DATA STORAGE AND ACCESS IN LHC++

Marcin Nowak
CERN IT Division, RD45 project - CH 1211 Geneva 23 Switzerland

Abstract

This paper presents LHC data requirements as well as some features of HEP data models and explains how an ODBMS can be used to address them. Essential features of object databases will be discussed, followed by those specific to Objectivity/DB, which is the database currently used in LHC++. The differences between transient and persistent data models will be given with some rules for how to convert the former into the latter. Next, the paper will focus on HepODBMS layer, which is a set of HEP specific classes extending the functionality of a database and forming an interface used by other LHC++ software components. The concept of event collections and object naming will be discussed.

1. INTRODUCTION

Experiments at the Large Hadron Collider (LHC) at CERN will generate huge quantities of data: roughly 5 petabytes (10^{15} bytes) per year and about 100 PB over the whole data-taking period (15+ years). Data will be collected at rates exceeding 1GB/s and later analyzed, perhaps many times. The analysis frameworks of the new experiments will be developed using object-oriented (OO) technologies and consequently their data will be represented in object-oriented data models, often of significant complexity.

These factors form a challenging data storage and management problem and it seems clear that the traditional solutions based on sequential Fortran files would not be adequate. In 1995 the RD45 project was initiated at CERN to investigate new solutions and technologies. The emphasis was put on commercial products, with the hope of minimizing development costs and maintenance effort over the very long period of use. The evaluation of different technologies such as language extensions for persistency, light-weight object managers, object request brokers and object databases led to the recommendation of an Object Database Management System as the main data management product, together with a Mass Storage System to provide physical storage.

Studies of the various ODBMS products on the market, particularly with respect to their ability to satisfy LHC data management requirements, resulted in the selection of a particular database: currently Objectivity/DB.

Experiment	Data Rate	Data Volume
ALICE	1.5 GB/sec	1 PB/year (during one month)
CMS	100 MB/sec	1 PB/year
ATLAS	100 MB/sec	1 PB/year
LHC-B	20 MB/sec	200 TB/year

Table 1 Expected Data Rates and Volumes at LHC

2. OBJECT DATABASES

2.1 Data Model

In OO programming style the data is represented as a set of objects interconnected with each other in various ways, depending on the object model. Figure 1 shows a simple example of the data model for a HEP Event.

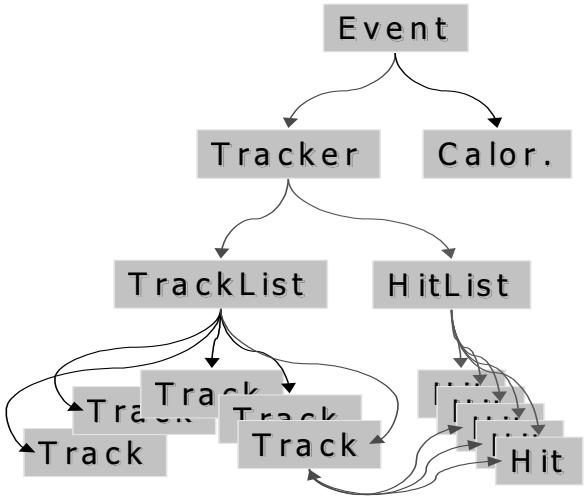

Figure 1: Simple Logical Model of Event Data Objects

An application working with a given data model would traverse the connections between objects to find the data it needs or to modify their contents. It may also modify the network of objects by adding or removing some of them. In the Event model example, the application could navigate from the main Event object to its tracking detector, retrieve a particular track from the track list, then navigate to all associated hit objects to perform a track refit.

2.2 Transient Objects

In the traditional run-cycle, an application would first create in memory the objects it needs and then fill them with some data. Next, it would perform the actual task it was designed for: working with its data representation. Finally, the program would store the results and delete objects from memory. In this scenario, the lifetime of an object is rather short and always bound to the application execution time – the objects exist only within the context of a single program. In the database terminology such objects are called *transient* (i.e. temporary).

OO languages support transient objects and navigation between them (the latter via pointers and references in C++). Creating and traversing in-memory networks of objects is very efficient and type-safe even for polymorphic classes. This, however, assumes that the entire network of objects is maintained in the memory. There is little support from today's languages regarding I/O operations on such networks of objects.

2.2.1 Object Input / Output

Providing I/O for complex data models is a difficult task for the programmer. In the first place, 2 different data formats have to be maintained for every class that is to be stored, namely:

- Class definition used by the application, including pointers to other objects
- Data encoding format used when storing in a file

The formats must be assumed to be different, as the run-time format is tightly coupled to the operating system and even compiler version. Thus, even if we start with an exact memory copy in a file, the possibility of handling different run-time formats must be provided: the application code has

to perform conversions between the two representations. The problem increases further since any individual class definition is likely to change over the long run period of LHC, leaving some objects in the old format stored on tape.

The programmer has also to decide:

- How to perform the conversion.
  The conversion of object attributes may require byte-swapping or alignment adjustment, which is a time-consuming, but rather straightforward operation. What is more difficult is storing the connections between objects, which constitute the shape of the object network. This requires translating pointers and references into a storable format and a special code that will be able to rebuild the network later.
- When to perform the I/O.
  All data transfers have to be initiated explicitly. Typically, some amount of data has to be read from disk when the application starts and all useful results have to be stored at the end. During the execution time, additional I/O operations may be required when the program follows a link referencing an object that is not yet in memory. In a multi-user environment, part of the data in memory may become stale as a result of an update performed – by another user or process – upon the corresponding object on disk. Such situations must be detected to avoid data corruption.
- How much data to transfer.
  In a complex HEP application it is difficult to predict which data items will be used. In many cases all of the event data is read, just in case it is needed. This approach may result in degraded performance.

Code that deals with object I/O often constitutes a large part of the application. Maintaining this code is both tedious and error-prone. Consistency between the disk and memory representation is not performed automatically and errors in this layer may lead to obscure software problems. In addition, large amounts of I/O related code in a class makes programs less understandable and may obscure the actual services provided by the class.

2.3 Object Persistency

Persistent objects are the opposite of transient objects. They do not disappear when the application finishes (they *persist*). This is possible because they do not belong to the application context, but rather to the context of a persistent store. In the case of an ODBMS, they belong to a database context. A persistent object will disappear only when explicitly deleted from the store.

Programs working with persistent objects do not “own” them – they receive only a copy from the store. It is possible for more then one program to access the same object at the same time in a “read” mode.

Object databases maintain the consistency between objects on disk and in memory. The programmer never deals with the disk representation – but sees only the run-time definition of the class. This feature is called “tight language binding”. The ODBMS also takes care of all I/O that has to be performed to retrieve an object. All the problems discussed in section 2.2.1 are handled by the system and not by the application programmer.

Persistent objects are real objects. They support run-time type identification, (multiple) inheritance, polymorphism (virtual functions) and abstract data types. In C++ they can also be instances of templated classes.

2.4 Transactions

Object databases provide transactions in a similar way that relational databases do. The transactions are atomic, consistent, isolated and durable (so-called A.C.I.D. [2] properties) and are usually not nested. All data access is done inside a transaction – otherwise the store is not accessible. The

standard transaction types are “read” and “update”. Some systems provide additional types of transactions that e.g. allow simultaneous read and write to the same objects An example of such a transaction type is the multiple reader, one writer (MROW) transaction supported by Objectivity/DB.

As all data access occurs inside a transaction, all I/O operations are transaction bound. At the start of a transaction, only the connection to the database is established. As the application proceeds to navigate in the data network and access objects, the relevant pieces of data are retrieved. The ODBMS tries to ensure that there are no unneeded data transfers, in order to optimise performance. If the application modifies objects or creates new ones, the changes may be kept in memory or written to disk, but they are not immediately visible to other clients. Only when the transaction is committed, all modifications are flushed to disk and registered in the database.

Transactions in database systems are the main tool to ensure data consistency. If a transaction is interrupted (aborted) in the middle, the database status is not changed.

2.5 Navigational Access

As described above, the main method of finding an object in the network is by navigation. Transient objects use pointers and references as links. A pointer is a memory address and uniquely identifies an object in the application context (or virtual memory address space). Persistent objects, which exist in the database context, need a different kind of identification.

When a new persistent object is created, the ODBMS assigns to it a unique Object Identifier (OID). The actual implementation of the OID varies between different systems, but they have common functionality – they allow the object to be found in the disk store. OIDs that point directly to the object are called physical and OIDs that use indirection are called logical. Logical OIDs give more flexibility at the cost of performance and scalability.

Object Identifiers replace pointers and references in persistent objects. They are used to create uni-directional (pointing in one direction, like a C++ pointer) associations between them. In most products they also enhance the idea of pointers by allowing:

- bi-directional associations
  bi-directional association is a relation between 2 objects. From an implementation point of view it may look just like 2 objects pointing to each other, but the ODBMS makes sure that pointers on both sides are set correctly (or reset) at the same time. It is not possible to modify only one of them, thus ensuring consistency.
- 1-to-n associations
  1-to-n association is a relation between one object and an arbitrary number of objects on the other side. It may be uni- or bi-directional.

The OID is typically hidden from the programmer by wrapping it in a *smart pointer* implementation. Smart pointers are small objects that behave semantically in the same way as normal pointers, but they also provide additional functionality. If the smart pointer provided by ODBMS is dereferenced (in C++ by using “*” or “->” operator on it) the system is able to check if the object pointed to is already in memory, and if not, read it from disk using the OID contained in the smart pointer. After that, the smart pointer behaves just like a normal pointer. All this happens without any additional code in the application.

The ODMG standard [1] defines ODBMS smart pointer as a templated class d_Ref<T>. Figure 2 shows an example program using d_Ref<> in the same way as normal C++ pointer.

```
Collection<Event> events;           // an event collection
Collection<Event>::iterator evt;    // a collection iterator

// loop over all events in the input collection
for(evt = events.begin();  evt != events.end(); evt++)
      {
      // access the first track in the tracklist
      d_Ref<Track> aTrack;
      aTrack  = evt->tracker->tracks[0];

      // print the charge of all its hits
      for (int i = 0; i < aTrack->hits.size(); i++)
            cout  << aTrack->hits[i]->charge
                << endl;
      }
```

Figure 2: Navigation using a C++ program

As a consequence of the tight binding of ODBMS to the programming language the application programmer perceives the database store as a natural extension to application memory space. Using the database one can create networks of objects much larger than would be possible in memory, with indefinite lifetime and the possibility to efficiently navigate among them.

2.6 Database Schema

If the ODBMS is to be able to perform automatic conversion between object representation on disk and in memory, it has to have detailed information about the object. It has to know the type, name and position of every attribute in the object. This information needs to be registered in the database before any object of a given class can be stored. All class definitions known to the ODBMS are called the *database schema*.

The schema registration process depends on the ODBMS and on the programming language. In Objectivity/DB a C++ class is entered into the schema by a program that pre-processes the header files. The headers may contain normal C++ classes, with the exception that object associations should replace pointers.

2.7 Concurrent Access to Data

ODBMS products provide support for multiple clients working on the same data store and concurrently updating it. Usually ODBMSs introduce a central “lockserver” that co-ordinates the updates by keeping a lock table for the whole system. To ensure data consistency in the system, all data changes are part of a transaction. If a transaction accesses part of the database, this region is locked with an appropriate lock mode (read, write or MROW). Subsequent clients trying to operate on the same region must first contact the lockserver to determine what type of access is allowed at a given time. All locks that a transaction has acquired last until the end of the transaction (either by commit or abort).

Locking and transactions are the mechanisms that allow concurrent access to a data store. Without them it would not be possible to guarantee data consistency.

3. CHOOSING ODBMS FOR LHC++

The next section describes specific features of Objectivity/DB - the ODBMS system that the RD45 project currently recommends as the data storage and management product for LHC experiments. The following list mentions requirements that were considered the most important for the selection:

- Standard compliance – ODMG [1]
  The use of a standards compliant API may make it easier to replace one ODBMS component with another system, if such need arises
- Scalability to hundreds of PB
- Mass Storage System interface
  LHC experiments will require a database able to store 100 PB of data, a large part of which will have to be kept in MSS (on tapes)
- Distributed system
- A centralised system will not be able to efficiently deal with such large amounts of data and serve many client applications accessing it concurrently
- Heterogeneous environment
  Research institutes have very diverse computing environments – a system that will be used by all of them should be interoperable between most of them
- Data replication
  Replicating the most frequently used data to remote institutes may have a big impact on performance
- Schema versioning
  The system should allow changes in the class definitions that will inevitably happen in the long run period of LHC
- Language heterogeneity
  LHC++ is written in C++, but there are graphical presentation tools implemented in Java that would profit from direct access to the database
- Object versioning
  This feature is used by various applications, such as a detector calibration database package

4. OBJECTIVITY/DB SPECIFIC FEATURES

This chapter focuses on specific features of Objectivity/DB.

4.1 Federations of Distributed Databases

The Objectivity/DB ODBMS supports a so-called *federation* of distributed databases. Each database within a federation corresponds to a filesystem file and may be located on any host on the network. There is one central federation (FDB) file containing the catalogue of all databases and the class schema. Hosts on which database files are located run the Objectivity data server (ooams). In addition, there is a central lockserver program located on a selected machine.

Client applications may use one Objectivity federated database at a time. To access the data within a federation, the database client software first reads the FDB catalogue to find where the data is located and then connects directly to the data server on a machine hosting the right database.

Before any data is read or modified, the client contacts the lockserver to obtain a lock. These operations are all performed transparently to the user, who only deals with (networks of) objects. Figure 3 shows an example of a Federated Database with 2 client applications accessing it from different hosts.

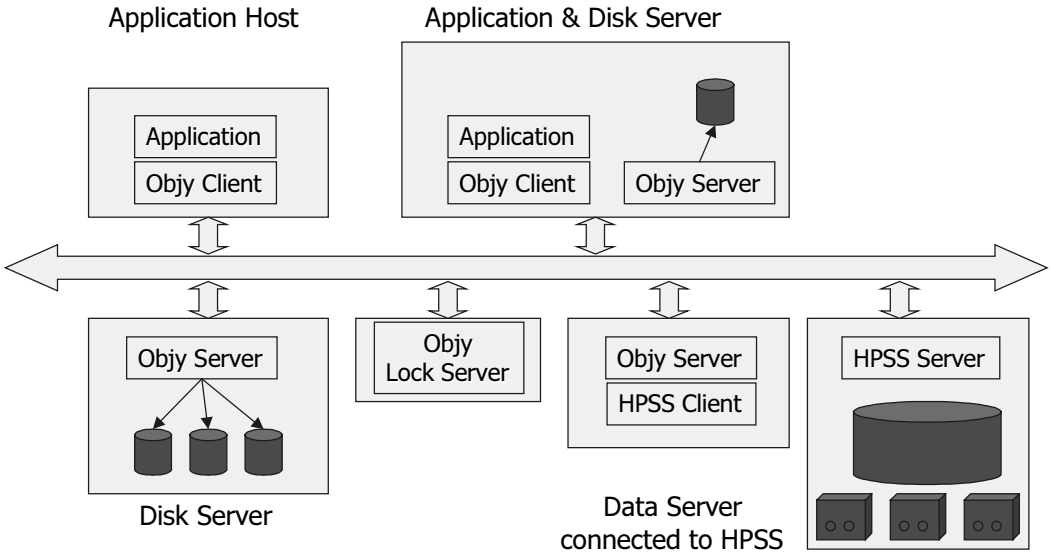

Figure 3 Distributed Applications Sharing a Federated

4.2 Physical Store Implementation

All ODBMS products use a multilevel hierarchy to implement the possibly distributed physical store. Objectivity/DB uses a hierarchy of five different levels. The topmost level - the Federated Database - keeps the catalogue of physical location of all databases that constitute the federation. Each database is structured internally into “containers” - contiguous areas of objects within a database file. Containers consist themselves of database “pages” – regions of fixed size determined at the federation creation time. Every page has “slots” for actual object data (but objects larger then a single page are allowed). Figure 4 illustrates the physical storage hierarchy in Objectivity/DB.

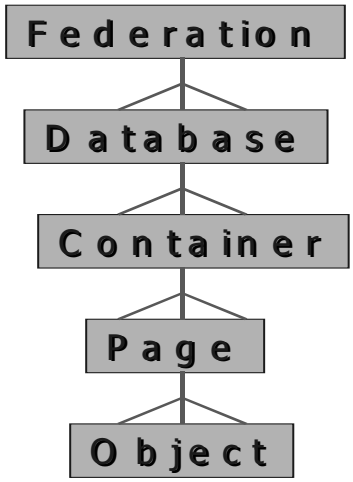

Figure 4 Storage Hierarchy in Objectivity/DB

The structure of the physical store hierarchy is directly reflected by the internal structure of the OID implementation. A 4-tuple of 16-bit numbers that represent database, container, page and slot number is used to uniquely references any object within the store.

Figure 5 Object Identifier Implementation used by Objectivity/DB

4.2.1 Separation of Logical and Physical Storage Model

The concept of OIDs allows any object to be accessed directly in the potentially large distributed store without requiring the application programmer to know the details of the store implementation, such as file and host names. Since information about the physical layout of the store is kept in a central place by the ODBMS, it is much easier to change the storage topography without compromising existing applications. One may change the location of a particular file to a new host by moving the data and changing the catalogue entry. Since the catalogue is shared by all database applications, they will use the data from the new location without any modifications.

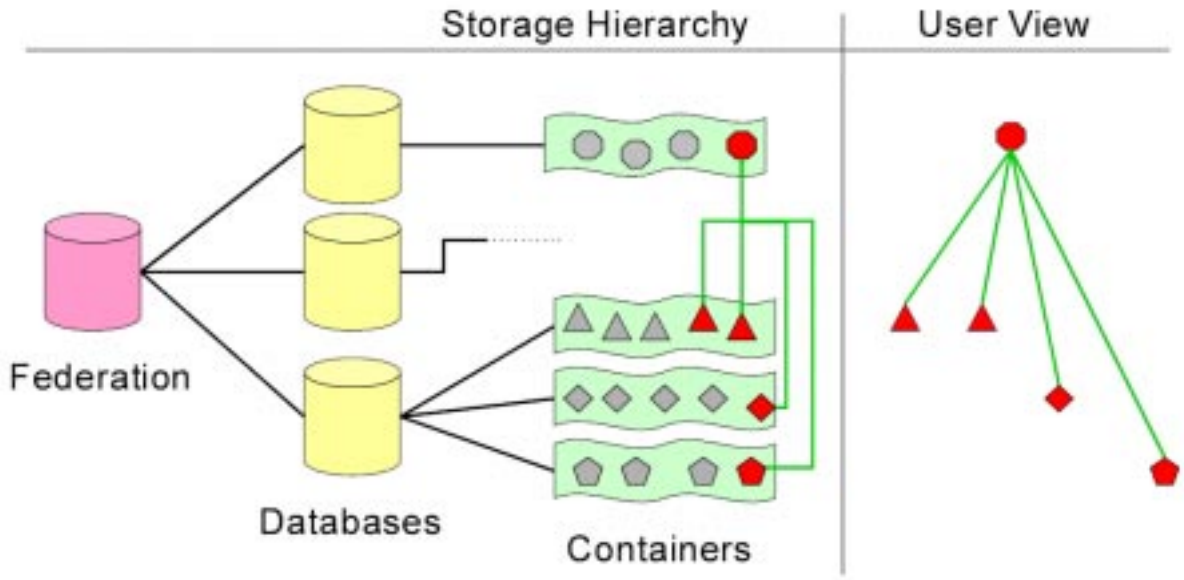

Figure 6: Physical Storage Hierarchy and Logical User View

4.2.2 Data Clustering and Re-Clustering

An important feature offered by several ODBMS products is the support for object clustering. When a persistent object is created, the programmer may supply information where the object should be placed within the physical storage hierarchy. In C++ a clustering hint may be passed as an argument to the new operator. For example, the statement

```
d_Ref<Track> aTrack = new(event)  Track;
```

instructs the database to create a new persistent track object physically close to the event object. This ability to cluster data on the physical storage medium is very important for optimising the performance of applications which access data selectively.

The goal of this clustering optimisation is to transfer only useful data from disk to the application memory (or one storage level below: from tape storage to a disk pool). Grouping data close together that will later be read together can drastically reduce the number of I/O operations needed to acquire this data from disk or tape. It is important to note that this optimisation requires some knowledge about the relative contributions of different access patterns to the data.

An simple clustering strategy is the "type based clustering" where all objects of some particular class are placed together: e.g. Track and Hit objects within an event may be placed close to each other since both classes will often be used together during the event reconstruction.

For physics analysis this simple approach is probably not very efficient since the selection of data that will be read by a particular analysis application depends more on the physics process. In this case one may group the analysis data for a particular physics process together.

4.3 Data Replication

Objectivity/DB supports the replication of all objects in a particular database to multiple physical locations. The aim of this data replication is twofold:

- To enhance performance:
  Client programs may access a local copy of the data instead of transferring data over a network.
- To enhance availability:
  Clients on sites which are temporarily disconnected from the full data store may continue to work on the subset of data for which local replicas are available.

Figure 7 shows a simple configuration where one database is replicated from site 1 to two other remote sites over a wide area network.

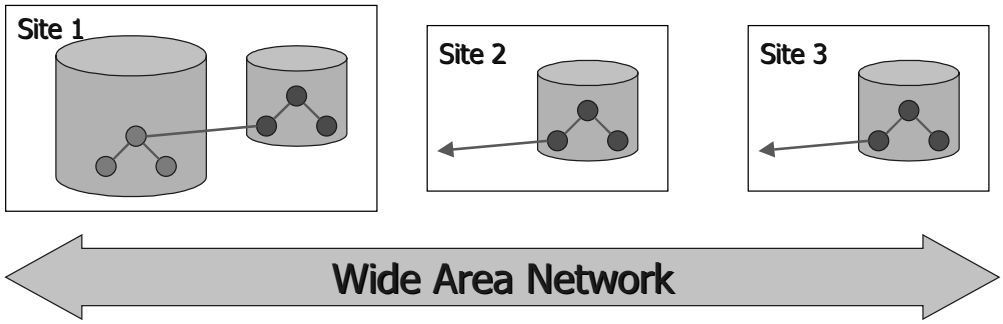

Figure 7 Database Replication

Any state changes of replicated objects on either site are transparently propagated to all other replicas by the database system. In the case that some of the replicas are not reachable, a quorum-based mechanism is used to determine which replica may be modified and a backlog of all changes is kept until other replicas become online again.

The data replication feature is expected to be very useful, for example to distribute central event selection data to multiple regional data centres.

4.4 Schema Generation

The schema generation for C++ classes in Objectivity/DB is performed using a pre-processor program (see Figure 8). The program scans class definitions of persistent classes in Objectivity's Data Definition Language (DDL) and generates C++ header and implementation files for persistent classes. The generated header files define the class interface for clients of a persistent class. The generated implementation files contain C++ code which implements smart-pointer types and various collection iterators for each persistent class. All generated files are then compiled together with any

other application code and linked against the Objectivity library to form a complete database application. The database schema is stored centrally in the federation file.

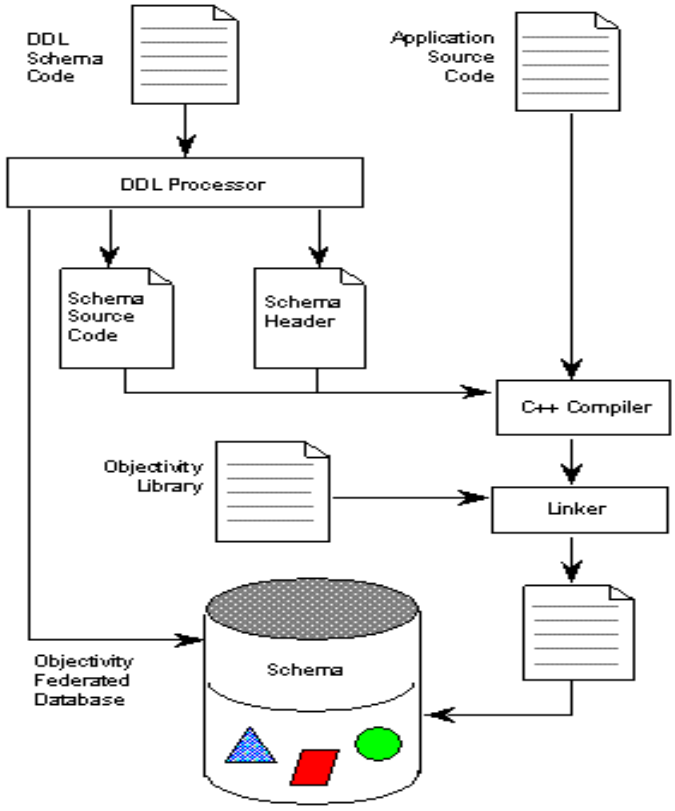

Figure 8 Schema Capture and Build Process

4.5 Creating Persistent Classes

The Data Definition Language used by Objectivity/DB is the C++ programming language extended with object associations. This makes it easy to convert transient applications and eliminates the need to learn a new programming language.

Classes become persistent by inheriting from d_Object class:

```
Class Event : public d_Object { ... };
```

Persistent classes should not contain pointers – memory pointers are meaningless in the persistent store address space. They should be replaced by references to persistent objects:

```
Event   *event_pointer;     // Event is transient
d_Ref<Event>  event_reference;   //Event is persistent
```

The “event_reference” follows the same semantic rules as the C++ pointer “event_pointer”.

The notion of pointers is further enhanced with 1-to-n and bi-directional associations. Below, the “events” attribute is a set of object references:

```
d_Ref<Event>  events[];
```

Bi-directional association is a two-directional link between objects. It has to be declared in both classes, but modification to it is an atomic operation that changes the values on both ends at the same time:

```
d_Ref<Event>  event <-> tracker;     // in Tracker
```

```
d_Ref<Tracker>  tracker <-> event;    // in Event
```

4.5.1 Persistent STL

There are some standard C++ classes that contain and use pointers internally, such as all STL containers. These classes can not be used directly with a database. Objectivity provides a special version of STL that can be used in persistent objects. The names of classes are the standard ones preceded by "d_", e.g. d_vector, d_map. Here is an example declaration of a vector of Events:

```
d_vector<Event>   my_events;
```

4.6 Object Naming

The normal way of working with a network of objects is navigation. However, the navigation has to start somewhere! Objectivity/DB allows any given object to be named and later located using this name. Objects can be named in different scopes:

- on the global level of the federation
- in scope of database or a container
- in scope of any other persistent object

Using different scopes enables the creation of personal namespaces.

4.7 Object Collections

It is very common to group objects into collections. Collections can be physical, logical or a mix of the two:

- Physical grouping is achieved by placing objects into one of the physical containers or databases of the federated database. The size of the collection is then limited by the size of the physical container it is located in.
- Logical collection is a group of references to persistent objects. The references may be stored in one of the container classes, such as a vector. The size of the collection is limited by the capacity of the collection class.
- Mixed collection is a logical collection of physical containers. The size of such a collection is practically infinite.

5. HEPODBMS LAYER

HepODBMS is a software layer that is located between the ODBMS and all other LHC++ modules. Its two main functions are to provide insulation from the database API and HEP specific storage classes.

5.1 API Independence

During the lifetime of the LHC, new versions of commercial components will be released and maybe even new products will be adopted. To make transitions between them easier, the dependence on the API of a specific vendor should be minimized. This can be achieved by using standard compliant products. However, many software products use a proprietary API that makes most efficient use of their internal architecture or are simply not fully standard compliant.

In Objectivity/DB, the structure of the federated database does not exactly reflect the ODMG database - for example, there is no notion of federation or containers in the ODMB standard. Hence, the API that deals with them is non-standard. HepODBMS tries to minimize dependence on these non-standard features by providing naming indirection and providing a higher-level database session control class.

5.2 API Enhancements

5.2.1 Database Session Control

HepODBMS contains a session control class **HepDbApplication** that provides:

- Easy transaction handling
- Methods to create databases and containers and to find them later by name
- Job and transaction level diagnostics
- The ability to set options through environmental variables

Figure 9 shows an example of a simple application using the HepDbApplication class to initialize the connection to a federated database, start a transaction and create a histogram.

```
Main(){
     HepDbApplication dbApp; // create an appl. Object
     dbApp.init("MyFD");     // init FD connection dbApp.startUpdate();
     // update mode transaction
     dbApp.db("analysis");   // switch to db "analysis"

  // create a new container
     ContRef histCont = dbApp.container("histos");
  // create a histogram in this container
     HepRef(Histo1D) h = new(histCont) Histo1D(10,0,5);

     dbApp.commit();      // Commit all changes
     }
```

Figure 9 Setting up a DB session using the HepDbApplication class

5.2.2 Object Clustering

The "new" operator generated by Objectivity for each persistent class accepts an additional parameter – the so-called clustering hint described above. Any other persistent object, container or database may serve as a clustering hint. The ODBMS will attempt to place the new object as close to the hint object as possible. In case the hint is a container or a database, the new object will be created in the container or database.

HepODBMS contains clustering classes that allow clustering objects according to different algorithms. The **HepContainerHint** class is used to store objects in a series of containers or even databases, creating a logical container of unlimited size. Special iterators allow access to all of the objects later as if they were in one container.

5.2.3 Event Collections

LHC++ users will require both "normal" size and very large (10^9) event collections. HepODBMS provides the **h_seq<T>** class that presents the programmer with a single STL-like API for all types of collections. The actual implementation of the collection depends on a strategy object that can be supplied by a user. Currently implemented strategies include:

- Vector of object references
- Paged vector of references
- Single container
- Vector of container references

The **EventCollection** class is defined as below:

```
typedef h_seq<Event> EventCollection;
```

Figure 10 shows an example of how to iterate over a collection of events using an STL-like iterator.

```
EventCollection evtCol();              // Event collection
EventCollection::const_iterator it;    // STL like iterator

For( it = evtCol.begin();  it !=  evtCol.end(); ++it )
  Cout << "Event: " << (*it)->getEventNo() << endl;
```

Figure 10 Iterating over an event collection

6. CONCLUSION

HEP data stores based on Object Database Management Systems (ODBMS) provide a number of important advantages in comparison with traditional systems. The database approach provides the user with in a coherent logical view of complex HEP object models and allows a tight integration with multiple of OO languages such as C++ and Java.

The clear separation of logical and physical data model introduced by object databases allows for transparent support of physical clustering and re-clustering of data which is expected to be an important tool to optimise the overall system performance.

The ODBMS implementation of Objectivity/DB shows scaling up to multi-PB distributed data stores and provides integration with Mass Storage Systems. Already today a significant number of HEP experiments in or close to production have adopted an ODBMS based approach.

REFERENCES

[1] The Object Database Standard, ODMG 2.0, Edited by R.G.G.Cattell, ISBN 1-55860-463-4, Morgan Kaufmann, 1997

[2] C++ Object Databases, Programming with the ODMG Standard, David Jordan, Addison Wesley, ISBN 0-201-63488-0, 1997

BIBLIOGRAPHY

Object Data Management: Object Oriented and Extended Relational Database Systems R.G.G.Catell, Revised Edition, Addison-Wesley, ISBN 0-201-54748-1, 1994

The Object-Oriented Database System Manifesto M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. In Proceedings of the First International Conference on Deductive and Object-Oriented Databases, pages 223-40, Kyoto, Japan, December 1989.

Object Databases As Data Stores for High Energy Physics, Proceedings of the CERN School of Computing, 1998, CERN 98-08

Object Oriented Databases in Hi High Energy Physics, Proceedings of the CERN School of Computing, 1997, CERN 97-08

The RD45 collaboration reports and papers,
http://wwwinfo.cern.ch/asd/cernlib/rd45/reports.htm

RD45 - A Persistent Object Manager for HEP, LCB Status Report, March 1998, CERN/LHCC 98-x

Using an Object Database and Mass Storage System for Physics Production, March 1998, CERN/LHCC 98-x

RD45 - A Persistent Object Manager for HEP, LCB Status Report, March 1997, CERN/LHCC 97-6

Object Database Features and HEP Data Management, the RD45 collaboration

Using and Object Database and Mass Storage System for Physics Analysis, the RD45 collaboration

GLOSSARY:

ACID Transactions – Atomic, Consistent, Isolated, Durable
MB – Megabyte, 1 000 000 bytes
GB – Gigabyte, 1000 MB
PB – Petabyte, 1 000 000 GB
HEP – High Energy Physics
MSS – Mass Storage System
ODBMS – Object Database Management System
ODMG – Object Database Management Group (standards committee)
LHC - Large Hadron Collider
LHC++ - project aiming to replace the current CERNLIB libraries with a suite of OO software
DDL – Data Definition Language used by Objectivity/DB
OID – Object Identifier
MROW - multiple reader, one writer transaction where the old contents of a database region that is being modified by a writer is still accessible to other database clients in read-only mode