# CDB - DISTRIBUTED CONDITIONS DATABASE OF THE BABAR EXPERIMENT

I.A. Gaponenko[*], D.N. Brown[#], for the BaBar Computing Group,

LBNL, Berkeley, CA 94720, USA

*Abstract*

This document presents CDB – Distributed Conditions Database of the BaBar Experiment [1]. CDB is the second major iteration of the database deployed in BaBar for production use as of October 2002. It replaced the original version of the database [2] used through the first three years of the data taking. The new design and its implementation are aiming at performance and scalability limitations of the original database, as well as at emerging challenges of a distributed data production and analysis system [3] of the Experiment.

## INTRODUCTION

The Conditions Database is used in the BaBar experiment to store time varying data about hardware and software environment (hence: *conditions*) in which detector data (or *events*) get acquired, modelled, processed and analysed. The kinds of information stored in the database include: detector alignments, various constants, electronics wiring maps, calibrations, to count just a few. The same database (both software and the data) is used through all distributed events processing chain of the Experiment [3], both by ON-LINE and OFF-LINE systems. Database update patterns differ from an application to an application. For certain types of conditions (for example, calibrations), the contents of the database gets updated for every run, meanwhile for others (alignments, constants) - it only happens once a month or even once a year.

Up to date, the database has gone through two major iterations. The original version of the database was commissioned by May 1999, by the time when BaBar started data taking. The original database was implemented using Objectivity/DB [4] as an underlying persistent technology. During its life span lasted through June 2002 that database saw a number of evolutionary improvements not touching foundations of its design. By the year of 2001 it became obvious that, mostly due to limitations of the original design, the evolutionary approach won't allow further development of the product to address emerging challenges of the Experiment. In particular, one of the major issues was that the database was not specifically intended to be used in a distributed environment [5]. The second major problem was the database API, which was exposing internal implementation of the database. And finally, we hit a wall of the performance and scalability limits [5, 6].

The second generation database (its codename was CDB) was designed, implemented and tested within one year by the summer of 2002. A migration of clients' code to the new API was accomplished during the accelerator shutdown in summer 2002. CDB was finally deployed for production use by October 2002.

CDB introduced a new conceptual model of the Condition Database and a brand new API. CDB was also designed as a *distributed database* from the ground up.

To facilitate a smooth migration of the BaBar experiment from the old database to the new one, CDB was first implemented using the same persistent technology the original one was based upon – Objectivity/DB. That allowed reusing an existing user-defined schema and persistent objects stored in the database. Though, these objects had been re-clustered in the new database to comply with new distributed model of CDB.

## AN OVERVIEW OF CDB

### *Distributed Database Design*

Perhaps a biggest challenge in building a data processing system of a contemporary HEP experiment is how to make it working in a (quite often geographically) distributed environment. In a specific context of the Conditions Database it means three major problems:

- The contents of CDB may be simultaneously updated in disjoined database installations. Hence we have a problem of a *consistent merging* of new data into the rest of the database to avoid problems like namespace conflicts.
- The data are also used in the same distributed environment. Each specific database installation may not have all the data known in the distributed database, neither these data should necessarily be up-to-date, but we do expect the right data to exist at a point of its use. This leads us to two problems: *availability* (of data) and *usability* (of a specific installation).
- *Synchronizing* the contents of multiple database installations is another issue. That's about a dynamics of a living distributed system. To accomplish this goal the corresponding protocols and (problem domain specific) data flow scenarios have to be envisioned.

*electronic address: IAGaponenko@LBL.Gov
#electronic address: Dave_Brown@LBL.Gov

A cornerstone concept of a distributed model of CDB is the *origin*. CDB is made of unique *database installations* each associated with its *native origin* (see Fig. 1).

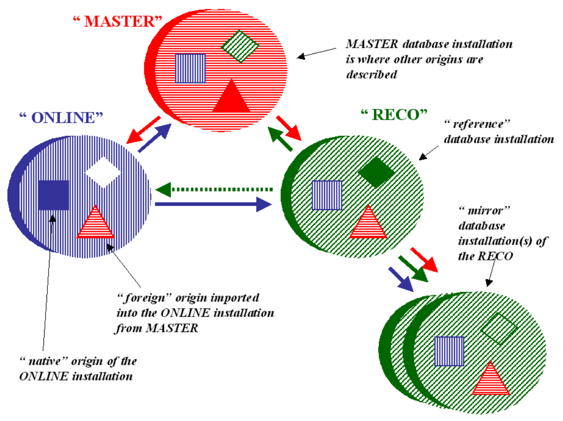

" MASTER"
MASTER database installation is where other origins are described
" ONLINE"
" RECO"
" reference" database installation
" mirror" database installation(s) of the RECO
"foreign" origin imported into the ONLINE installation from MASTER
" native" origin of the ONLINE installation

Figure 1: Distributed model of CDB.

Origins provide a scope for persistent data *originating* at the corresponding database installation. These data include: *conditions*, *partitions* and *views* (see Fig. 2).

Data associated with a specific origin can only be updated at a database installation of the same origin. It means that origins (not database installations!) *own* their data. A particular database installation may also have data coming from others, or so called *foreign origins*. That data are always available in read-only mode.

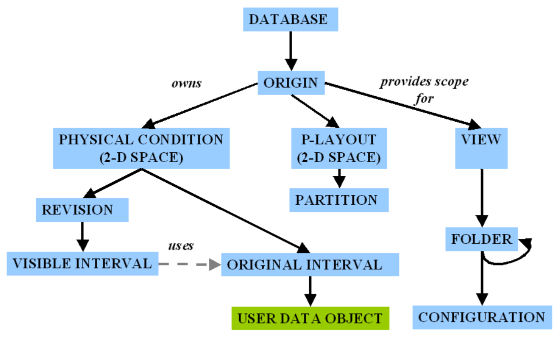


Figure 2: Scope and ownership diagram.

In addition, as it's shown on the Figure 1 above, CDB allows establishing exact copies (mirrors) of the corresponding unique database installations (reference). Each mirror is associated with the same origin as its reference. However, all the data in a mirror are only meant to be used for reading. The first application for mirrors is to resolve performance bottlenecks in highly parallel processing scenarios, for example, when multiple physics analysis jobs are using the same type of a database installation. The second target is geographically distributed copies of the same installation (the above mentioned example of analysis jobs can also be a good example).

## *2-D Space of Conditions*

For individual *conditions* (a specific CDB term for calibrations, alignments, etc.) CDB introduces a simple geometric model in which conditions are containers providing 2-D space of *insertion* and *validity* timelines for *condition objects* (see Fig. 3).

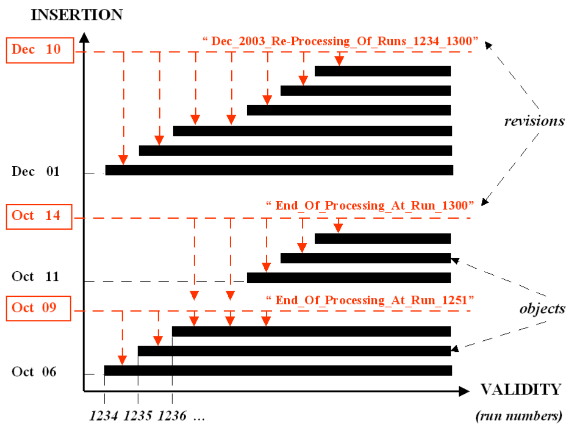

INSERTION
Dec 10
" Dec_2003_Re-Processing_Of_Runs_1234_1300"
revisions
Dec 01
Oct 14
" End_Of_Processing_At_Run_1300"
Oct 11
Oct 09
" End_Of_Processing_At_Run_1251"
objects
Oct 06
VALIDITY
1234 1235 1236 ...
(run numbers)

Figure 3: 2-D model of a condition.

Each object is shown at the picture as a bar. It has an interval of validity (*validity interval*) and an insertion timestamp. These objects representing original user intentions are known in CDB as *original intervals*.

Objects get resolved from certain point of the insertion timeline, which is known as *revision*. Revisions separate what was stored in a condition before from what may be stored after. The objects lookup process goes from the top (of a revision) down to the bottom (of the condition). The first object intersected is the one reported. A client gets only a "visible" part of the found original interval. That part is known in CDB as *visible interval*. Visible intervals are associated with revisions.

Revisions must be created before to use them. They're identified either by their user-assigned names or by their timestamps at the insertion axis. Each condition always has a predefined revision – the *topmost* one. A timestamp of this revision is the +Infinity on the insertion time axis.

The above shown picture also provides an illustration of how three revisions can be used to access objects produced after three stages of an event reconstruction.

## *Distributed Conditions and Partitions*

Sometimes it's required that different validity ranges of the same condition were simultaneously updated in two or more disjoined database installations. That poses a problem of merging new condition objects stored in these installations. This kind of conditions represents so called *distributed conditions*. A good example of this can be "rolling calibrations" in the BaBar Experiment [7].

A CDB way to address this requirement is to define non-overlapping subspaces in 2-D space of these (distributed) conditions and assign each subspace to a dedicated origin. The corresponding database installation will be allowed to put new objects into the subspace. These subspaces are known in CDB as *partitions*. All distributed conditions are partitioned in the same way. The MASTER origin has a special data structure ("Partitions Layout") to maintain partitions.

Partitions are owned by origins. An example of using partitions to produce calibrations for initial reconstruction and a subsequent re-reconstruction is shown below:

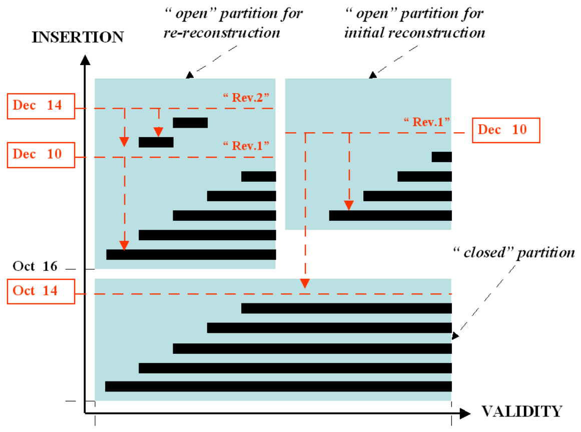


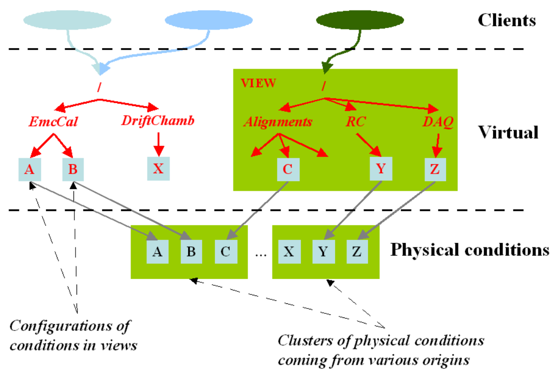
Figure 4: Partitions.

Once created a partition is assigned a sub-range of the validity time. And it's also "open" from the top meaning that new objects can be added into its subspace. When the corresponding activity (reconstruction) is over then the partition can be "closed", so that new partition(s) can be created above it.

For normal client applications reading from the database, partitions are transparent. Also, in distributed conditions, the scope of revisions is restricted to condition and a partition.

## *Views and Configurations*

In order to insulate client applications from knowing exact details on how conditions are stored in the database, and to add extra flexibility for the database management, CDB has two-layered namespace for conditions:



Clients
/
EmcCal
DriftChamb
A
B
X
VIEW
/
Alignments
RC
DAQ
C
Y
Z
Virtual
A
B
C
...
X
Y
Z
Physical conditions
Configurations of conditions in views
Clusters of physical conditions coming from various origins

Figure 5: Namespaces for Conditions.

These two layers are:

- *virtual* represented by *views*
- *physical* represented by *physical conditions*

The virtual layer of views is the one client applications directly encounter when dealing with the database. A view provides a hierarchical namespace for conditions as well as *configurations* for each condition in the namespace. A role of configurations is to represent the corresponding physical conditions in views acting like symbolic links and also to put restrictions on how the physical conditions can be used through the views. In the latter case a configuration uses the previously described mechanism of revisions to specify which revision (partition) should be used when looking for objects at a condition. The general assumption is that an expert or a database contents manager would prepare a view with consistent configurations of conditions, give it (the view) a name and let users to use the view without worrying about knowing explicit secondary keys (views) for each condition.

Views are owned by origins. In the scope of its owner origin, each view has a name and an identifier. The most recently created view of the native origin of a database installation becomes a *default view* of the installation. The default view is the one picked by applications if they don't specify explicitly which view they're interested at. This mechanism allows to specialize database installations for specific uses (for example, ON-LINE) rather than configuring applications.

On the other hands, the lower physical layer is organized in most optimal way for managing and distributing the contents of conditions. The conditions at this layer come from various origins.

## *State Identifier*

Another new feature introduced in CDB is *state identifier*. It's a small (64-bit) data structure uniquely describing a state of the database as it's seen by a user application. The state identifier can be used as a CDB-wide secondary key (in addition at the validity time) for the contents of the database. In BaBar, a stack of identifiers is stored in an event header. A new identifier gets added to the stack every time the event undergoes through a modification (for example, re-reconstruction) to record a configuration (and the contents) of CDB used for the modification. This information can be used to recover the state of CDB at each stage of the event's life.

## *API*

The new CDB API reflects new conceptual model of the database. It's mostly (persistent) technology neutral, allowing multiple underlying implementations to be used by an application. The persistent technology specific extensions of the API, which are needed to handle user defined payload, are confined within well controlled converters and factories.

## USING CDB IN BABAR

At a time when the current document was being written, there was just one implementations of the CDB, which was using Objectivity/DB as an underlying persistent technology. There is also an ongoing work on implementing CDB in MySQL [8] and ROOT I/O [9].

The schema of the Objectivity/DB based CDB includes about 50 persistent classes representing metadata and over 400 unique classes representing user defined payload (the actual contents of the database). There was also a successful effort to provide users with predefined table-like persistent containers with a transient interface.

The total amount of data in the database is over 32 GB. There are 8 core origins, 40 views, 20 partitions and about 2000 of physical conditions.

### *CDB Installations*

Here is a map of the distributed database, in which database installations are grouped into four major applications:

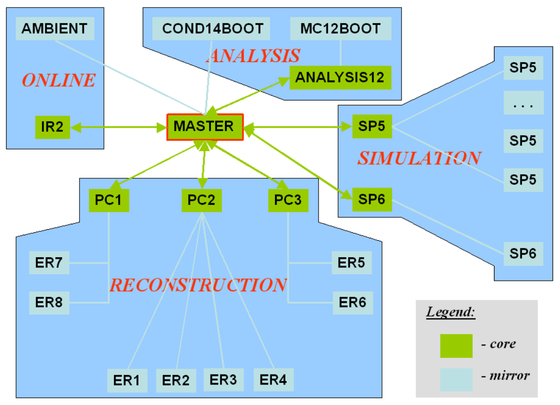


Figure 6: Map of the Distributed CDB.

The overall number of installations exceeds 40. All core installations are located at SLAC. The cross-database synchronization tasks for all core installations and for many reference ones are fully automated.

### *Problems*

A biggest problem inherited by CDB from the first generation Conditions database was a complex user defined schema (400+ classes). At that time there was a very little control of how conditions developers did the data modelling. Obviously many use cases could easily be solved with predefined containers (like tales). This schema significantly complicates any implementations of the database on non-OO based persistent technologies (that's why ROOT I/O has been chosen to replace Objectivity/DB for the user defined schema and database payload).

As to new problems, then it wasn't a surprise that a richer logical model has come with a price:

- non-negligible database management efforts are put into synchronizing distributed database installations, even though the Distributed Model of CDB has the corresponding provisions
- database contents management (configuring views) requires more attention from conditions developers and database administrators

The latter sometimes becomes a source of troubles when mistakes are made at database configurations. A general solution to the last two problems is an automation of as much database management and data distribution operations as possible to eliminate a "human (mistake) factor".

## CONCLUSIONS

A major rework performed on the database took 15 months from its start to the final deployment in October 2002. CDB got an advanced conceptual model in its foundation as well as a brand new API. The performance and scalability limitations of the original database were also resolved. The subsequent two years of using CDB in production have proven the correctness of the new model. Valuable lessons have been learned in a course of using and developing the Conditions database for the Experiment.

The most important outcome was that CDB has significantly extended BaBar's ability to process event data in a distributed realm of the Collaboration.

## REFERENCES

[1] D. Boutigny et. al., "*BaBar Technical Design Report*", SLAC-R-95-457.
[2] I. Gaponenko, et al., "*An overview of the BaBar Conditions database*", CHEP 2000, Padova, Italy, January 2000.
[3] P. Elmer, et al., "*BaBar computing – from collisions to physics results*", CHEP 2004, Interlaken, Switzerland, September 2004.
[4] http://www.objectivity.com
[5] I. Gaponenko, et al., "*The BaBar database: Challenges, Trends and Projections*", CHEP 2001, Beijing, China, September 2001.
[6] J. Becla, I. Gaponenko, "*Optimizing Parallel Access to the BaBar Database System Using CORBA Servers*", CHEP 2001, Beijing, China, September 2001.
[7] J. Ceseracciu, et al., "*Distributed Offline Data Reconstruction in BaBar*", CHEP 2003, San Diego, USA, March 2003.
[8] http://www.mysql.com
[9] http://root.cern.ch