CLEO III Data Storage

M. Lohner[1], *C. D. Jones*[1], *Dan Riley*[1]

Cornell University, USA

Abstract

The CLEO III experiment will collect on the order of 200 TB of data over the lifetime of the experiment. The challenges facing CLEO III are how to store such a large dataset. We will describe our experiences with Objectivity/DB on top of a Hierarchical Storage Manager.

Keywords: data storage, Objectivity, OODBMS, storage, data, database

1 Introduction

Objectivity/DB, a distributed object database management system (OODBMS), has been adopted as the storage solution by ongoing experiments (BaBar [1]) and proposed for future experiments (LHC [2]). CLEO III has adopted Objectivity as its main storage format on top of a Hierarchical Storage Manager (HSM).

To set the scale, the CLEO III experiment will collect about 20 TB in the first year, and then quickly ramp up to collect on the order of 200 TB over 5 years. Each event is roughly 40 kB at a data-taking rate of about 100 Hz, yielding a data flow rate of 4 MB/s. We expect the start of the first physics run in March 2000. A challenge facing CLEO III is how to store such a large dataset.

We will discuss our experiences with Objectivity and the various design decisions we made to limit the complexity of our storage system, given the resources a (relatively) small experiment has, shortage of development personnel, uncertainty of the future of commercial databases, and the use of commercial software in a long-term mission-critical environment.

2 Use of Commercial Software in CLEO

A number of ongoing and proposed HEP experiments have adopted Objectivity as their primary storage format to store Terabytes and Petabytes of data. Objectivity is a commercial product by Objectivity/DB, Inc., a privately held company in California. Historically HEP experiments developed their own data storage formats, usually tightly bound to their data analysis applications. This is the beginning of a new era, where experiments have adopted commercial database software (and other tools, e.g. HSMs) for their storage solutions.

The dangers of such an approach are many: The software is only available as binaries without source code, which ties it to particular versions of operating systems and compilers. The software is property of a commercial company which may have a different lifetime than the experiments using the software. The company may go out of business; it may be bought up by another company which discontinues the product. Since the source code is not available, one has to program the database based on released information (public interfaces in header files, printed manuals, product support of the company).

In the light of the dangers mentioned above, CLEO III has decided not to store its raw data coming out of the online system directly in Objectivity, rather in its own binary format which is dumped to tapes in a tape robot. Only as a second step is the database populated from those tapes. We also designed the CLEO III Data Access system to be storage format independent. This means that we can switch to a different database product or our own proprietary formats, but once a substantial amount of data has been stored in the database, switching would be difficult, if not impossible.

3 Database Design

We describe here our experiences with Objectivity and the various design decisions we made to limit the complexity of our storage system.

To clarify Objectivity terminology: a federated database is a federation of databases (which translate to physical files) on a network, where each database houses containers, which in turn contain objects; objects can reference (or link to) other objects.

3.1 Datastorage as part of the CLEO III Data Access System

The CLEO III data access system, described fully in another submission to this conference [3], is designed to be input/output data format independent. All data are accessed through a memory-based "data-bus" consisting of "Records". Different records are synchronized with respect to each other to provide a consistent view of all of the CLEO detector for a particular instant in time. The data in these Records can be served by any number of "Sources" (input) and "Sinks" (output). Any storage format plugs into this system via a concrete implementation of a Source and/or a Sink a la a device driver.

This approach requires the clear separation between transient and persistent objects. The user's analysis or reconstruction code is written in terms of the transient objects and does not have to change when he wants to process data from the database vs from some other storage format. We do not know of any benefits of using persistent objects directly in code, except that one might be worried about the added overhead of translating persistent into transient objects and vice versa. In tests we have never found this overhead to be significant, perhaps because we disallow direct links between different types of data, as we will discuss later, and because data is served on demand.

The main data access application, called "Suez", is a skeleton program, which provides runtime job setup and control, and allows dynamic loading and/or static linking of modules. The database application is implemented as an "Objectivity Source and Sink" module. In this approach the database is just one of many device drivers that can be used in Suez.

3.2 Database Layout

The natural unit of data in CLEO III is a Record. Each record contains different types of data (e.g. the Event Record contains Tracks and Showers). Sets of Records, called streams, are naturally grouped in data-taking "Runs" (i.e. they share the same run conditions, hence the same run number.)

We translate these transient concepts into database concepts. The Records become Record "Headers", which merely contain links to the various types of data. A stream of records for a given run are clustered in the same "RunContainer" in the database. Each Header stores the links to the various types of data, which themselves are stored in their own database containers, clustered by type. (E.g. an Event Header would store links to the Tracks and Showers, which themselves are clustered in the TrackContainer and ShowerContainer, respectively, grouped by Run. This is

illustrated in Figure 1). We intend to cluster different types by access patterns (although for now will rely mainly on historical CLEO access patterns.)

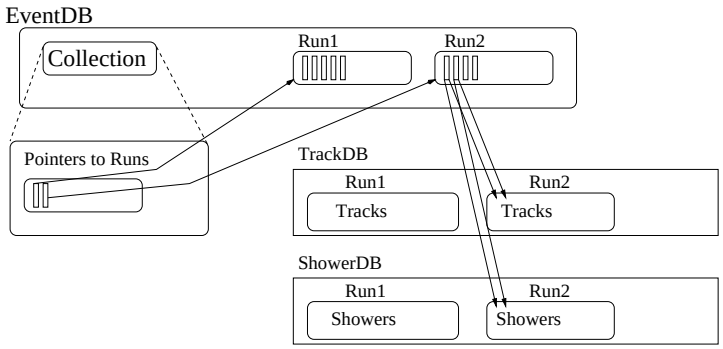

Figure 1: Clustering by Runs

The CLEO III Online software trigger and the Offline reconstruction categorize events by an event classification scheme (hadronic, bhabha, muon, tau, etc.). To allow for fast and efficient selection of events of a certain class, the database layout clusters headers in databases by their event class, as shown in Figure 2. To further ease fast selection of relevant events, the headers link

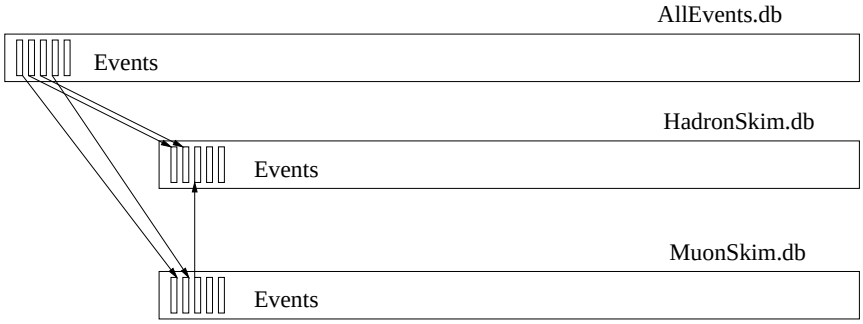

Figure 2: Clustering by Event Class

to a tag object containing fast-selection information (e.g. event tag would contain: the number of tracks, the visible energy in the event etc.).

3.3 Schema Management, StorageHelpers, and Compression

The database keeps track of the types of the stored objects; this type information in the database is called the "Schema", which is central for the entire federation. When a stored type has to be changed, the schema has to be "evolved". A corrupt schema can render the database unreadable. Evolving a type requires all objects of that type to be converted to the new type in the database. Since the schema is potentially fragile, we cannot allow new user data types to be stored in the database. And storing objects as their real types prevents compression at the object level; compression could then only be done at the file/database level.

These issues led us to a radically different approach: we store all data types as binary blobs. Only the transient data access layer knows how to interpret these binary blobs, although we do store the compression information in the database. The translation from transient to persistent object and vice versa is handled with the help of "StorageHelpers", described in another submission

to this conference [4].

Of course, this made direct links and references between data objects impossible. But it had never been our intention to store links, because our data access system has to support links in sequential access formats (files), and even links between objects in sources of different formats; for that purpose we have developed an index-based linking approach [5].

3.4 Data Organization

Objectivity has a fixed limit on the amount data that can be stored in a federation. We do not intend to store all data from day one to the end in the same federation, rather we will most likely divide our data into separate data sets which physically translate to separate federations (e.g. dataset 1 which holds runs 1-10000, dataset 2 which holds runs 10001-20000 etc.). Data taking run ranges between online maintenance shutdowns provide a natural division.

To allow processing of data in separate federations in the same process, we have to force the schema to be the same for all federations. Since our schema is already rather simple (data types are stored as one binary blob type), this should not pose a problem.

3.5 User Data

The issue of how to store user data has not been sufficiently addressed. While we have certain ideas of how to handle user data, we have not implemented any of them. Since we store binary blobs, no schema management is needed for supporting user data. Database id ranges would have to be set aside for ease of creating a user database in the official database, "skimming" or "collapsing" the interesting data into that user's database, unattaching the user's database, and reattaching it to the user's federation for further processing.

It is not clear, if all CLEO collaborating institutions will have the resources to deploy Objectivity at home, and we may implement a file-based storage format (based on historical CLEO formats) for storing user data. Our Data Access system allows us to handle data from multiple formats in the same job.

3.6 Concurrency Issues

In the reconstruction phase, multiple processes will try to write to the same database, but only one process can be updating the "collection" object at any one time during a transaction. Since we would like to use relatively few transactions for processing data, that can lead to lock collisions. Objectivity's Multiple-Reader-One-Writer mode (MROW) is an option, but can lead to database inconsistencies if used improperly. Another solution is to preallocate containers and attach them to the collection object, and run reconstruction in a second pass.

3.7 Objectivity and Mass Storage

We use the Objectivity Advanced Multithreaded Server (AMS) to interface with the Veritas Storage Migrator, a Hierarchical Storage Manager (HSM), sitting on top of a tape robot holding AIT tapes. The AMS in Objectivity 5.2 provides hooks to interface to the underlying file system. Prior to Objectivity 5.2 we had to live with Objectivity time-outs due to the HSM latency times of larger than 25s. We plan to use the Defer-Request Protocol to deal with these time-outs, and the Redirect-Request Protocol to allow load-balancing.

3.8 Compilers and Platforms

We support Solaris 2.x and OSF1 4.x, and we plan to add Linux/Intel in the near future. We encountered problems with the persistent Objectivity STL producing name clashes with the normal transient C++ STL on the OSF1 platform. We found no satisfactory solution and abandoned persistent STL in favor of our own STL classes, implemented on top of an "`ooVArray`". With the arrival of Objectivity 5.2 we plan to make use of the java-style collection classes, which we have tested on both our current platforms.

4 Summary

We have described various design decisions to limit the complexity of the CLEO data storage system. Our CLEO III data access system is format-independent, and therefore allows any number of storage formats to be used concurrently. The Objectivity data storage format is only one of several formats we support, and hooks into the system transparently to users' code. We believe this to be a major advantage of our system.

Since we store binary blobs rather than real objects, we use Objectivity more like a data-location manager, rather than a true object store, avoid schema evolution problems, and allow storage of user data. We have stress-tested our system in mock-data challenges and have found good performance with Objectivity 5.2.

References

1 BaBar's home page at `http://www.slac.stanford.edu/BFROOT`.

2 CERN's RD45 home page at `http://wwwinfo.cern.ch/asd/rd45/index.html`.

3 C. D. Jones, M. Lohner, "CLEO's User Centric Data Access System", CHEP 2000, Padova, Italy, Spring 2000.

4 C. D. Jones, D. Riley, M. Lohner "StorageHelpers: A Format Independent Storage Specification", CHEP 2000, Padova, Italy, Spring 2000.

5 J. Thaler, "How to use Lattice", CLEO-internal publication.