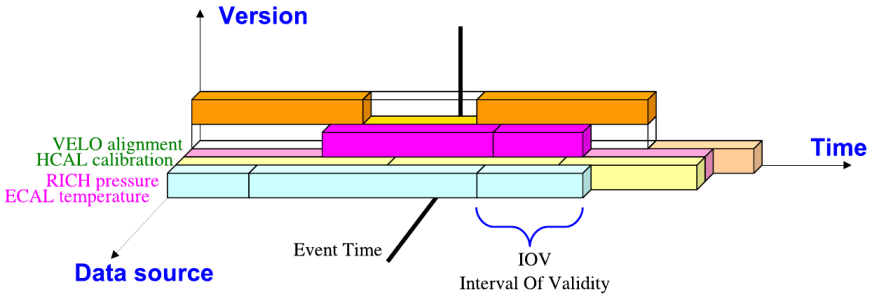

Figure 1. Representation of condition values as blocks in their three-dimensional space.

It must be noted that a condition value is not valid only for a single point in time, but for a range of values usually called Interval of Validity or IoV.

Not all axes are always used at the same time. For example for some condition sources it does not make sense to have multiple versions (e.g. readings of temperature and pressure probes), or conditions used in simulation studies may not need to take into account time variations.

The term Condition Database refers to a tool or system that allow management and retrieval of condition values.

1.2 LHCb Conditions Database

For most of Run 1 and Run 2 phases, the LHCb Conditions Database was managed and accessed via the Conditions Database library COOL, that via the CORAL library[3] provides an implementation of a Conditions Database on relational database engines such as Oracle® or SQLite.

While COOL allows for multiple usage patterns, LHCb focused on a simple pattern with conditions recorded as XML strings and identified by a filesystem-like path. To simplify access and deployment LHCb divided the conditions in *partitions* by type of data and usage pattern:

- *detector description*: source and versions (no time variations)
- *conditions*: all three axes
- *environment information*: source and time (no versions)

where the prefix of the path identifying a condition source is used to choose the partition.

Due to the limited size of the detector and the control exercised on what was allowed to be recorded in the Conditions Database, LHCb managed to keep the size of the data to a few GB (< 4) for all Run 1 and Run 2 conditions, so we could host our full Conditions Database in SQLite files deployed via CVMFS[4].

1.3 Limits of the current system and requirements for the future

Although COOL served us well for all these years, the project entered a phase where only minimal maintenance was guaranteed (just bug fixes and adaptation to new versions of the

*e-mail: marco.clemencic@cern.ch

underlying libraries). This was not enough for LHCb plans for the new software being developed for Run 3, in particular for what concerns thread-safety.

Searching for a new Conditions Database solution, on the one hand LHCb wanted to find some of the features available in COOL:

- access to conditions with the familiar three coordinates (path, time, version)
- possibility of file based storage

on the other hand, it was felt necessary to improve maintainability both of the code and of the condition data, and performance in terms of access speed and disk size should be comparable to those of COOL.

1.4 Git

Git[5] is a distributed version control system, originally developed to track the development of the Linux kernel, now used by numerous software projects including LHCb ones. Among all the features of Git, those more relevant for this project are

- tracking of multiple versions of a filesystem hierarchy
- each clone of a Git repository contains all versions of all files
- file based compact storage
- built-in incremental replication and synchronization

2 Implementation

2.1 Git as a Conditions Database backend

While investigating possible alternatives to COOL we realized that Git could be a perfect fit for the detector description partition, where we only store multiple versions of a number of XML files. Moreover, if using only the so called Git *bare* repositories (i.e. repositories with only the Git database and no working copy of the files), the space occupied by the repository is very small.

Starting from that idea we developed a prototype of an access layer, based on the libgit2 library[6], that replaced the SQLite detector description partition with an equivalent Git repository. The result was promising and we decided to try to add the missing axis (time) to the prototype.

2.2 Adding the third dimension

To add the time dimension to our prototype we decided to introduce a special convention on a filesystem, where regular files represent condition values with IoV spanning for the whole allowed range of times, while directories matching some criteria are used to represent condition values varying with time. In particular, the directory must contain a file called `IOVs` consisting of a simple list of pairs IoV and file, such that the file represents the condition value for the matching IoV (see figure 2).

The format or name of the special `IOVs` file is not particularly important, but the convention must be well defined. For example, for the format of the file we opted for an extremely simple text format where each line contains first a Unix time stamp (seconds since January 1st 1970 UTC), a space and the path to a file (relative to the directory containing the `IOVs` file). The Unix timestamp identifies the start of the IoV of the matching file, while the end of

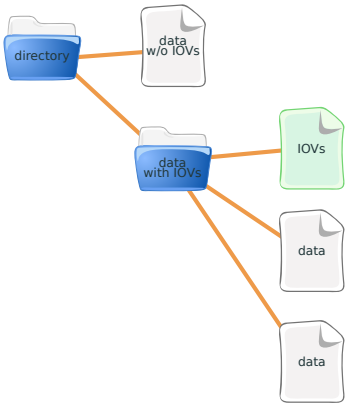

Figure 2. Scheme of the directory structure of a Git CondDB in case of condition value with or without specific IoVs.

the IoV is identified by the start of the next IoV, if present, otherwise we use the upper limit of the full time range.

We could have chosen a more compact and efficient binary format, but the text format was good enough for the initial prototype and has the advantage of being more friendly towards existing tools in the Git ecosystem.

For sake of simplicity of the implementation of the prototype, we required that the entries in the `IOVs` file were in chronological order, so that the look up of a given IoV could be performed with a trivial scan of the file, line by line.

The only optimization we implemented for the IoV lookup was a mechanism to group the IoVs such that, we could reduce the algorithmic complexity from $O(n)$ to $O(\log n)$, by means of a simple recursive algorithm, described in section 2.3.

2.3 The lookup algorithm

The simple convention we set up for representing IoVs on a normal filesystem layout allows for a simple recursive algorithm for the retrieval of condition values from a Git repository, represented in figure 3.

Given a version id (as a valid identifier for a Git commit), a condition id (as filesystem path in the repository) and a point in time, we initialize the *current* IoV to the maximum allowed range, then we look for an entry in the repository matching the commit id and the path. If the entry is a regular file we return the data and the current IoV. If the entry is a directory, we extract the special file `IOVs` and search for the IoV containing the requested time point and the corresponding path, we intersect the current IoV with the one just found and use the new IoV as the current one, then we iterate looking for the new path and we stop the recursion when the last path points to a regular file.

In this simple algorithm we basically use only the API call of the `libgit2` library to retrieve the object at a given path and version in the repository. If the object represents a regular file, we extract the data out of it, otherwise we retrieve the `IOVs` file and continue. Thus the code we have to maintain on top of this simple call is a thin layer only taking care of the IoV management logic.

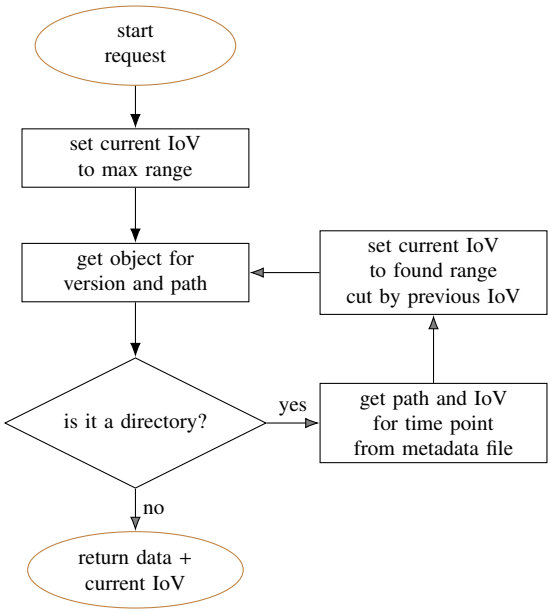

Figure 3. Flow graph of the algorithm for the retrieval of the IoV and condition value for a given path, version and time point.

Backend	Total Time
COOL/SQLite	8.1 s
Git	3.5 s

Table 1. Time to load the whole content of a predefined version of the Conditions Database on an Intel® Core™ i7-7560U 2.40 GHz CPU, with SSD disk, accessing the Conditions Database files from CVMFS (hot cache). In this test we perform ~13000 requests to the Conditions Database API.

3 Performance

With the Git CondDB prototype being complete functionality wise, we could compare it with the COOL based solution.

From the results of speed measurements reported in table 1, it appears that although we did not invest time optimizing the code, Git CondDB is more than twice faster than COOL with SQLite. It must be noted that loading the whole content of the conditions data requires also parsing of several XML files and the creation of detector description objects in memory, which amount to ~20 s. In the normal execution of LHCb reconstruction application we never load the whole Condition Database content, so the speed up due to the switch to Git is negligible in normal cases.

Thanks to the efficient compression of data and metadata in a Git repository, the space occupied by the Git CondDB is more than 5 times smaller than the same data in COOL managed SQLite files (table 2).

	SQLite	Git bare rep.
Detector Description	34	3
Alignment and Calibration	730	285
Environment	2300	357
Simulation	25	9
Total	3089	654

Table 2. Space on disk in MB occupied by the LHCb Conditions Database (as of May 2017) by COOL SQLite files and Git bare repositories. Actually the Git CondDB repositories also contain detector description and simulation conditions for the studies for the upgrade of the detector, which could not be stored in COOL databases because it does not support branches.

4 Commissioning

With the first experiments started in August 2016, after only a few months of development and testing, in December 2016, we have been able to commission the Git CondDB access layer as an alternative to the existing COOL based Conditions Database, including the tools to copy from COOL to Git.

Thanks to the Git built-in incremental synchronization of the cloned repositories, we have been able to replace the complex custom tools we used for the replication of SQLite files with extremely simple scripts based on standard commands.

With the core component and the support tools ready so quickly, we managed to use the new Git CondDB as main LHCb Conditions Database for the data taking started in March 2017, less than one year after its conception.

In April 2018 we managed to backport to old version of the LHCb software stack the code to access the Git CondDB, opening the road to the complete decommissioning of the old system based on COOL.

5 Conclusion

The Git CondDB proved to be a valuable replacement for the COOL/SQLite Conditions Database solution. In addition to the improvements in terms of space occupancy and time, we reduced enormously the maintenance cost replacing several custom tools with standard tools and services, like the Git built-in synchronization and the GitLab[7] installation at CERN. Moreover, the improved workflow has been so welcome in LHCb that instead of maintaining the old SQLite databases for old versions of the software stack, we decided to backport the Git CondDB to those old versions of the software.

It must be pointed out that when LHCb started COOL was the best available option and served us well for all this time. Only the incredible widespread adoption of Git after its first release in 2015, and the amount of tools and services developed around it made it possible to imagine it as a possible Conditions Database backend.

Despite the benefits of the new solution, it is fair to mention that it has some limits. In particular, as any other file based solution, Git CondDB will not be able to scale easily to extremely large Conditions Databases, although Microsoft keeps Windows source code in a very large Git repository[8].

References

[1] A. Valassi, R. Basset, M. Clemencic, G. Pucciani, S.A. Schmidt, M. Wache, *COOL, LCG conditions database for the LHC experiments: Development and deployment status*,

Proceedings, 2008 IEEE Nuclear Science Symposium, Medical Imaging Conference and 16th International Workshop on Room-Temperature Semiconductor X-Ray and Gamma-Ray Detectors (NSS/MIC 2008 / RTSD 2008) : Dresden, Germany, October 19-25, 2008 pp. 3021–3028 (2008)

[2] SQLite project, *SQLite*, [software], `https://sqlite.org`, [accessed 2018-10-27]

[3] R. Chytracek, D. Düllmann, G. Govi, A. Kalkhof, Z. Molnár, A. Valassi, *Distributed database access in the LHC computing grid with CORAL*, pp. 3029–3035 (2008)

[4] J. Blomer, C. Aguado Sanchez, P. Buncic, A. Harutyunyan, *Distributing LHC application software and conditions databases using the CernVM file system*, J.Phys.Conf.Ser. **331**, 042003 (2011)

[5] Git project, *Git Distributed Version Control System*, [software], `https://git-scm.com/`, [accessed 2018-10-27]

[6] libgit2 project, *libgit2*, [software], `https://libgit2.org/`, [accessed 2018-10-27]

[7] GitLab project, *GitLab*, [software], `https://about.gitlab.com/`, [accessed 2018-10-28]

[8] Brian Harry, *The largest Git repo on the planet*, `https://blogs.msdn.microsoft.com/bharry/2017/05/24/the-largest-git-repo-on-the-planet/`, [accessed 2018-10-28]