LHCb Conditions database operation assistance systems

M Clemencic1, I Shapoval1,2, M Cattaneo1, H Degaudenzi1,3 and R Santinelli1

1 European Organization for Nuclear Research (CERN), CH-1211 Geneva 23, Switzerland
2 Kharkiv Institute of Physics and Technology (KIPT), UA-61108 Academyichna 1, Kharkiv, Ukraine
3 École Polytechnique Fédérale de Lausanne (EPFL), CH-1015 BSP 616, Lausanne, Switzerland

E-mail: illya.shapoval@cern.ch

Abstract. The Conditions Database (CondDB) of the LHCb experiment provides versioned, time dependent geometry and conditions data for all LHCb data processing applications (simulation, high level trigger (HLT), reconstruction, analysis) in a heterogeneous computing environment ranging from user laptops to the HLT farm and the Grid. These different use cases impose front-end support for multiple database technologies (Oracle and SQLite are used). Sophisticated distribution tools are required to ensure timely and robust delivery of updates to all environments. The content of the database has to be managed to ensure that updates are internally consistent and externally compatible with multiple versions of the physics application software. In this paper we describe three systems that we have developed to address these issues. The first system is a CondDB state tracking extension to the Oracle 3D Streams replication technology, to trap cases when the CondDB replication was corrupted. Second, an automated distribution system for the SQLite-based CondDB, providing also smart backup and checkout mechanisms for the CondDB managers and LHCb users respectively. And, finally, a system to verify and monitor the internal (CondDB self-consistency) and external (LHCb physics software vs. CondDB) compatibility. The former two systems are used in production in the LHCb experiment and have achieved the desired goal of higher flexibility and robustness for the management and operation of the CondDB. The latter one has been fully designed and is passing currently to the implementation stage.

1. Introduction

The CondDB [1] of the LHCb experiment [2] provides versioned, time dependent geometry and conditions data for all LHCb data processing applications (simulation, high level trigger (HLT), reconstruction, analysis) [3] in a heterogeneous computing environment ranging from user laptops to the HLT farm and the Grid. The use cases of computing environment impose front-end support for multiple database technologies (Oracle [4] and SQLite [5] are used). These technologies are interchangeable in the LHCb computing model [3] in the sense that any LHCb application can use both of them to access CondDB: Oracle-based and SQLite-based CondDB are kept synchronized. Although depending on the particular computing environment (Grid Tier-1s and Tier-2s, LHCb online event filter farm or a user laptop) one or another technology occurs to be more appropriate in terms of data processing requirements and efficiency. This implies a
non-trivial deployment model of CondDB described in detail in [6]. For our consideration it is important to mention the separation of the roles between the Oracle-based and SQLite-based CondDB from the point of view of the CondDB content. The CondDB internal structure is partitioned. There are five specialized partitions that we will refer to throughout the paper:

- **DDDB**: LHCb detector multi-version description partition. It is used to describe the geometry of the LHCb detector and it is common between real data reconstruction and simulation;
- **LHCBCOND/SIMCOND**: LHCb detector multi-version conditions partitions. They are used to describe time dependent conditions like calibrations, alignment, etc. The former partition is used for real data reconstruction and analysis, and the later one for experiment’s detector simulation;
- **DQFLAGS**: LHCb multi-version data quality flags partition. It is used for tracking the quality of physics data taken by the experiment.
- **ONLINE**: LHCb experimental area single-version conditions partition.

The evolution of all partitions, apart from the last one, starts at SQLite-based CondDB and is replicated then to the Oracle-based CondDB. The Online partition instead is written first to the Oracle-based CondDB, with which the SQLite-based CondDB is then synchronized (an Online SQLite snapshot is created).

Firstly, sophisticated deployment tools are required to ensure timely and robust delivery of CondDB to all computing environments. We devote sections 2 and 3 to present such tools. Secondly, the CondDB content has to be managed to ensure that the payload is internally consistent and externally compatible with LHCb computing entities, the LHCb physics applications and LHCb physics data. A design of a system to track and verify the internal and external CondDB compatibility relations is presented in section 4.

2. LHCb Conditions database state tracking system

2.1. Motivation

The LHCb experiment is using the Oracle 3D Streams [7] technology to replicate the Oracle-based CondDB to Tier-1 Grid sites. Unfortunately, this technology didn’t show itself as fully reliable. In particular, we have faced several situations when the new content of the Oracle-based CondDB was not successfully propagated from Tier-0 to Tier-1s replicas while the Oracle Streams had been giving the successful propagation status. This caused the LHCb production jobs to fail at Tier-1s in cases when the CondDB integrity corruption was critical. But the worst consequence of such a silent propagation failures happens when the integrity corruption is not critical enough to raise a job execution failure. In this case the job executes successfully but using the wrong CondDB state. Such a case has been never spotted since it’s difficult, if possible at all, to discover such a problem by just analyzing the job’s physics results.

Thus, the decision was taken to develop a CondDB state tracking system (STS) discovering both mentioned above CondDB integrity corruption cases. It was initially developed to ensure the correctness of the Oracle-based CondDB replication performed by Oracle 3D Streams but can be adapted to any other CondDB replication machinery (e.g. the one presented in section 3).

2.2. STS design and implementation

The overview of the CondDB state tracking system is shown in figure 1. In the following subsections we will describe the mainstream of the system.
2.2.1. **Input entities**  
The input data (see the *Context diagram*) is essentially a CondDB tag name together with its corresponding partition name. This information is injected by the CondDB Manager as soon as the new CondDB tag is created. The reference hash sum is then computed by the *publish* process for inserted tag and saved in a tags status database described in the next subsection. Right upon the insertion, the tags’ status is *BAD* for all Tier-1 sites.

2.2.2. **Tags status database**  
The core of the system is the tags status database (TSDB). This is the SQLite database which has a simple two-table schema (see figure 2). The database is used to store the status of each tag at specific moment in time.

2.2.3. **Tracking tags integrity**  
The *Check&Update* process, described at Level 1 in figure 1, is run regularly (every two hours) and is managed by SAM jobs [9]. The *dispatcher* process extracts the set of CondDB tags which meet the following conditions strictly in the order:
- if a tag is inserted not earlier than one month ago, and
- if the last validation time of a tag is older than two weeks, or
- if a tag has a *BAD* status.

Then this set of tags is fed to the *agent* processes wrapped into the SAM jobs which are submitted to each Tier-1 site. Each *agent* returns the results of its validation to the *dispatcher* process which manages the update of TSDB with the new status information through the *update* process.
2.2.4. Output entities  The STS output (see the Context diagram in figure 1) is managed by the extract process. It can respond to any client querying the TSDB through the HTTP request for any particular CondDB tag status at any site. One such client is the LHCb DashBoard web portal (see figure 3). It displays the current status of the CondDB tags at all Tier-1 sites. One can see that most of the tags were propagated successfully while the others have the BAD status which in this case is due to the new “head-20110823” tag has not yet been propagated to any of the Tier-1 sites. The other Checking status in the case of IN2P3 site means that the two-weeks age limit, crossing which declares a tag’s status as expired, has been passed recently and STS is now re-checking the integrity of all the relevant tags at this site.

Figure 3. The LHCb DashBoard web portal displaying the status of the CondDB tags at Tier-1 sites.

2.3. Summary
The system developed has been put in LHCb production and monitors currently the Oracle-based CondDB integrity at Tier-1s. It allows to improve LHCb productions in the following aspects:

- once any tag integrity violation is detected by the STS at some Tier-1 site one can take a preventive measure of excluding that problematic site from the list of the active ones for a particular LHCb production, which helps to:
  - make the production quicker (the time is not spent on the job execution that will anyway fail and no job waiting in a queue until it is re-submitted for a new execution);
  - save site’s computing resources for other tasks (since the jobs that will anyway fail at some moment will not be started there at all)
- the system provides an extra layer of integrity verification to spot any CondDB propagation problems left undetected by the Oracle Streams.

In the next section we present an alternative CondDB deployment system that will make the CondDB propagation more fast, robust and efficient. The STS system described in current section will be integrated with the new deployment system once the switch to it will be fully completed.

3. New deployment system for SQLite databases
3.1. Motivation
The standard LHCb package-based deployment model for SQLite-based LHCb CondDB suffers of two main problems: big space requirements and rare updates.

In this model CondDB, consisting of a few distinct SQLite files, is deployed as a subdirectory of the package SQLDDDB. This means that it follows the rules of a regular software package,
even if it is not the most appropriate way for that kind of content. The main difference is that a conditions database contains, by definition, all the previous versions of the informations (tags), while the files in regular packages are changing between versions of the package so that the only way to access the old version of the information is to use the old version of the package.\(^1\) Because of this difference, while it is very appropriate to keep several copies of a regular package in the release area, a single copy of the CondDB would be enough, provided that it is up-to-date.

Since the CondDB is deployed as any other package, we keep several copies of it on the AFS release area at CERN and on shared spaces at the computing centers. While this was not worrying at the beginning because the size of the package was rather small (less than 100MB), the situation radically changed after a couple of very big updates of the conditions (about 600MB each) bringing the size of the package above 2GB. Because of the nature of the CondDB, the space wasted (occupied but not needed) is a lot.

The other big limitation that the deployment model used for regular packages is imposing on the CondDB is that each update has to be prepared, released, packaged and installed. The release procedure is not very long by itself, but each release implies a new copy of the package to be installed, so we wait until it’s strictly needed. The frequency of the updates depends on the ongoing activities and, unless some exceptional occasions, it is not higher than once a week.

It must be noted that the SQLite version of the CondDB was meant to be used in special cases, like simulation productions at Tier-2s and disconnected analysis, while all the other cases should have used the direct Oracle access. This would have allowed us to reduce even further the frequency of SQLite releases. Unfortunately, the Oracle access has proven itself less satisfactory (in terms of efficiency, stability and ease of use) than the SQLite one, so we decided to use Oracle only when strictly needed (first pass reconstruction) and SQLite in every other case.

### 3.2. The new deployment model

Because of the problems with the size and the frequency of updates, it has been decided to develop a deployment model specific to the CondDB with the aim of avoiding those pitfalls.

The new model has been thought and designed in a generic way that can be applicable to other files or directories, like the decay files, used to control the simulation of the B decays by the LHCb event generator, or the LHCb high level trigger configuration keys.

The basic idea is to keep a single local copy of the SQLite files that constitute the CondDB and efficiently keep it up-to-date.

#### 3.2.1. Procedure

The procedure is entirely driven by a distribution and backup system (DBS) and thus a high-level information flow can be expressed with just three entities (see figure 4, Context diagram):

- a master copy
- DBS
- a local copy

The distribution system is decomposed in turn into a public repository and three mainstream processes (see figure 4, Level 0):

- preparing
- publishing
- updating

\(^1\) The choice of this deployment model has been made to be able to have CondDB in production quickly, reducing the amount of specific work by reusing the existing infrastructure.
The mainstream processes are as well compounds and consist of a set of helper subprocesses (shown at Level 1 of figure 4). Each of the helper processes will be described in detail in subsection 3.2.2.

The required changes to the CondDB files are applied to the master copy then published, or uploaded, to the public repository. The update process consists of comparing the content of the public repository with the content of the local copy and subsequent synchronization applying the differences with respect to the latest snapshot at the repository to the local copy.

This procedure is very similar to the one at the core of the CERN Virtual Machine FileSystem (CVMFS) [10]. Actually, one could just reuse the same infrastructure and machinery, but the implementation of CVMFS does not contain reusable parts, so the only way to do it would be to rewrite the client (update) part of the protocol from scratch, based on the protocol specification. The amount of work required to reuse the CVMFS infrastructure was thought to be too much to achieve results in a reasonable time scale, if compared with the work required to have a home-made, simplified implementation.

Figure 4. Leveled data flow diagram for the new deployment model of the SQLite files (shown in the Yourdon-DeMarco DFD notation [8]). Levels 1A and 1B are described in detail in subsection 3.2.2.

3.2.2. Implementation

Master copy  The master copy, shown in figure 4, is a set of files on top of which the preparation process is run to take into account all up-to-date modifications. The version to be used as the master copy is chosen by the CondDB Manager and can be extracted from the backup of the public repository which will be described further on.

Preparing to publish  The preparation process described here is specific to the CondDB deployment case only but can be adapted to any of the SQLite database packages. It consists of three processes working on the two copies of the file sets (see figure 4, Level 1A) to prepare and pass the final copy to the publishing process.
• **Gateway copy.** The *gateway copy* is the copy which is analyzed regularly by the *publishing* process. If any change of state is found in the SQLite files set, this change is published to the *public repository* (described in the next section). The layout of this copy is shown in figure 5.

• **Generate Online.** The automatic *Generate Online* process is run regularly generating the Online partition snapshot file(s) in the *gateway copy*. This process detects the current state of the Online SQLite snapshots set in the *gateway copy* and synchronizes this set with the Oracle-based CondDB. This includes not only latest (by year, or month) snapshot but also all those previous lost or intentionally removed ones.

• **Development copy.** The *development copy* is the copy which is used by the CondDB Manager to apply new changes to. The copy is also used by the LHCb Nightlies Builds system [11] to test the correctness of the development SQLite files by the LHCb software. The structure is similar to the one of the *gateway copy* (figure 5).

• **Patches.** Patches, requested by the subdetector groups, are applied to the *development copy* by the CondDB Manager and, together with the latest Online snapshots delivered automatically from the *gateway copy* (see synchronizing process described below), are tested by the LHCb Nightlies Builds system.

• **Synchronizing.** Synchronizing the *gateway copy* with the *development copy* consists of two data flows:
  - automatic delivery of the latest Online snapshot files to the *development copy*, with the same frequency as the Online snapshots are updated in the *gateway copy*;
  - delivery of the patched SQLite files to the *gateway copy* launched by the CondDB Manager when the files are tested and ready to be deployed.

| path  | hash | size   |
|-------|------|--------|
| Dir   |      |        |
| FileA | X    | 100 MB |
| FileB | Y    | 50 MB  |
| FileC | Z    | 1 MB   |

Figure 5. Example of a *gateway copy*. The data to be published consists of one directory (Dir) with two files (FileA and FileB) and a third file (FileC) at the same level of the directory.

**Public repository** The public repository is an HTTP server hosting files organized in a special way.

The file *repository* contains the version number of the repository as plain text. This number gives us the possibility to implement easily backward compatible clients in case we need to change the layout of the repository.

The *catalogs* directory contains several XML files, each describing the status of the files in the gateway copy at a specific moment in time. The number of the catalog files, and thus the number of the gateway copy states kept in the repository, is tunable and is controlled by the publishing process. The content of the catalog files is described in the section about the publishing process.

The file *current* is a simple text file containing the name of the most recent file in the *catalogs* directory. More details about the role of this file are given in the section about the updating process.

The last entry in the repository is the directory *pool* which contains the actual data in a compressed format. Each file in that directory correspond to one file present in the development
copy at the moment of a publishing. To simplify the bookkeeping, the files are indexed by content, using a cryptographic hash as name.

| path      | size  |
|-----------|-------|
| repository | 3 B   |
| current   | 19 B  |
| catalogs  |       |
| 2011-11-19-20:25:44 |   |
| pool      |       |
| X         | 5 MB  |
| Y         | 3 MB  |
| Z         | 0.05 MB |

**Figure 6.** Content of the repository after the publishing of the data described in figure 5. The three files are compressed and stored in the directory pool, while the catalog is saved in the directory catalogs using the current date as name. The date used is then stored in the file current. The file repository contains the version number of the repository layout.

```xml
<SQLDDDB date="2011-11-19-20:25:44" tsize="151000000B">
  <file path="Dir/FileA" sha1="X" size="100000000B" />
  <file path="Dir/FileB" sha1="Y" size="50000000B" />
  <file path="FileC" sha1="Z" size="1000000B" />
</SQLDDDB>
```

**Figure 7.** Content of the catalog file 2011-11-19-20:25:44 showed in figure 6. For each file there is an entry with the path of the file, its cryptographic hash (sha1), which is also the name of the corresponding file in the pool, and the uncompressed size.

Publishing The publishing process is run automatically with the same frequency as the most frequent change of state in the gateway copy (the latest Online snapshots) has.

For each file in the gateway copy, the publishing script collects its cryptographic hash, the size and the full path relative to the top level parent directory.

Before the actual data transmission to the repository the script checks whether an identical file in the repository pool exists for each of the gateway copy files. If it does not exist, it means that we are dealing either with a new file or with a new content of an existing file, so the original file is compressed and stored in the pool with the hash as name. In the opposite case, i.e. the hash is found in the pool, the content we have is already present in the repository and there is no need for its transmission.

Once all the missing contents are added to the repository pool, a new catalog is generated. The format of the catalog is a simple XML file that contains one tag per recorded file and, for each of them, attributes for the path, the cryptographic hash and the size. The new catalog is added to the directory catalogs using the current date and time as file name.

The last step is to write into the file current the name (date and time) of the catalog file just written.

---

2 We use the sha1 algorithm since it is the one commonly used for this purpose, e.g. by the git revision control system [12] or by CVMFS [10].

3 The algorithm chosen for the compression is bzip2 because it has shown itself as the one with the best compression ratio for our use case (CondDB SQLite files contain mostly text).
Local copy  The local copy is equivalent to the image of the gateway copy, but not identical. Its structure has been studied to allow efficient updates and to avoid disruption to the processing that may be using its content while the update is performed.

The hidden directory .pool is essentially a subset of the pool directory in the repository, with the difference that in this case the files are not compressed and there is one extra file which is a copy of the current catalog in the repository at the moment of the latest update.

The file catalog is a link to the catalog in the .pool just mentioned above. More details about its role are given in the next section.

For each non-empty directory in the gateway copy, a directory with the same name is found in the local copy. For each file, instead, the local copy contains a symbolic link that points to one of the files in the local .pool directory (see figure 8).

| path               | size  |
|-------------------|-------|
| catalog → .pool/2011-11-19 20:25:44 |       |
| Dir                |       |
|   FileA → ../.pool/X |       |
|   FileB → ../.pool/Y |       |
| FileC → .pool/Z    |       |
| .pool              |       |
| X                  | 100 MB|
| Y                  | 50 MB |
| Z                  | 1 MB  |

**Figure 8.** Content of the local copy after the update from the repository of figure 6. The file catalog is a link to a copy of the most recent catalog found on the repository and it used to quickly compare the content of the local copy with the repository. The file system hierarchy found in figure 5 is reproduced with symbolic links pointing to the files in the .pool directory that holds the actual data.

Updating  The update process (see figure 4, Level 1B), performed by a Python script, synchronizes the content of the local copy with the latest data published on the repository.

The first file that is downloaded is the repository file from the repository URL. From the version number in the file, the script decides if it can continue or it has to abort because it does not understand the format of the repository.

The second file downloaded is current, the content of which is used to select the most recent catalog file from the catalogs directory. The usage of the intermediate file, before downloading the catalog, reduces the possibilities of interference between the publishing and the updating processes.

Once the most recent catalog is downloaded, its content is compared with the catalog in the local copy. The three regular cases handled by the updating process are shown in table 1.

When a file has to be updated, the new content, identified by the cryptographic hash, is downloaded from the repository pool, decompressed and saved in the local pool directory (.pool) using the same name. At this point, a symbolic link pointing to the file in the local pool is created to reproduce the hierarchy in the gateway copy.

Before the actual download the update process detects also if there are shared installation areas available and if yes it reuses the shared files (local links to the shared area SQLite files are created) and continues the update process described above on top of them. The copies in the shared installation areas on Grid sites will be updated on a regular basis (by SAM [9] jobs) thus reducing a lot the load on the distribution server which comes from the Grid jobs.
Table 1. A set of most common local copy states and actions performed on them by the update process.

| State detected                                                                 | Action                                                      |
|-------------------------------------------------------------------------------|-------------------------------------------------------------|
| a file entry exists only in the local catalog                                | the local file is removed                                   |
| a file entry has the same cryptographic hash in both catalogs                 | the file has already the most recent content, no action     |
| a file entry exists only in the downloaded catalog or there is a difference in the cryptographic hash | the file is updated with the most recent version of the content from the repository |

Exceptional cases Several special cases are considered in the implementation of the scripts to make the system robust. Those can be decomposed into two families: non-standard, but foreseen, cases encountered by the update process in the structure of the local copy file hierarchy (see table 2) and the cases of “external” problems encountered by the update process during its execution (see table 3).

Table 2. A set of exceptional local copy state changes and related actions on them performed by the update process.

| State detected                                                                 | Action                                                      |
|-------------------------------------------------------------------------------|-------------------------------------------------------------|
| a local file is absent while being declared in the local catalog              | the latest file is downloaded to fix the local file hierarchy |
| the local catalog is not found, or requested to be ignored (while the file hierarchy is fully/partially in place) | the local catalog is reproduced re-hashing the file hierarchy and normal update process is started |
| a local file is modified without corresponding update of the local catalog file record | the file is left not updated (unless the update process is forced to update it) |

Table 3. A set of update process execution problems and related actions performed to recover from them. “Clean exit” recovery action means untouched local file hierarchy and removal of all the local temporary update process files on exit.

| Execution problem                                                                 | Action                                                      |
|----------------------------------------------------------------------------------|-------------------------------------------------------------|
| requested file can not be downloaded from the repository for any reason          | perform 10 extra attempts in a randomized short period of time and exit cleanly if all of them were unsuccessful |
| decompression engine is not found in the system PATH, or decompression failed for any reason | perform clean exit                                         |
| I/O FS problems (including lack of destination’s free space)                     | perform clean exit                                         |

3.3. Deployment

From the procedural perspective, there are two main differences between the old, package-based, deployment of the CondDB SQLite files and the new, CVMFS-like, one.

In the old system, any change to the CondDB required a new release of the SQLDDDB package to be deployed. Installing the package was enough to get a local copy of the SQLite
files and the new version number was used to push an update to the Grid shared software areas (plus CVMFS and Online). Individual users had to explicitly install the latest version to get an update.

With the new system, a change in the CondDB does not imply changes in the SQLDDDB package so there is no need for a new release. This also means that the changes cannot be pushed but have to be pulled. A pull approach is not directly applicable to the Grid shared software areas, because we do not have the control on the remote systems. Moreover, the SQLDDDB package will not contain the SQLite files, so it is not enough to install it to get the local copy of the files.

To make the installation of the new SQLDDDB functionally equivalent to that of the old one we introduced the concept of an “update hook” in install_project.py. The update hook is different to the already existent post-install hook in the fact that it is called every time the installation of a package is required, not only when it is actually performed.

When installed from scratch, the new SQLDDDB will trigger a call to the update script, which will download and save the SQLite files in a predefined directory. When install_project.py is called again to install the same version of SQLDDDB (either explicitly or as a dependency of another project), the installation will not take place, but the update script will be called, so the SQLite files will be updated.

Below the main use cases are described in some more details.

3.3.1. User local installation
The technique of the update hook described above makes the switch from the old approach to the new one almost transparent to the casual user.

The first installation of SQLDDDB, as already mentioned, produces a fully functional local copy of the CondDB, as it was the case with the previous versions.

The updating, instead, is not transparent, but is very simple and more efficient. The user can update the local copy by calling explicitly the update script or by re-issuing a call to install_project.py for the package already installed. It should be noted too, that the installation of any LHCb software will imply a request for the installation of SQLDDDB, so an update of the local copy.

3.3.2. CVMFS
The software installation to the CVMFS repository is technically identical to a user local installation, so the comments above about the installation and the update are still valid.

What is special in the installation to CVMFS is that we have some control on the machine on which the software is installed and we are able to set up an automatic procedure to perform the update. This procedure is now a cron job running on the CVMFS installation machine.

3.3.3. AFS at CERN
Even though the long term plan is to replace the special AFS release area with something else (e.g., CVMFS), we still support it.

The update script does not make any difference between the AFS release area and a local installation, because it uses special environment variable to deduce where the SQLite files have to be installed or updated.

What makes a difference between a local installation and the AFS release area is that the software is built and not installed in the release area, so we need to trigger the updates in an automatic or semi-automatic way, more or less like on CVMFS.

3.3.4. Grid (non CVMFS)
We do not have the control of the machines hosting the software shared areas on the Grid, so we cannot set up regular automatic updates. What we can do is to leverage on the calls to install_project.py.
install_project.py is called somehow regularly in SAM jobs on the Grid to install new software releases. Every time the installation job is run, it calls install_project.py for all the software releases required, when it is called for SQLDDDB the CondDB SQLite files will be updated.

install_project.py is called by LHCb Grid job wrapper as well to ensure that the required software is installed before using it. Since SQLDDDB is included, directly or indirectly, in the software to be installed for each job, the most recent version of the SQLite files will always be used4.

3.4. Frequency of updates and repository load

A possible drawback of the new pull-style approach is that the load on the infrastructure may become too high under some circumstances. For example the amount of data downloaded from the web server can be too much because all the grid jobs are trying to download the latest versions of several partitions of the CondDB.

It is difficult to estimate the amount of load put on the infrastructure because it depends on several parameters, most of them tunable. But what is important to check is that the new deployment model will not push the load to critical values which may hit, e.g., the LHCb software distribution server. For that purpose we set up the worst-case environment of clients in which the update process is launched by the LHCb Grid job wrapper of every LHCb Grid job5. This extreme scenario will help to reveal the horizons of the load which may be put by the new deployment system on the infrastructure.

An update process always downloads from the repository the two small files repository and current plus the most recent catalog. In all of the load measurements the effect of these downloads was found to be negligible. So in all our tests described below we took into account only the actual SQLite data download.

We have simulated two extreme load cases:

- The most common low-load (LL) case in which only the latest Online SQLite file has to be downloaded. The amount of data to be downloaded may reach the order of 11 MB for a full year snapshot. When the density of clients per unit of time is very high, even this LL case may become problematic.
- The high-load (HL) case in which all, or several, large SQLite files (e.g. DDDB, LHCBCOND and SIMCOND CondDB partitions) have to be downloaded. As up to 2012 year they sum up to about 120 MB.

Below we are presenting a set of measurements for both HL and LL cases showing the load from various perspectives. Figure 9 shows the percentage of the SQLite to total6 data amount downloaded from the distribution server per day during the Production tests. One can see that the LL case has a negligible 0.5% out of the total data downloaded from the server, whereas the HL case demonstrates a slightly more significant 6.5%.

4 The implementation of the update script takes correctly and efficiently into account the case of a local installation directory on top of a shared one (multiple MYSITEROOT).
5 Note that only LHCb software installation SAM jobs and LHCbDirac job wrappers of prompt reconstruction LHCb Productions will launch the update process in real environment.
6 The LHCb distribution server is a general purpose server used to distribute all LHCb software to the consumers.
Figure 9. The percentage of the SQLite to total data amount downloaded from the distribution server per day.

More important issue is the load during the peak periods of Productions. Figures 10 and 11 show the evolution of the SQLite versus non-SQLite data amount downloaded over the Production period. Again we see that the LL case doesn’t produce significant extra load over the background values. HL case is much more significant though, but still much below the critical load value which we have chosen for some reference to be the server’s Ethernet card speed (1Gb/s).

Figure 10. The low-load case. SQLite (latest Online snapshot only, around 0.5MB early 2012) versus non-SQLite data amount downloaded from the distribution server during the Production test (LL) shown in figure 9.
In spite of the low load put on the server by the new deployment system being run out of the box several extra measures may be taken to ensure that the load will not become critical during even greater jobs spikes comparing to those we have managed to achieve in our tests:

(i) adding HTTP proxy servers around Tier-1s and Tier-2s centers to reduce the load on the distribution server (as it is already done for CVMFS)

(ii) to reduce the regular load on the server due to Online snapshot downloads we can either reduce the frequency of the updates of the Online partition or make smaller snapshots (currently we use yearly snapshots).

(iii) publishing large SQLite files to the repository at specific moments in time, i.e.:

(a) such that the interval between the publishing time and the Production start is not less than CVMFS typical propagation time (which is currently of the order of 2 hours) and regular Grid shared areas installation time (which is tunable).

(b) if we have to deploy CondDB during the Production then publish new files out of ongoing Production peak periods

The (a) item ensures that the update process, triggered by LHCb Grid job wrapper, will pick the new published files from the local shared area (either CVMFS, or regular one) of the job skipping thus the actual download from the distribution server. The effect is visible in figure 12. The publishing of several SQLite files was triggered by the CondDB Manager (keep in mind that automatic Online snapshot publishing is run in addition every hour), and it was done right before the Production test in contrast to what suggested above in item (a), thus forcing the update process to download all published files from the distribution server (green area) because the shared areas were not yet updated. Once CVMFS propagated the new files to clients, the update process does not need to download everything (black and red areas), so the load on the distribution server is reduced. One can see that the black area is caused by switching off intentionally one of the Online publications and is segueing into the red area as soon as the next Online snapshot is published. The red area then keeps persistent through the rest of the Production. This is because the new Online snapshot keeps publishing and CVMFS is almost never in time to propagate it to clients before the next new Online snapshot is published. Thus Online snapshot will always be downloaded from the distribution server directly.
Figure 12. Number of jobs sorted by the number of SQLite files downloaded from the distribution server versus time (time granularity is 10 minutes). The measurement done during the Production test (HL) shown in figure 9.

3.5. Summary

The system described in this section has been recently put in production in the LHCb experiment. It solves the deployment and management problems of the previous distribution model. It provides the following benefits:

(i) Faster turn-around for the SQLite-based CondDB releases (no dependency on SQLDDDB package release cycle)

(a) immediate visibility of the new CondDB release once published to the public repository

(b) always up-to-date CondDB Online partition

(ii) More efficient storage of the SQLite-based CondDB releases (no duplication of the database payload)

(iii) More flexible and robust data processing on the Grid. The fact that new generation SQLite-based CondDB is now automatically kept up-to-date allows to switch the data processing on the Grid from the Oracle-based to the SQLite-based one. With that we gain:

(a) no Oracle Streams latency when propagating new changes to Tier centers (both new CondDB tags and Online partition). In fact, "Tier-0 → Tier-1s" LHCb Oracle CondDB replication streams can now be turned off.

(b) the new system makes it possible to run prompt reconstruction at Tier-2s\(^7\) (was not foreseen in the original LHCb Computing model).

Any LHCb SQLite database package may profit from the migration to the new deployment model described in this section.

4. LHCb Software and Conditions Database Cross-Compatibility Tracking: a Graph Driven Approach

4.1. Motivation

In the current LHCb computing model [3] the compatibilities of LHCb applications and LHCb CondDB states are not tracked and are managed in a manual and very limited way. This is too error prone. More over, even if no explicit failure is discovered by the test application executions the danger still exists that the wrong CondDB state may be used for some of the versions of the LHCb applications in which case the results of the LHCb application execution might not be

---

\(^7\) This is not yet used in the LHCb production.
entirely correct. This section proposes a system that will help to track and perform automatic verification of mentioned compatibilities reliably.

We start out with an introduction of objects between which the compatibility will be considered (section 4.2), then, following a definition of the criteria of compatibility (section 4.3), we will propose a design of the compatibility tracking system (section 4.4).

4.2. Compatibility entities
All compatibility entities of our consideration can be subdivided into three classes: the LHCb application states, the LHCb CondDB states and the physics data states. Let’s now briefly describe each class.

4.2.1. Conditions database entities
A general CondDB state is fully defined by the sub-states of its three partitions. So the role of the CondDB compatibility entity is given to this partition state. Each partition state is versioned: the complete partition image is tagged at specific moments of partition’s evolution. So the CondDB state is fully defined by a triplet of tags each of which corresponds to one of the three partitions described in section 1. Not every triplet of tags forms a self-consistent CondDB state (see section 4.3 below).

4.2.2. Application entities
Each LHCb application is a client of CondDB. Some examples of LHCb applications are as follows:

- Moore - used for high level triggering;
- Brunel - used for event reconstruction;
- DaVinci - used for event selection and data analysis;
- Gauss - used for event generation and detector simulation;

For our consideration the concrete application flavor is not important since there is no conceptual difference between them from the point of view of compatibility with a CondDB state. But what is important is that the LHCb computing model forces each application to be versioned and the version number will uniquely identify the application entity in our consideration.

4.2.3. Physics data entities
The raw physics data taken by the LHCb experiment passes several phases of processing by the applications mentioned in the previous subsection. Each data transformation is accomplished using a particular CondDB state (tag) which is stored together with a new data state. So we say that each LHCb data entity is associated with a CondDB tag.

4.3. Compatibility relations
Before going any further we have to specify what stands behind the compatibility concept in our problem. In the LHCb computing model we encounter the following categories of the compatibility relations for the entities described in the previous section:

- CondDB self-compatibility
  - Inside-partition tags compatibility
  - Cross-partition tags compatibility
- External CondDB compatibility
  - Compatibility of a tag with the LHCb application
  - Compatibility of a tag with physics data

So let’s consider all the compatibility categories in more detail.
4.3.1. Conditions database self-compatibility  One should note that the case of inside-partition compatibility (i.e. the compatibility between the tags of the same CondDB partition) has sense in our problem only for more granular CondDB partition states (so called local tags). We will not touch this deeper level of compatibility in this paper and will concentrate on a general CondDB state (a global tag) described in section 4.2 which is most commonly used for the LHCb applications execution. This general CondDB state in turn is a player on the cross-partition tags compatibility field only. Let’s now consider how does this compatibility evolve.

The compatibility relations between the CondDB tags are fully defined by the content of the CondDB partitions versioned with those tags. Suppose, we have a fully compatible set of tags. Suppose also, that we are going to create a new tag in the partition which will include some new set of changes. Depending on the type of the coming change we may obtain the new tag compatible, or incompatible, with the previous tags. Figure 13 and table 4 are summarizing two common cases showing the compatibility consequences of adding new changes to CondDB.

![Figure 13](image-url)

**Figure 13.** CondDB partitions together with their compatibility relations. The COND notation generalizes LHCBCOND and SIMCOND partition cases.

| Potential to break certain compatibility relation | Type of CondDB change |
|-------------------------------------------------|-----------------------|
|                                                  | non-structural changes (e.g., alignment, calibration or particle table updates) |
| A                                                 | No                    |
| B                                                 | No                    |
| C                                                 | No                    |

|                                                  | structural changes (e.g., new detector or condition elements, new database objects' dependencies) |
| A                                                 | Yes                   |
| B                                                 | No                    |
| C                                                 | No                    |

Declaring the compatibilities of a new tag with the previous ones is a responsibility of the LHCb sub-detector groups.

4.3.2. External Conditions database compatibility  We say that a CondDB tag, say tag T, is compatible with the LHCb application and a data entity (see figure 14) if the following three conditions are met:

- The LHCb application does not fail while executing over the CondDB of a version identified by the tag T;
• The CondDB structure being expected by the LHCb application maps identically to the actual CondDB structure declared by the tag $T$;
• The tag, associated with a data, is equal to the tag $T$.

Most of the failures of the first bullet type are caused by the second bullet condition violation.

**Figure 14.** More complete view (with respect to figure 13) of the compatibility relations tree. The LHCb application and physics data entities are added.

### 4.4. Compatibility tracking system

The compatibility tracking system is aimed to perform the following tasks:

• automation of both internal and external (see section 4.3) CondDB compatibility verification:
  - at a job configuration stage;
  - at events processing stage;
• smart full setup of CondDB tags for all LHCb applications (with no user configuration needed):
  - at specific moment in time (e.g., the latest compatible tags);
  - by specific data type;
  - by a given *subset* of tags (if any user’s hint is given);
• easy exploration of the internal and external compatibility relations’ trees (e.g., through the web interface).

The first bullet’s sub-item will solve most of the compatibility problems since currently most of them comes from the wrong job configuration. The second sub-item should provide a deeper compatibility verification level for special cases when a job meets events processed using tags different from those declared by a job configuration. The second bullet will give an option of smart and silent compatible tags setup for a given application by a set of key requirements. And the latest bullet will serve mostly the purposes of more convenient and reliable CondDB management for the CondDB sub-detector support groups and for CondDB managers.

In the following subsection we will see how all mentioned tasks can be processed within a generic approach.

#### 4.4.1. Handling all compatibility categories in a generic way

Now that the nature of all compatibility entities and relations between them are described (Sections 4.2 and 4.3) we can put the problem on a higher abstraction level. The complete compatibility relations tree can be expressed in the framework of the graph theory in terms of a directed acyclic weighted attributed graph (DAWAG). We can define the graph elements in the following way:

• A node: a compatibility entity (either an application, or a tag)
  - An application node with an application version as a node name
  - A tag node with a tag version as a node name
• An edge: compatibility relation (either between application and a tag node, or between two tag nodes)

Note that for our purposes the graph must be weighted - we assign each edge a numerical value that will represent a sequence number of the compatibility relation inserted for that node. This will enable the compatibility history tracking and the possibility, for example, of fast search for the latest compatible tags. The graph must be also attributed to store extra information about the compatibility entities to make the graph nodes unique, i.e.:

• for an application node: LHCb application name;
• for a tag node: CondDB partition, tag release date, physics data type this tag is intended for.

An example of such a graph is shown in figure 15.

Figure 15. An example of a DAWA graph for Brunel application. The A-nodes represent Brunel versions while the T-nodes - the CondDB tags entities. The green text of each node represents the attributes, e.g. the tag node’s attributes contain a CondDB partition name field (DDDB, COND, DQFLAGS) and a data type (2011, 2012) field.

Such a representation will enable the extraction of the compatibility information in a natural way. In such approach the procedure narrows down to a search of the graph path which meets conditions of a query. The graph path will always start with an application node which always plays a broker role (see figure 15) and will have a four-nodes length (an application node plus one tag per each CondDB partition). One can imagine some of the possible compatibility queries which can be extracted from the graph tree:

• derive the full set of the latest compatible tags for an application of a specific version and for a specific data type;
• check that a set of entities is compatible (tags, tags and an application);
• extract all possible entities that can augment the requested ones to a full compatible path under specific requirement.

4.4.2. Overview of the compatibility tracking system. To process the tasks listed at the beginning of this section the compatibility tracking system (CTS) will have to “talk” to a graph described in the previous subsection (see figure 15). In particular, it will perform too kind of actions:

• storing the new compatibility relations in the graphs;
• analyzing the stored graphs to get a compatibility solution.

In the latest bullet with the compatibility solution we mean depending on a user query type either the answer whether some entities are compatible, or a set of compatible entities which meet requested conditions.

A possible scheme of such a system is shown in figure 16. One can see that the system expects to receive an input which can be:

• the new compatibility information (supplied by the CondDB Manager);
• the user compatibility query (a user can a regular job, or a web compatibility portal robot).

The output is obviously expected to reply the user query only with a compatibility solution.

![Leveled data flow diagram for the compatibility tracking system (CTS) (shown in the Yourdon-DeMarco DFD notation [8]). The Level 0 layer depicts the CTS intrinsics. See Subsection 4.4.3 for the Compatibility database description.](image)

**Figure 16.** Leveled data flow diagram for the compatibility tracking system (CTS) (shown in the Yourdon-DeMarco DFD notation [8]). The **Level 0** layer depicts the CTS intrinsics. See Subsection 4.4.3 for the Compatibility database description.

### 4.4.3. Persistency choice: a relational database versus a graph database

A core of the compatibility verification machinery will be a database which will be populated with the compatibility graphs. The choice of a technology which can drive such a database depends on several factors. The key ones are:

• how does the technology of a choice fit the general concept of a task:
  - information entities to store;
  - querying efficiency;
• what are the requirements of the chosen technology (available engine’s API, database hosting options, etc.);
• technology support;

The most appropriate database technologies for the considered problem in our opinion are:

• A **relational** one (e.g., SQLite, see [5]);
• A **graph-oriented** one (e.g., Neo4j graph database, see [13]).

The comparison of the relevant to our compatibility problem features of both technologies is shown in table 5. The SQLite does not need any promotion - it is very popular as an embedded SQL database engine with an excellent product support. Storing a graph in a relational database is simple and the SQLite engine could be an out-of-the-box solution taking into account the...
availability of the database engine libraries in the LHCb computing environment (Neo4j instead needs deployment). It has although an obvious drawback: SQLite’s relational nature doesn’t fit the graph-oriented spirit of our compatibility tracking system design with graphs as a storage unit. This has the following consequences:

- increased amount and complexity of the database client-side code comparing to the graph-oriented database one;
- increased execution time of the client-side code due to needed graph object assembling and disassembling;
- low database querying efficiency compared to graph-oriented database approach.

As mentioned in the last bullet, querying a relational database for a graph-like information, particularly traversing it, can be expensive due to the number of potential SQL joins comparing to a graph database case which is naturally optimized for the graph-related queries [14].

| Database technology features | Database technology features |
|-----------------------------|-----------------------------|
|                            | Field-oriented | DBMS          | Python API | Deployment needed | License         |
| SQLite                      | No             | Embedded      | Yes        | No                | Public domain   |
| Neo4j                       | Yes            | Embedded, Server-based | Yes        | Yes, No                  | GPLv3, AGPLv3, commercial |

4.5. Summary
We described the essence of the CondDB compatibility problem faced by the LHCb computing model and proposed a generic way of tracking the compatibility relations between the LHCb applications and the CondDB entities. This approach is based on the graph-theory vision of the problem. In particular, we proposed a simple compatibility tracking system with a graph as a basic compatibility information tracking unit which allows to store and access all needed relations elegantly. The only semi-open question left is the choice of the Compatibility database underlying technology which will need a minor investigation during the CTS development phase. But the preliminary conclusion is that the choice will probably fall on one of the graph-oriented databases the most advanced of which on the market nowadays is the Neo4j database.

5. Conclusion
We have described three CondDB operation assistance systems. Two of them, the new deployment system for SQLite databases and LHCb CondDB state tracking system, are currently in LHCb production. They have achieved the desired goals of higher flexibility and robustness for the management and operation of the CondDB. The third system demonstrated, the LHCb compatibility tracking system, has passed the design stage and is now in the implementation stage. Its elegant underlying abstraction will allow to cope with the wide set of LHCb compatibility relation categories and will ensure the higher reliability and consistency not only of the LHCb CondDB by itself but also of the common LHCb computing process led by the LHCb applications, LHCb CondDB and LHCb physics data entities.

References
[1] Clemencic M, Gilardi N, Palacios J 2006 LHCb Conditions database 15th International Conference on Computing in High Energy and Nuclear Physics Mumbai 347-350
[2] LHCb collaboration 2008 The LHCb detector at the LHC JINST 3 S08005
[3] LHCb Collaboration 2005 LHCb Computing: Technical Design Report CERN-LHCC-2005-019 LHCb-TDR-11 CERN Geneva 117
[4] Oracle database URL http://www.oracle.com
[5] SQLite database engine URL http://www.sqlite.org
[6] Clemencic M 2008 LHCb distributed Conditions database J. Phys.: Conf. Ser. 119 072010
[7] Duellmann D 2007 Oracle Streams for the Large Hadron Collider at CERN Oracle Open World San Francisco
[8] Yourdon E and Constantine L L 1979 Structured design: fundamentals of a discipline of computer program and systems design Yourdon Press
[9] Duarte A et al 2008 Monitoring the EGEE/WLCG grid services J. Phys.: Conf. Ser. 119 052014
[10] Cern virtual machine file system URL http://cernvm.cern.ch/portal/filesystem
[11] Kruzelecki K Roiser S Degaudenzi H 2010 The nightly build and test system for LCG AA and LHCb software J. Phys.: Conf. Ser. 219 042042
[12] Fast version control system URL http://git-scm.com
[13] Neo4j NOSQL graph database project URL http://neo4j.org
[14] Vicknair C et al 2010 A Comparison of a Graph Database and a Relational Database ACM SE ’10 Proceedings of the 48th Annual Southeast Regional Conference New York