Experience with the Open Source based implementation for ATLAS Conditions Data Management System

A.Amorim, J.Lima, C.Oliveira, L.Pedro, and N.Barros
CFNUL/FCUL, Universidade de Lisboa, Portugal

Conditions Data in high energy physics experiments is frequently seen as every data needed for reconstruction besides the event data itself. This includes all sorts of slowly evolving data like detector alignment, calibration and robustness, and data from detector control system. Also, every Conditions Data Object is associated with a time interval of validity and a version. Besides that, quite often is useful to tag collections of Conditions Data Objects altogether. These issues have already been investigated and a data model has been proposed and used for different implementations based in commercial DBMSs, both at CERN and for the BaBar experiment. The special case of the ATLAS complex trigger that requires online access to calibration and alignment data poses new challenges that have to be met using a flexible and customizable solution more in the line of Open Source components. Motivated by the ATLAS challenges we have developed an alternative implementation, based in an Open Source RDBMS. Several issues were investigated land will be described in this paper:
-The best way to map the conditions data model into the relational database concept considering what are foreseen as the most frequent queries.
-The clustering model best suited to address the scalability problem. -Extensive tests were performed and will be described.
The very promising results from these tests are attracting the attention from the HEP community and driving further developments.

1. Introduction

The ATLAS Conditions Database Management System, ATLAS ConditionsDB for short, is the system responsible for storing data from conditions associated with ATLAS experiment. Conditions Data reflect the conditions in which the experiment was performed and the actual physics data was taken. This includes detector calibration and alignment, robustness data (DCS) and detector description. Stating in another way, conditions data is the slowly evolvind data associated with the experiment besides event data itself. This data is produced in many different subsystems and each provides data at different rate and store objects of different granmularities. From the data consumer side the situation is similar, having consumers with many different demands concerning data rate and latency. Figure 1 depict the dataflow between the different data producers and data consummers of the Conditions Data.

The Open Source based Implementation of the ATLAS ConditionsDB described in this paper is the result of a development effort of almost 2 years. The aim of this effort was to provide an alternative solution to those that have been presented by the CERN-IT, based in comercial solutions like Objectivity and Oracle9i. It is our belief that a Open Source Solution is preferably for many good reasons:

- Free of charge.
- A growing community of programmers have a good understanding of open source technologies.
- Code availability allow fine tunning and specific optimizations when necessary.
- available for most common platforms.

We also believe it is possible and preferably to achieve the desired performance by carefuly taken design choices than by relying in the technological advanced features present in high-end commercial solutions.

Our implementation uses a RDBMS server as the underlying storage technology. The development was done using mostly MySQL, but any other RDBMS that understands standard SQL could be used. Yet the choice of MySQL was sensible. The MySQL is proven to be a reliable and very fast RDBMS, although it lacks some important features found in many common comercial RDBMS.

2. The ATLAS Conditions Data Model

in this section we shall breifly explain what is the data model used in the ATLAS ConditionsDB. In this context, data model means what model is exposed to the user/client through the API [1]. This is something diferent from the model used to actualy store the data. The later we shall designate *database schema* and will be explained later. The API ConditionsDB specification [1] was the starting point for this work.

The present ATLAS ConditionsDB model shares many of the features found in the BaBar ConditionsDB model [4]. The data is organised in a filesystem like hierarchical structure, where each folder holds a particular type of data. A version and a time of validity (IOV) is also associated with the data. Figure 2 shows the three variation axis associated with the data. No matter which underlying technology we use,

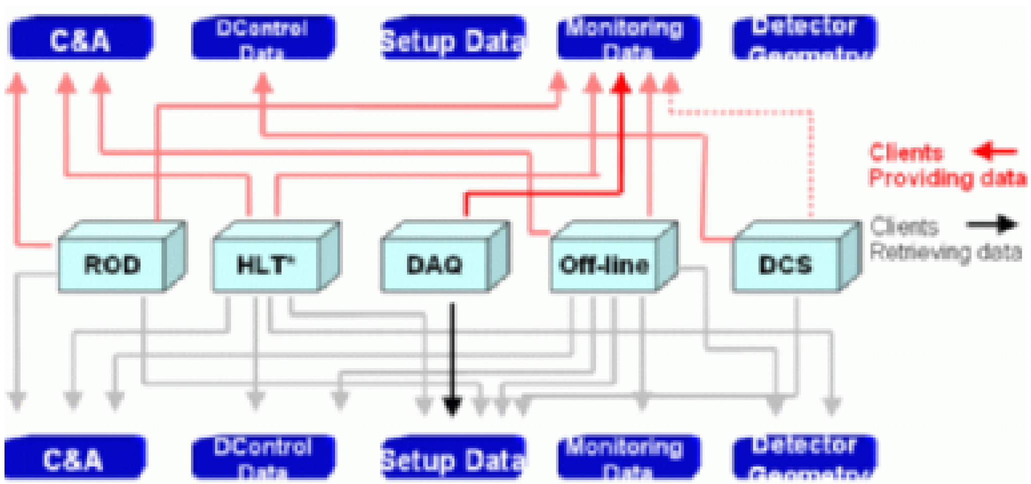

Figure 1: Clients and data providers of the ATLAS ConditionsDB

the exposed model is always the same. In our case, we use a relational model. The first problem was to understand how to map this data model that uses an hierarchical folder structure on the underlying relational model, that uses tables and columns. The second problem was to understand the tipe of queries that were being used: which were more common queries; how could one redesign the tables to make the life easier for the database server.

3. Design Aspects

In this section we will try to give an overview of the architecture and design of the Open Source ATLAS ConditionsDB Management System. The system was designed with three features in mind: backend independency; data volume scalability and extensibility. By backend independency we mean that it should be possible, and easy, to replace the underlying RDBMS, as long as it understands standard SQL. Note that we carefully stated backend independency, not technology independency, which has a more generic mean. Data volume scalability should allow us to reach, at a certain point, a $O(1)$ performance as function of data volume. It also means that should be easy to switch the data, in small chunks, back and forth from tertiary data storage. Extensibility means that the software system must be modular with well defined interfaces in order to improve maintainability and extension of functionality with new modules. Next we will show how we achieved each of these features.

3.1. Data volume scalability

This issue is closed connected to the overall performance of the system, Nevertheless, it should be stressed that no post design optimizations can help to get rid of a O(n!) behavior. This is an obvious fact, although so often completely ignored. Yet our approach to this problem was not to devise a full O(1) compliant design from the beginning. We took what we can call a successive approximation with design/test/redesign approach. The drawback of this approach is that one takes a lot of effort in design, knowing from the beginning that it will be thrown away. Lets be honest with ourselves, we end up throwing away a lot of work even when we do not assume that fact from the beginning.

The task of making a Database management system that can scale well to the data volumes expected from an experiment like ATLAS pushes the limits of our imagination. Every thing that can be done to improve performance must be done. Not only the database schema must be optimum for the problem in hands, one must understand which data clustering model is best suited for the Conditions Data. Our implementation supports a data clustering model which should scale very smoothly over large data volumes. By design the clustering can naturaly be made in a per folder basis. Moreover, our implementation supports data partitioning inside folders in a time or version basis. Thus, the data partitioning mechanism is highly configurable and flexible enough to adapt be adapted to several situations.

3.2. Backend independency

Willing to use Open Source also means willing to get rid of supplier's dependence. Using standards instead of proprietary or specific vendor features is a good thing, anyway. Unless there was a very good reason to stick with a specific backend technology e would always prefer to be able to switch to a different backend whenever we feel like. For the time being almost all the development has been done with the MySQL backend, but plans exist to test our implementation

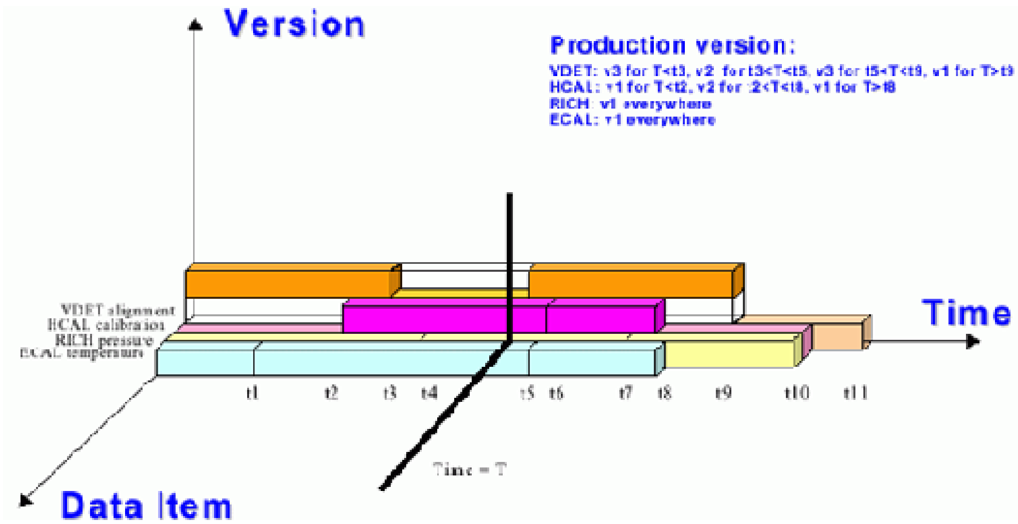

Figure 2: Conditions Data variation axis.

with other RDBMS backends and even two support different backends in the same implementation. A successful experiment have already been made that used the Postgress as a RDBMS backend.

3.3. Extensibility

A modular design policy was implemented since the beginning. The software is composed of several layers and each accomplishes a particular task. This design has made possible, or at least easier to implement new features and to maintain the code in a cooperative development environment. Figure 3 depicts the modular software architecture as it is for the latest version.

The recent developments that will be described in this paper on section 4.

4. New Features

4.1. New database schema

The new database schema was designed in order to allow extension of the API to support new features. Notably the Tiny Objects. A considerable redesign of the database schema was also needed in order to improve storage performance. The results from comparison tests before and after redesign are presented in section 5. Figure 4 shows the table schema as it was at time or writing this paper.

4.2. Support for tiny objects

The *Tiny Objects* in the context of the ConditionsDB were proposed few time ago to the CERN ATLAS community in response to a requirments expressed from users whom wished to store objects with simple structures but with a huge granularity. The best example of objects of this kind are the data coming from the DCS part of the detector. To deal with this type of objects the API was extended in order to handle the `PVSS` interface.

What the people from DCS needed was a way to efficiently store and retrieve very small data objects, typicaly, integers or floats. The idea of *Tiny Objects* extension is to provide such way at the cost of droping some general features. Features which, after all, are not need in this case.

The DCS objects are often simple scalar object representing the ddp or the temperature in certain detector elements. This are very small object that will get a huge storage overhead if stored with the primitive ConditionsDB approach. On the other hand, people from the DCS don't foresee the absolute need for using versions for their objects, only the interval of validity.

Starting from version 0.3, the API features storage and retrieving of tiny objects. That feature actualy match a user requirment. Remarkable is the fact that people from the DCS are already using this brand new *Tiny Objects* extension in the test beam at CERN.

4.3. TCL Test Framework

The first tests performed on this particular implementation of the ATLAS Conditions Database were based in the test suite from the Oracle based ITs implementation. The main purpose was then to evaluate in what extent the behavior of the API was the same regardless of the underlying implementation. Yet these tests were inappropriate for an optimization phase: they do not fully explore the eventual weaknesses and bottlenecks of the implementation and, on

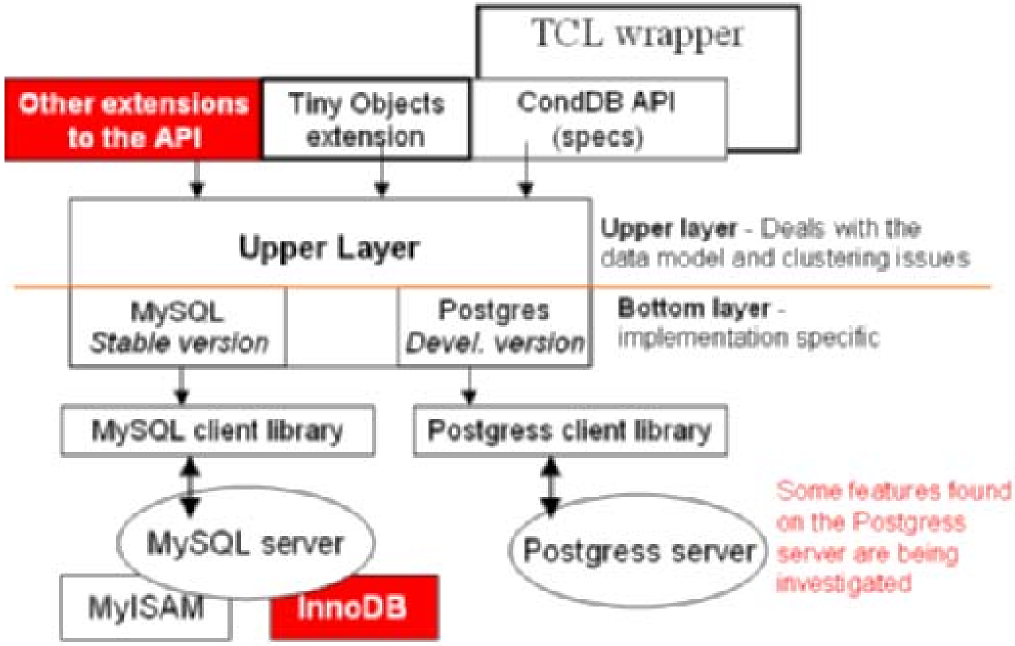

Figure 3: System software layout.

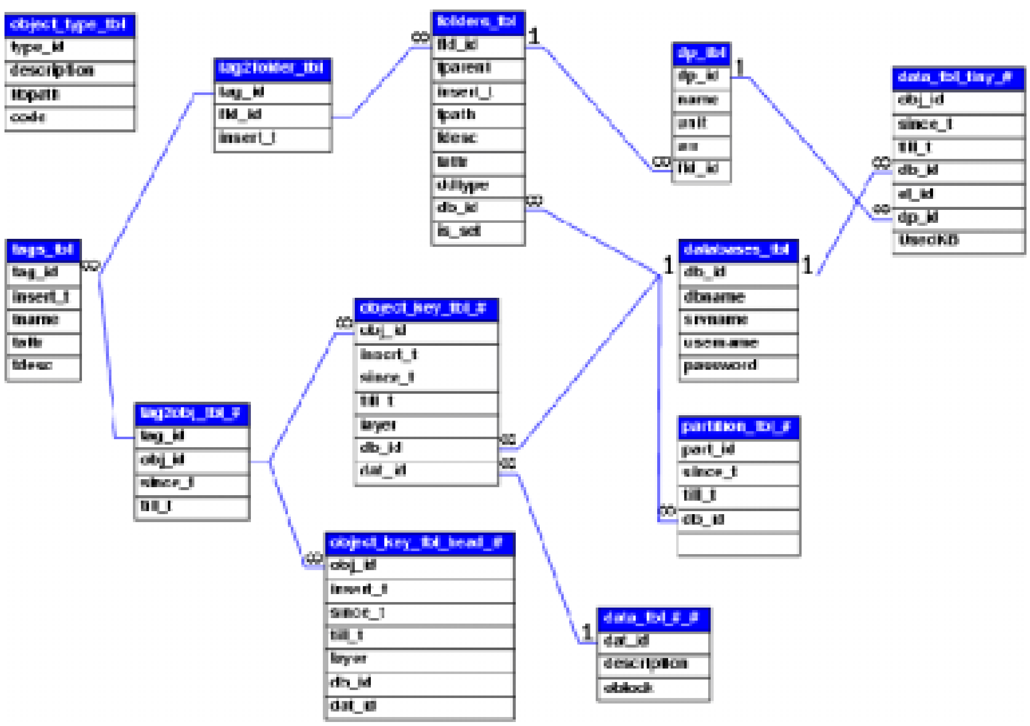

Figure 4: The new database schema.

the other end, they do not reproduce the foreseen run-time operations. Thus, recently, a considerable effort was devoted to fully understand the system in terms of performance. Soon become clear that we needed a flexible, highly configurable, test framework. More like a scripting system. So, wrappers for the CondDBs C++ API classes were developed for TCL, giving rise to a complete CondDB scripting system.

To illustrate the idea we reproduce a simple test program, that opens a database and create a folder structure, both in C++ and in TCL. Immediately becomes clear that the TCL version is more concise, besides being less error prone and having the advantage of not requiring the usual compile/link cycle.

The exception handling code was removed from both, the C++ example and the TCL example for sake of clarity.

a. C++ example

```
// File: "basicSession.cxx"
//
// Created at Mon Aug 26 18:25:22 WEST 2002
// by Jorge Lima, portuguese TDAQ group,
// to the atlas collaborations.
//
// Based on the CERN IT Objectivity Implementation.
//

#include <ICondDBMgr.h>
#include <CondDBMySQLMgrFactory.h>

#include <string>
#include <iostream>
using namespace std;

int main ( int argc, char* argv[] )
{
    ICondDBMgr* CondDBmgr =
         CondDBMySQLMgrFactory::createCondDBMgr();

    CondDBmgr->init();

    // Create a ConditionsDB

    CondDBmgr->startUpdate();
    CondDBmgr->createCondDB();
    CondDBmgr->commit();

    CondDBMySQLMgrFactory::destroyCondDBMgr( CondDBmgr );
    return 0;
}
```

b. TCL example

```
#!/bin/sh
#
# TCL Wrapper for ConditionsDB
# Lisbon ATLAS-DAQ Collaboration
# Ported from C++ to TCL by Jorge Lima 2003/04/15
#\
exec tclsh "$0" "$@"

package require conddb 1.0

set CondDBmgr \
     [CondDBMySQLMgrFactory::createCondDBMgr]

$CondDBmgr init

# Create a ConditionsDB

$CondDBmgr startUpdate
$CondDBmgr createCondDB
$CondDBmgr commit

CondDBMySQLMgrFactory::destroyCondDBMgr $CondDBmgr
```

4.4. Other uses for scripting

Although initialy developed as a test framework, the scripting bind to the ConditionsDB has a very broad scope. It is important to remark that no performance penalty is noticeble by using the scripting interface to the conditions DB instead of C++. Of cource, the scripting environment is not intended to replace the C++ programming environment, and as more complex object manipulations are necessary the scripting environment might became rather limited. Yet as far as storing, retriving and browsing is concerned a TCL program is as fast as a C++ program. For example the ConditionsDB TCL/TK browser, which used to interact with the ConditionsDB through a command line based tool written in C++, can now be completly rewritten in a very straight forward way. For very long that users made suggestions regarding this browser in order to make of it a more usefull tool. Work is in progress in this area and it will hopefully possible to provide very soon a complete and fully functional browser using the TCL API. Figure 5

Moreover, the experience acquired while building this scripting environment will make the job easier if we happen to need to write bindings for other scripting language like, for instance, *python* or *php*.

The TCL API also includes tools for performance measurement, which make this API specialy suited for building performance tests.

5. Performance Test Results

From time to time we have presented performance comparisons between our implementation of the ATLAS ConditionsDB and the reference ORACLE based implementation supported by the CERN-IT. Soon it became clear that this was a development that was worth being investigated further. Even before any optimizations or use of indexes, our implementation with the MySQL backend, outperformed the ORACLEs one by a factor of 10 to 50 for most common operations, as shown in table I.

There was still a problem to be solved: we identified a worst case for storage operations where the performance was unusualy poor. Moreover, storing an object took $O(n)$ time where n were the number of objects already stored in the table. Although its clear that, in the ConditionsDB a storing is complex operation (involving far more SQL queries) than a fetch operation, there was no trivial explanation for such a different behavior between this case and the storage in

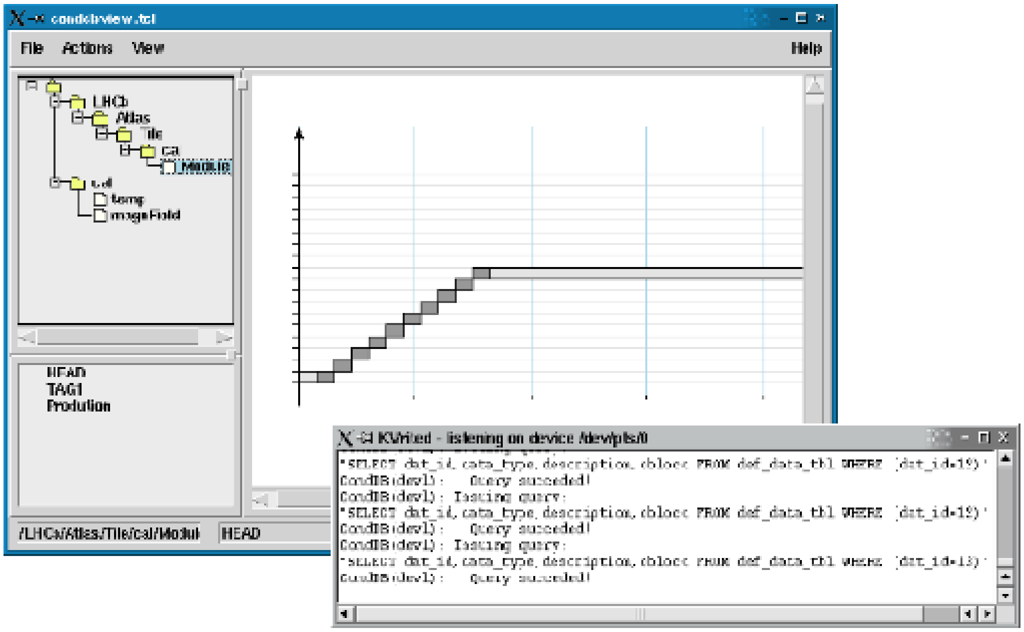

Figure 5: System software layout.

Table I Comparisons between schema

	Oracle (local)	MySQL (local)	Oracle (remote)	MySQL (remote)
createFolderx	0m15.173s	0m0.034s	0m11.857s	0m0.072s
storeDatax 10	0m4.973s	0m1.447s	0m5.434s	0m1.127s
storeDatax 100	0m9.749s	0m1.368s	0m15.820s	0m2.345s
storeDatax 10.000	9m22.103s	0m23.175s	11m12.554s	0m56.929s
storeDatax 100.000	109m40.878s	3m49.184s	104m13.283s	8m16.510s
storeDatax 1.000.000		23m12.563s		40m48.256s
readDatax 10	0m0.324s	0m0.025s	0m0.955s	0m0.058s
readDatax 100	0m1.403s	0m0.050s	0m3.061s	0m0.135s
readDatax 10.000	2m1.919s	0m2.851s	4m37.315s	0m7.926s
readDatax 100.000	25m46.423s	0m27.273s	46m53.846s	1m18.850s
readDatax 1.000.000		4m40.315s		10m26.124s

other situations. This test was not performed in Oracle9i based implementation but, after investigating the problem, we believe this was a problem specific to our implementation.

The problem was successful identified and corrected. Now the store time is nearly constant with respect to the number of stored objects. Though, this implied, not just simple optimizations, like indexes or caching policies, but rather a schema redesign. The system is converging to a $O(1)$ performance. There are, however, some operations that still must be tuned. We can even expect complete table schema redesign in order to cope completely with the $O(1)$ goal.

Current performance comparisons between our ConditionsDB implementation, prior and after the schema redesign are shown in tables 1 and 2. It must be stressed that the new implementation also includes the extensions mentioned in section X. So, the new implementation is, both, more functional and is closer to the $O(1)$ goal.

In the table for different results are shown, two for store operations an two for read operations. The *Store A* test stores objects having the same interval of validity while the *Store B* test stores objects with overlapping intervals of validity, with monotonicaly increasing start times. This was identified as aq worst case situation for the store operation using the old database schema. The *Read A* and *Read B* tests iterate over the objects stored with the *store A* and *store B* tests respectively.

Table II Performance test results with old table schema

N. obj	Store A	Read A	Store B	Read B
10	0.030540	0.007037	0.036987	0.010162
100	0.277502	0.134629	0.851968	0.219335
1000	5.393425	0.784574	9.428609	0.772833
10000	25.127041	3.409653	607.253235	7.694590
50000	134.046234	17.873844	14631.000000	35.907280

Table III Performance test results with new table schema

N. obj	Store A	Read A	Store B	Read B
10	0.037271	0.008258	0.035450	0.010978
100	0.196226	0.031853	0.318195	0.062430
1000	2.180148	0.263181	3.608150	0.651171
10000	22.151493	2.572790	44.693687	6.285625
50000	115.495209	14.725236	230.663757	33.950783

6. Future developments

The ConditionsDB is a rapidly evolving subject. In the last months, not only considerable human resources have been allocated to this development effort which resulted in many improvements in several areas, but also, the HEP community is getting aquired of the true dimension of the problem and has started to contribute with their feedback. Thus, it's not an easy task to foresee what will be the future developments in this area. Nevertheless the next few modifications will most likely to happen.

6.1. New tagging mechanism

The redesign o the tagging mechanism will be the next milestone. As pointed out several times [Malon], the current tagging mechanism is, both, incomplete and unreliable. This is not a problem specific to our implementation. It is also present in the ORACLE's implementation as it was in the Objectivity's one. The problem lies in the original API specification [CondDBSPec] that is dated back to XX-XX-1999. Changing this situation would require that we rewrite the API specification.

It is incomplete because doesnt allow tagging of other objects besides head. This is contrary to what is stated in the requirements collected so far [Req]. Several scenarios for tagging were presented by [Malon] that cannot simply be accomplished by the current implementation.

It is also unreliable, because it can effectively lead to data loss, as no other way to reference a particular version exists besides referencing it by head or by a tag. If after inserting some objects one user wishes to tag the head for later reference, he/she can end up tagging a completely different version, because some other user, in the meantime, inserted new object, completely replacing the head.

A proper resolution to this problem necessarily passes by rewriting the API specification with the involvement of all the community. Yet, although many times stressed the importance of this question, no significant advances have been seen from that side. A much ruder approach will be to develop a working solution, get the feedback from users, iterate a few times and finally, if we happen to converge to a viable solution, write the specification. The later approach will be taken for practical reasons.

6.2. Expandable Objects

The concept of *Expandable object* will be soon introduced. Briefly this are objects for which schema is known at the DBMS level. This means that any client can query the DBMS about the object structure and get as reply a platform independent representation.

The implementation for *Expandable Objects* will benefit from the work done for the *Tiny Object*

In the present situation two main frameworks for Memory Databases exist in the ATLAS context. Each use their own object schema. The Online Memory Database is based in the OKS, and the Offline Memory Database is based in the Athena Transient Store. Typicaly, the Memory Databases will interact with the ConditionsDB storing their objects as BLOBs. Anybody from outside a given framework will be completly unable to understand objects stored from within that framwork.

The *Expandable Objects* concept will allow any client to browse through the object schema because the database management system will be schema aware.

6.3. Minor issues

Work has still to be done in many different areas. Now that most of the performance bottlenecks have been removed, more realistic tests involving many clients can be performed. Different multi-server configurations can be experimented in order to understand in which direction to go.

For a long time that an $< XML >$ integration in the API as been deferred due to other priorities. An embedded $< XML >$ parser can be an effective way of configuring the database in a way that is very extendible.

7. Conclusions

The performance results shown in this paper prove that the approach chosen for this project is a possible way to follow. I would dare to say: is the way to follow. It is at least, far from any doubt, a way it is worth investigate further. Furtermore:

- The clustering model devised, which should scale well over data volume and time , can be used with almost any kind of RDBMS backend.
- Porting between defferent backend implementations doesn't contitute a major effort as verified while porting the original implementation to Postgress.
- The MySQL implementation outperformed the Oracle's one by a factor of 50 on most usual operations.

There is still a long way to go to be able to make a stable product. And making this a stable product can take far more resources than it took to design it. Yet it involves far less risk to allocate human resources to this project now that we are more confident of its usefulness.

Acknowledgments

The authors wish to thank all the ATLAS-TDAQ community by their feedback and valuable suggestions. A special word of thanks for David Malon for the way he clearly pointed out some important design flaws. Thanks to those contributing actively to the T-DAQ ConditionsDB requirements [8] effort.

References

[1] Stefano Paoli, "Conditions DB Interface Specification", CERN- IT Division

[2] A. Amorim et. Al., "The HERA-B database services", Computer Physics Communications 140 (2001) 172-178

[3] "The MySQL database manager", http://www.mysql.com/

[4] Igor Gaponenko, "An Overview of the BaBar Conditions Database", CHEP2000 Conference, Padova Italy, January 2000.

[5] N. Gaspar, A. Amorim, C. Oliveira, L.Pedro, "Exporting Internal PVSS Databases", ATLAS DCS Note, DCS-IWN18, http://atlas.web.cern.ch/Atlas/GROUPS/DAQTRIG/DCS/iwn.html

[6] A.Amorim, N.Barros, D.KLose, J.Lima, C.OLiveira, L.Pedro, "Evaluation of the Relational Implementation of the Conditions Database Interface", CERN, Feb 2003 (to be published).

[7] A.Amorim, N.Barros, D.KLose, J.Lima, C.OLiveira, L.Pedro, "Manual for the extended ConditionsDB interface for ATLAS", CERN, May 2003 (to be published).

[8] "Requirements on the Conditions Database interface to T/DAQ" http://atlobk01.cern.ch/ConditionsDB/requirements/