Exercise :

Data Storage and Access in LHC++

Objectives:

1. Converting transient data model into persistent one
2. Using HepDbApplication class from HepODBMS
3. Populating a Federation with persistent events
4. Browsing the contents of the database

Getting started

- Login on your computer account
- Execute the setup script: (advanced users running bash shell should source anaphe.sh)

```
> source anaphe.csh [Enter]
```

- Change directory to the working directory for this exercise:

```
> cd day1/populateDB [Enter]
```

- Check the contents of the directory

```
populateDB> ls
```

You should find following files:

- populateDB.cpp - the main program
- Event.ddl - definitions of Event classes
- GNUmakefile - the makefile for GNU make
- randomSource.h - random data generator to create
- randomSource.cpp fake Events
- setfd.sh - bash script to define the environment for this exercise
- setfd.csh - tcsh script to define the environment for this exercise

You have to “source” the setfd script (choose the right version depending on your shell type).
For bash shell users :

```
populateDB> source setfd.sh  [Enter]
```

For tcsh shell users:

```
populateDB> source setfd.csh  [Enter]
```

The data model

The data model for this exercise consist of 5 classes:

- Event - the main entry point into the Event
- Tracker - a list of Tracks of some imaginary tracking detector
- Calo - a list of Clusters form some calorimeter
- Track - a simple track
- Cluster - a simple cluster

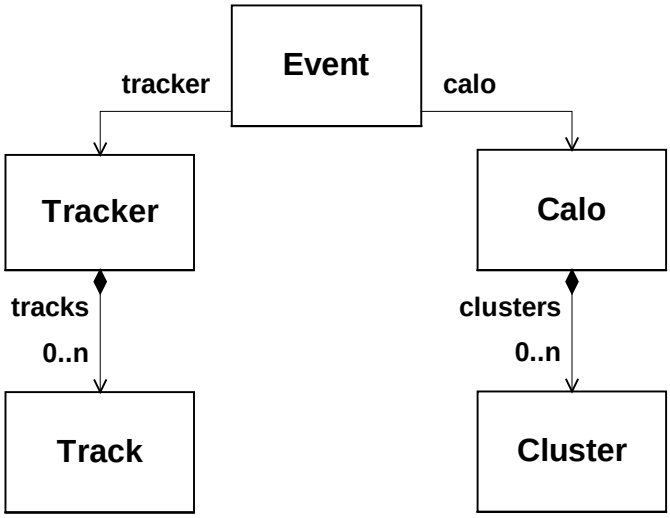

Figure 1 The Event data model

Both Track and Cluster classes are simple structures containing only basic numerical data types. Calo and Tracker are container classes that store an arbitrary number of Clusters and Tracks in an STL vector. Together they contain all data that belongs to a single event. The Event class itself does not contain any real data – it only keeps references to one Calo and one Tracker object and binds them into one logical entity.

All classes are declared in the Event.ddl file.

Converting transient model into persistent model

The first objective of this exercise is to convert the transient Event model into a persistent Event model.

1. Requirements for persistent classes

Read the requirements, but do not modify the source files yet. You will do it in the next section.

- Classes that are directly persistent (as opposed to persistence by embedding in other persistent classes) have to inherit from d_Object class.

  Example: **class** MyPersistentClass : **public** d_Object **{** ... **};**

  In the Event model the following classes are directly persistent:
  - Event
  - Tracker
  - Calo

  Track and Cluster type objects will become persistent just by embedding them in Tracker and Calo. They do not require any modifications.

- Persistent classes may not contain STL containers. All STL containers have to be replaced by their persistent equivalents. In this exercise there is one STL class used in the transient version of the program – the vector class – that may be replaced by a persistent d_Varray class:

  Example: vector<Track> -> d_Varray<Track>

  In the Event model 2 classes contain vectors of other classes:
  - Tracker
  - Calo

- Persistent classes should not contain C++ pointers. Replace all pointers with database object references (d_Ref);

  Example: Tracker* -> d_Ref<Tracker>

  The only class that contains C++ pointers is
  - Event

- When creating new persistent objects the database system expects that you will provide an argument to the new() operator telling it where to place the new object (e.g. in which container). HepODBMS provides a HepContainerHint type that may be used to cluster objects of the same type together. It is usually done by defining a static member of HepContainerHint type in a persistent class and using it as an argument to the new() operator.

  Example: (inside a persistent class) static HepContainerHint clustering;

- All directly persistent classes need such a member:

- Event
- Tracker
- Calo

2. Modifying the class definitions

The **Event.ddl** file contains the definitions of the 5 classes. As was mentioned, Track and Cluster classes do not require modifications. The third class – Tracker - is already adapted to the persistent model -–it may serve as an example of how to modify the Calo class, as they are very similar.

Two classes that you have to change are Calo and Event.

Using your favorite editor open the **Event.ddl** file and make the changes.
All the places that require modifications are marked by #error directives. When making changes, remove the corresponding #error lines. If you fail to do all required modifications (or forget to delete the error directive), the compiler will print an error message with the line number where the modification should have been done.

You should change only the names of types. **Do not change the names of the attributes** (data members) themselves, as they are used in the program (.cpp) files.

Here is the list of necessary actions:

- Calo
  - Take a look at the Tracker class – it is almost identical to the Calo
  - Add public inheritance from d_Object to Calo
  - Change vector<Cluster> into d_Varray<Cluster>
  - Uncomment the “clustering” member
- Event
  - Add public inheritance from d_Object to Event
  - Change the C++ pointer type Tracker* into d_Ref<Tracker>
  - Change the C++ pointer type Calo* into d_Ref<Calo>
  - Uncomment the “clustering” member

3. Generating database schema

When you finish all modifications, save the file and try to enter your schema into the database. Use the following command to do it:

```
populateDB> make new_schema  [Enter]
```

You should see following output:

```
creating schema in populateDB
Schema import from HISTO to EXAMPLE_FD

Updating Name Service values...
Now updating System Name Space (catalog) values...
Now updating Database File locations...

Federated Database Installation complete.
```

```
generating schema for Event
```

Watch for any errors. If you see any, go back and try to correct the Event.ddl.
If you get a message that "compare failed for class X ", run

```
populateDB> make clean [Enter]
```

before trying again to make "new_schema".

If you are stuck, take a look at the Event.ddl file in the **populateDB.solution** directory. It contains the complete, working example program.

If you successfully pass the schema generation step, two things will happen:

- Interface (Event.h, Event_ref.h) and implementation (Event_ddl.cpp) files will be generated:

```
populateDB> ls [Enter]
```

  - Event.h - C++ header generated by DDL preprocessor
  - Event_ddl.cpp - implementation file generated by DDL preprocessor
  - Event_ref.h - additional header generated by DDL preprocessor

- The schema will be checked for consistency and stored in the Federated Database.

Check if these files are in your working directory. If you see them, then you may proceed to the next part of the exercise.

Using HepDbApplication class from HepODBMS

HepDbApplication class provides certain functions that make using a database easier and hide unnecessary details from the user. Some of the interesting methods of this class are:

- init() - begin using the Federated Database
- run() - execute user-supplied main function
- startUpdate() - start a transaction in "write" mode
- db("*db_name*") - create a database called "db_name"
- container("*c_name*") - create a container called "c_name"
- commit() - commit a transaction

The common use of HepDbApplication is to create a class that inherits from it and supply the run() function that will do the actual job.
Open the **populateDB.cpp** file with an editor and find the **populateApp** class. It has the run() function that will populate the database with 1000 random events. The function first starts a transaction in "write" mode calling startUpdate(). Next, it creates 3 databases called "Events", "Tracks" and "Calo" and a container with the same name in each of them. The containers are then used as clustering directives for Event, Track and Calo classes. Then the function enters a loop and creates Event objects filling them with

random data. At the end it calls commit() to register all the changes in the database. Without commit(), all the created events would be discarded.

Actually, the run() function is not doing all of this yet. Two things are missing and your task is to complete the implementation.

1. Creating the "Calo" database and container
The run() function creates the "Events" and "Tracks" databases, but not the "Calo" database nor container. Add the missing calls to db() and container(). It is important to place the call to container() after the call to db(), because the new container will be created in the most recently accessed database.
The value returned by the container() method should be used as the clustering hint for the Calo class.

2. Creating persistent events
The loop in the run() function does not create events. It requires the call to the **new()** operator in the place marked by the comments. The new() operator for persistent objects has the following syntax:

HepRef(*class_name*) *object_reference*; // this is the declaration of *object_reference*
object_reference = **new**(*clustering_hint*) *class_name*(*constructor_arguments*);

(The HepRef is the same as d_Ref, but has to be used with the new() operator due to Objectivity/DB specifics)
For the Event class, you will have to replace:

object_reference	= evt
class_name	= Event
clustering_hint	= Event::clustering() *(a function!)*
constuctor_arguments	= i *(the event loop counter)*

3. Compiling and linking the program
To compile and link the whole program run make:

```
populateDB> make [Enter]
```

At this stage, the populateDB.cpp file and the files produced by schema preprocessor will be compiled and linked. The executable file name is **populateDB** – check if it has been created in the working directory.

Populating the Federation with Events

If you have successfully created the populateDB.exe program, try to execute it:

```
populateDB> ./populateDB [Enter]
```

You should see following output:

```
*** starting execution of ./populateDb
*** about to initialise the database session
*** creating 1000 Events ... done.
*** generated 19948 tracks and 19948 clusters in total.
```

The program should create 1000 Event objects (with associated Tracker and Calo objects) and store them in the federation. We will use the **ootoolmgr** tool to browse the database and look at the Event objects.

Browsing the database

The "populateDB" program creates four databases in the example Federation: system database named "**System"** and user databases named "**Events**", "**Tracker**" and "**Calo**". Objects of type "Event" are stored in the "Events" database and in the "Events" container. Tracks are stored in the "Tracker" database in the container "Tracks" and Clusters in the "Calo" database in the container "Clusters. The structure of the Federation is illustrated by Figure 2.

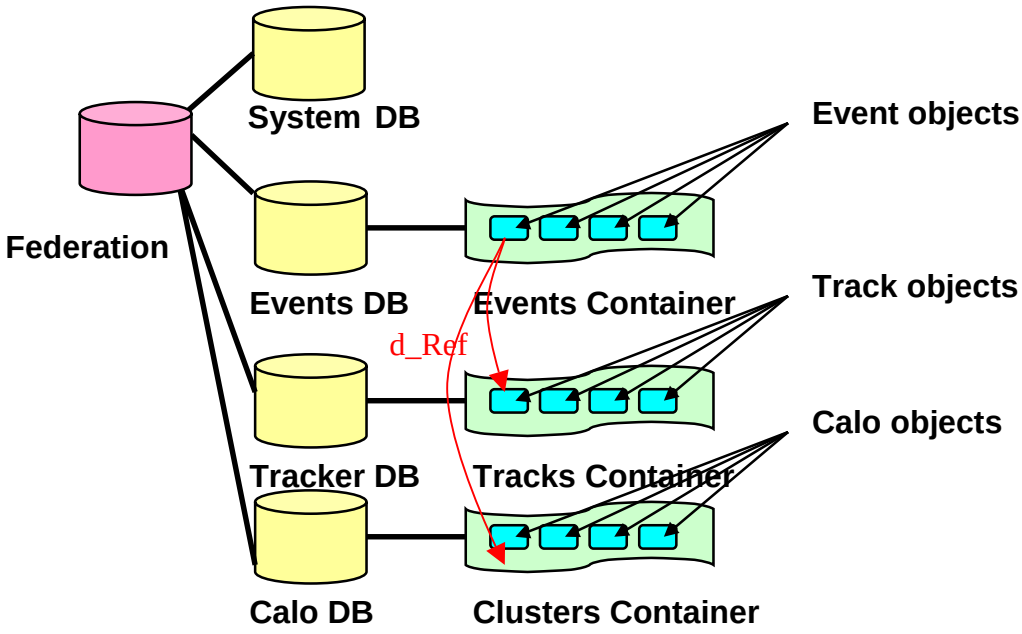

Figure 2: populateDB Federated Databse

To browse the contents of the database, start the Objectivity/DB 'oobrowse' tool:

```
populateDB> ootoolmgr  [Enter]
```

A browser window should appear. From the “File” menu select the “default” option, and from the “Tools” menu select “Browse FD”. A window as in Figure 3 should appear.

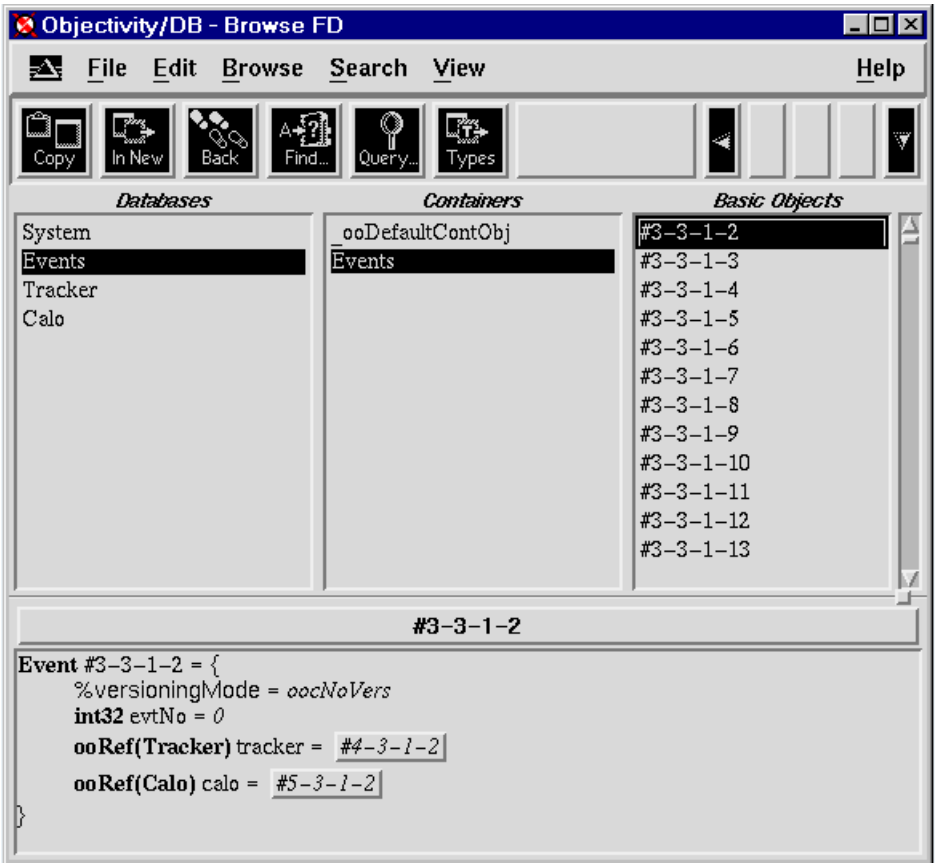

Figure 3 Objectivity/DB browser window

The 3 fields in the upper part of the window show the structure of the federation. The leftmost field lists all databases in the federation, the middle filed lists containers in the currently selected database and the rightmost field lists all objects in the currently selected container.
In the beginning you should see in the first field all the databases that were created by the “populateDB” program (plus an additional “System” database).
Now:

- Click on the “Events” database. The “Events” container (and a default container called “_ooDefaultContObj”) will appear in the container list.
- Click on the “Events” container. A list of object IDs (OIDs) in the form of 4-tuples will be displayed in the right-most field. They are the identifiers (OIDs) of Event objects that the program stored in the database.
- Select one of the OIDs. The object contents will be displayed in the large field in the lower part of the window.
- You will see the event number and references to the Tracker and Calo components of the Event.

Click on one of the references to navigate to the associated object that is stored in different databases. You may notice that the current database and container change if you follow the reference.

Finishing the exercise

Congratulations! You have finished this exercise.
Close the Objectivity/DB browser if it is still open and run:

```
populateDB> make clean
```

to remove the database and all other intermediate files from the working directory.