Conditions DB

MySQL Based Backend Implementation

Document Version: 0.2
Document Date: 3/15/02
Document Status: draft
Document Author: Jorge Lima

Abstract

This document reports the development status of a MySQL based implementation of a Conditions DB API.

1 − Introduction

The implementation of this alternative backend for the Conditions DB was driven by our belief that, due to the real time demands imposed to the online software, it is important to try different implementation approaches, to test and benchmark them. Also an open source solution for the client API allows compilation in DAQ specific platforms like LynxOS.

We have chosen a MySQL based implementation because we already have a great deal of experience using MySQL and also because MySQL is a light−weight DBMS, which contrast with the other full−featured commercial ODBM'S (Objectivity and Oracle) being used for other Conditions DB implementations.

This experimental work is being based in the existing Objectivity implementation [2]; this approach is more straight−forward than to start it from scratch and should allow trivial migration for the client applications.

2 − Database Schema

The API doesn't force us to use a particular database schema, so we've started the development by establishing a database schema which optimizes what we think it will be the typical online queries: queries for some particular condition or set of conditions at a given point in time. A very important requirement is that this schema should also allow a smooth scalability for a time increasing number of objects .

We discuss the features of our implementation from the point of view of each of the interfaces provided by the API.

2.1−Object Management

For object management we mean the operations available through the **`ICondDBDataAccess`** interface. These include the most frequently used operations in the online environment.

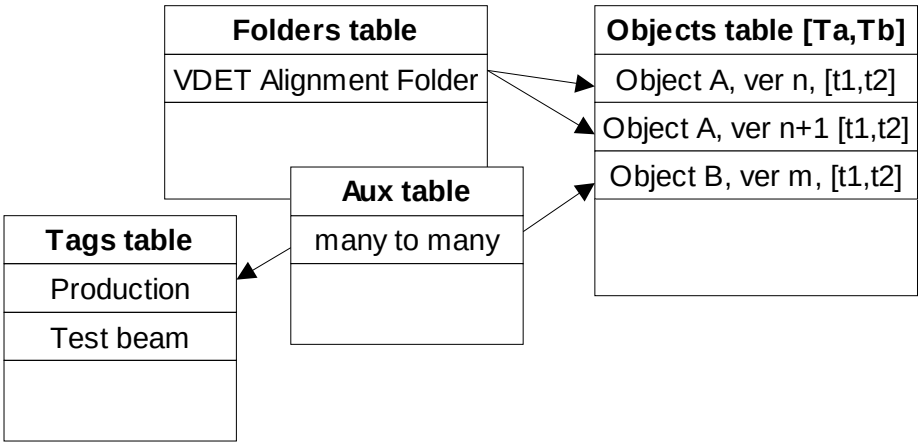

Figure 1: Conditions DB Schema in the MySQL implementation

In the database schema of figure 1 the conditions objects are all stored in a single table regardless of which folder or folderset they belong. This means that to fetch a particular object we need a single query and that only one table must be searched. This scheme, however, has the drawback of becoming very inefficient as the table grows−up. Of course we could use indexes for the validity time fields to increase the search speed, but we still have another problem in this scheme: it doesn't scale very well over time.

To solve the object retrieval efficiency and the time scalability problems at once we introduced a small change in the database schema. This change comprises an additional **time table** and the partitioning of the **objects table** into smaller tables each one valid for a specific time interval.

The new schema is shown in figure 2. Now to retrieve a particular object, the initial query is split into two but, hopefully, faster queries: first a search is performed on the **time table** for a particular time interval, then the **objects table** corresponding to that time interval is searched for the specified time point.

This table partitioning system causes data duplication whenever a configuration object

crosses a time interval boundary. Nevertheless we expect it reduces substantially the typical object table size, provided the time intervals are wisely chosen.

Open questions: how should the time partitioning be performed?

Should it be triggered automatically by table size?

Should it be based on fixed time intervals?

Or should an extension be provided to the API for this purpose?

Should it be performed by external administration tools?

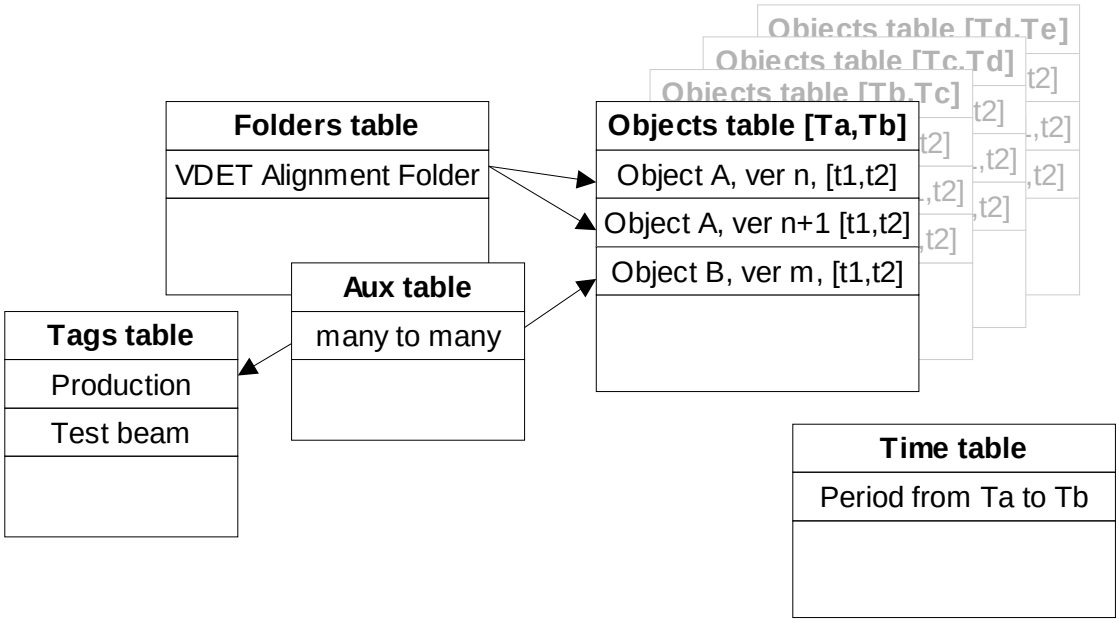

Figure 2: Conditions DB Schema using time interval partitioning

2.2−Folder Management

This is the interface provided by the **`ICondDBFolderMgr`** class. It is essentially an administration interface and it was implement in a straight forward way.

2.3−Tag Management

This is the interface provided by the **`ICondDBTagMgr`** class.

To be implemented

3 − Development Status

3.1−Database schema

Status: *stable.*

3.2 − Database time partitioning

The code for database time partitioning scheme is not yet written.

3.2−Specific implementation classes

This is the structure of classes derived from the interface which are particular to each implementation. This implies the C++ code but not the actual queries.

Status: *completely coded, all the code, including the examples compile and link successfully. The examples run as expected but more tests must be performed*

3.3−Code for queries

This code hides the MySQL details and the schema details from the upper API layers.

Status*: about 80% of the code is written (the code for time partitioning in multiple object tables is not yet written). The code compiles and works correctly. We plan to rewrite this code to improve maintainability and extendibility.*

4 − (Still) Immature Ideas

4.1−Client side BLOB preprocessing.

To store an arbitrary object (Binary Large OBject) into MySQL one use the BLOB column type. When passing the data in the SQL query, we must replace all occurrences of a specific characters by escape sequences. In the worst case, the escaped string will have twice the length of the original string. This length overhead, however, won't be reflected in the storage, only in the query sent to the server. Nevertheless it can represent a waste of bandwidth. Then, on the server side, escaped string is reverted to the original string and after compressed to reduce storage space.

My feeling is that there are, in the whole process, several unnecessary steps and that this approach is far from optimum when the bandwidth (more than storage) is at premium.

I propose the following approach to store arbitrary (binary) objects:

1. During object storage, in the client side
    1. Decide whether should or should not use compression, which compression algorithm and, if so, compress the data.
    2. Encode the data from the previous step in ASCII text. (The ASCII must not contain linefeeds or carriage returns.)
2. And, in the server side
    1. Store the data in a bare TEXT column (to avoid character escape processing)
    2. Disable the data compression feature of MySQL server.
3. During object retrieval the process is reverted.

This approach has the following advantages when compared to the current one

- It frees the server from the overhead associated with escape processing and compression
- Frees network bandwidth because the objects are exchanged in their final storage size.
- Lets the client, which has some knowledge about the nature of the data, to decide which compression scheme to use.

References

[1] − Stefano Paoli, Conditions DB Interface Specification

[2] − R.D. Schaffer, ATLAS Database Basics