Table of Contents

Tutorial: creating conditions objects

Complete: ▨▨□
Detailed Review status

Goals of this page:

This page explains how create new objects to be stored in the conditions database:

- creating the corresponding C++ classes;
- filling in the database;
- retrieving the data.

The tutorial is aimed at software developers of conditions-related data.

Introduction

CMSSW provides an interface to database implementations for the storage of conditions data. The relevant packages are located in the various `Cond...` subsystems of CMSSW. This interface allows to store objects as C++ class in various types of database.

In this tutorial, we define a simple data structure (C++ class) `MyPedestals` and all the components the interface needs. We then store data in one of the supported DB schemas and retrieve it back.

Goal

Your **Goal** is to store objects of a given type (in this example `MyPedestals`) in the condition database each associated to a given **Time Interval of Validity** and be able to retrieve the one valid for a given event during reconstruction (or analysis) using the standard `EventSetup` mechanism i.e.

```
void
MyAnalizer::analyze(const edm::Event& e, const edm::EventSetup& setup){
    edm::ESHandle<MyPedestals> pPeds;
    setup.get<MyPedestalsRcd>().get(pPeds);
    MyPedestals const * myped=pPeds.product();
```

Setup your environment

This tutorial works with CMSSW release CMSSW_7_6_5.

Create a local developer area:

```
cmsrel CMSSW_7_6_5
cd CMSSW_7_6_5/src
cmsenv
```

We also the package that includes all records (where the framework actually stores the conditions data, see below):

```
git cms-addpkg CondFormats/DataRecord
```

(do a `scram b -j 8` at this point to gain time)

How to create a new conditions DB object

We first have to create the package containing our object. This has to be put in the `CondFormats` subsystem:

```
mkdir -p CondFormats/MyPedestals/{interface,src,test}
cd CondFormats/MyPedestals
```

caveat: this tutorial is based on code actually existing in the pakage `CondFormats/Calibration`

The object class

We then define a simple container class for the data (see some restrictions here). Since we want to keep it simple, it all fits in the header file `interface/MyPedestals.h`:

```
#include <vector>
class MyPedestals {
    public:
      struct Item {
       float m_mean;
       float m_variance;
      };

    std::vector<Item>  m_pedestals;
   };
```

For a class to be stored in DB, it has to be serializable.

To make it serializable we add the COND_SERIALIZABLE macro:

```
#include "CondFormats/Serialization/interface/Serializable.h"

#include <vector>

class MyPedestals{
public:
  struct Item {
    float m_mean;
    float m_variance;
    COND_SERIALIZABLE;
  };

  std::vector<Item> m_pedestals;
  COND_SERIALIZABLE;
};
```

This of course adds "CondFormats/Serialization" as a dependency.

See other examples in CMSSW Git

We also need a `src/headers.h` file, which holds include statements to all our serializable classes. This file is read by the serialization system to find the serializable classes and generate serialization code for them.

Our `src/headers.h`

```
#include "CondFormats/MyPedestals/interface/MyPedestals.h"
```

Now, we can write a test program for the class's serialization, using built-in tests. Here's `test/test_serialization.cc`:

```
#include "CondFormats/Serialization/interface/Test.h"

#include "../src/headers.h"

int main()
{
    testSerialization<MyPedestals>();
    testSerialization<std::vector<MyPedestals>>();
    testSerialization<std::vector<MyPedestals::Item>>();
}
```

The `test/BuildFile.xml`:

```
<bin file="test_serialization.cpp">
    <use name="CondFormats/MyPedestals"/>
</bin>
```

If we want our class to be available as EventSetup data (and in this case we do), we have to have an LCG dictionary made for it, which is needed for the class to be used in ROOT.

This is done through two files to be put in the `src` of our project: `classes_def.xml` and `classes.h`.

`src/classes.h` holds the class definitions:

```
#include "CondFormats/MyPedestals/interface/MyPedestals.h"
```

In this case, the contents are the same as headers.h, but this might not be always the case.

The rules for the dictionary are the very same as for any other persistent class: for instance those forming the event. There are only two additional attributes specific to the DB: `class_version` and `mapping`.

`src/classes_def.xml` contains the names of all the classes we want to store and their constituents, as well as class template instantiations. In our case, here is how it might look like this:

```
   <lcgdict>
       <class name="MyPedestals" class_version="0"/>
       <class name="MyPedestals::Item"/>
       <class name="std::vector<MyPedestals::Item>"/>
     </lcgdict>
```

If you really want more details on the dictionary generation, have a look here

For our class to be useable as EventSetup data, it needs to be registered into the CMSSW framework as such. The framework will then handle it as any EventSetup data. This is done by convention in a corresponding file `T_EventSetup_ClassName.cc`. In our case `src/T_EventSetup_MyPedestals.cc`:

```
// T_EventSetup_MyPedestals.cc

#include "CondFormats/MyPedestals/interface/MyPedestals.h"
#include "FWCore/Utilities/interface/typelookup.h"

TYPELOOKUP_DATA_REG(MyPedestals);
```

This, along with LCG dict, adds a "FWCore/Utilities" dependency to our BuildFile.xml .

See here for more details about the EventSetup.

Now, our project needs a BuildFile.xml to tell scram how to compile it. For our component `BuildFile.xml` shall be:

```
<use name="FWCore/Utilities"/>
<use name="CondFormats/Serialization"/>

<export>
  <lib name="1"/>
</export>
```

Now we can compile this first component (you will see the dictionary and serialization generation):

```
scram b
```

More complex classes

Classes with a complex structure and objects with a large number of data items gives performance problems in storage and retrieval. In such cases we advice to use a **blob** representation of the complex or large data member. This can be easily achieved adding in the dictionary the attribute "mapping=blob". For instance for our `MyPedestals` class the declaration in the dictionary to store the `vector` as `blob` will look like this:

```
   <lcgdict>
       <class name="MyPedestals" class_version="0">
         <field name="m_pedestals" mapping="blob" />
       </class>
       <class name="MyPedestals::Item"/>
       <class name="std::vector<MyPedestals::Item>"/>
     </lcgdict>
```

Complex data structures will inevitably produce performance degradation at runtime. For persistent objects please try to optimize data structures for simple Write-Once Read-Many use cases: usually flat vector-like structures suffices. Consult the Condition core team if you think you need to implement more complex structures to represent your data.

The `EventSetup` record

As we just said, the conditions data is handled by the framework through the EventSetup. We then have to define a **Record** where the objects are actually stored and from which it can be retrieved. The record class must be defined in `CondFormats/DataRecord`. This is easily done with the `mkrecord` script provided by the framework:

```
cd $CMSSW_BASE/src/CondFormats/DataRecord/interface
mkrecord MyPedestalsRcd
mv MyPedestalsRcd.cc ../src
```

This should compile straight away:

```
cd ..
scram b
```

caveat for conditions each record can contain objects of only one super type (single-inheritence).

The `DataProxy` plugin

The last component we need will tell the DB interface which data do associate to which record. This has to be located under `CondCore`, in a package specific to each task. Here, we define it as `CondCore/MyPedestalsPlugins`:

```
cd $CMSSW_BASE/src/CondCore
mkdir -p MyPedestalsPlugin/src
```

```
cd MyPedestalsPlugin/src
```

and we create the file `plugin.cc` in there:

```
#include "CondCore/PluginSystem/interface/registration_macros.h"
#include "CondFormats/DataRecord/interface/MyPedestalsRcd.h"
#include "CondFormats/MyPedestals/interface/MyPedestals.h"

REGISTER_PLUGIN(MyPedestalsRcd,MyPedestals);
```

Note one can register several such pairs in the same plugin. See some examples here.

The corresponding `src/BuildFile.xml` is:

```
<use name="CondCore/ESSources"/>
<use name="CondFormats/CondTest"/>
<use name="CondFormats/DataRecord"/>

<flags EDM_PLUGIN="1"/>
```

We can now build our plugin system (`scram b`). After succesful compilation, we check that the proxy for the EventSetup has been correctly built:

```
$ edmPluginDump | grep MyPedestalsRcd
MyPedestalsRcd@NewProxy
```

How to fill the DB with data

In order to actually write our data into the database, we need a small module that will create the data. We then configure the generic `PoolDBOutputService` which will write them to the database. (In production this **HAS** to be done using the `CMS.PopCon` infrastructure)

The data maker

We first create a new (dummy) subsystem:

```
cd $CMSSW_BASE/src
mkdir MyCalibrations
cd MyCalibrations
```

and we use the `mkedanlzr` script to create a skeleton of our maker:

```
mkedanlzr MyPedestalsMaker
```

The resulting file `MyCalibrations/MyPedestalsMaker/src/MyPedestalsMaker.cc` needs to be edited in the `analyze` method, to create the desired data. In our example, we will not create anything: we just save a dummy object through the DB interface.

Here's how the analyze method then looks like:

```
void
MyPedestalsMaker::analyze(const edm::Event& iEvent, const edm::EventSetup& iSetup)
{

  MyPedestals* pMyPedestals = new MyPedestals();

  // Form the data here

  edm::Service<cond::service::PoolDBOutputService> poolDbService;
  if( poolDbService.isAvailable() )
      poolDbService->writeOne( pMyPedestals, poolDbService->currentTime(),
                                                 "MyPedestalsRcd"  );
  else
      throw std::runtime_error("PoolDBService required.");

}
```

A few comments (more will come later):

- The interface is called as a framework **service**. We first have to check that it is available (*i.e.* it was configured), otherwise we throw an exception.
    - Using an unavailable interface would throw an exception as well.

- Data is **tagged** in the database (see configuration below): we then check if the tag already exists.
    - If the tag does *not* exist: a new **interval of validity** (IOV) for this tag is created, valid till "end of time".
    - If the tag *already* exists: the IOV of the previous data is stopped at "current time" and we register new data valid from *now* on (`currentTime` is the time of the *current event*!).
- The data is then registered in the service for writing out (done at the end of the job).

Everything you want to know about IOV is available at SWGuideIOVAndIOVMetaDataInANutshell

This producer will compile with the following `BuildFile.xml`:

```
<use name="FWCore/Framework"/>
<use name="FWCore/PluginManager"/>
<use name="FWCore/ParameterSet"/>
<use name="CoralBase"/>
<use name="CondFormats/MyPedestals"/>
<use name="CondCore/DBOutputService"/>
<flags EDM_PLUGIN="1"/>
```

Do not forget to compile everything again. Just do it:

```
cd $CMSSW_BASE/src/MyCalibrations
```

And execute this build command:

```
scram b
```

If this build fail, try this one:

```
scram b -f
```

it will skip reading the cache.

Configuring the DB output service

We now want to run the analyzer and create an output file. We will configure the DB service to write to a SQLite file. Here's the test-maker_cfg.py file (for structure's sake, place it in `MyPedestalsMaker/test`):

```
import FWCore.ParameterSet.Config as cms

process = cms.Process("MyPedestalsMakerwrite")

# Load CondDB service
process.load("CondCore.CondDB.CondDB_cfi")
```

```
# output database (in this case local sqlite file)
process.CondDB.connect = 'sqlite_file:MyPedestals.db'

# A data source must always be defined. We don't need it, so here's a dummy one.
process.source = cms.Source("EmptyIOVSource",
    timetype = cms.string('runnumber'),
    firstValue = cms.uint64(1),
    lastValue = cms.uint64(1),
    interval = cms.uint64(1)
)

# We define the output service.
process.PoolDBOutputService = cms.Service("PoolDBOutputService",
    process.CondDB,
    timetype = cms.untracked.string('runnumber'),
    toPut = cms.VPSet(cms.PSet(
        record = cms.string('MyPedestalsRcd'),
        tag = cms.string('myPedestal_test')
    ))
)

process.pedestals_maker = cms.EDAnalyzer("MyPedestalsMaker",
    record = cms.string('MyPedestalsRcd'),
    loggingOn= cms.untracked.bool(True),
    SinceAppendMode=cms.bool(True),
    Source=cms.PSet(
        IOVRun=cms.untracked.uint32(1)
    )
)

process.path = cms.Path(process.pedestals_maker)
```

- We first include the standard settings `CondDB_cfi.py` (have a look at it in Git).
- We customize it:
    - the `connect` parameter defines the output file; `sqlite_file` is the protocol to use (other possibilities are `oracle` and `frontier`);
- We then configure the `PoolDBOutputService`:
    - We use the previous process.CondDB for the connection.
    - the `record` name is a local keyword to associate the object to a tag.
    - the `tag` string is the tag we want to attach to our data (see also above, when storing the data).

To run the configuration file `test/test-maker_cfg.py`:

```
cmsRun test/test-maker_cfg.py
```

It should create the file we asked for `MyPedestals.db`.

How to retrieve the data

Now that we have a DB file, we can also retrieve the data. This again requires the configuration of the DB interface, this time using the `ESSource`, in a way very similar to the output service.

In order to check that we can retrieve the data through the EventSetup, we will use a framework module that exactly checks that, the EventSetupRecordDataGetter.

Here's `test/test-retrieve_cfg.py`:

```
import FWCore.ParameterSet.Config as cms
```

```
process = cms.Process("MyPedestalsMakerRetrieveTest")

process.load("CondCore.CondDB.CondDB_cfi")
# input database (in this case the local sqlite file)
process.CondDB.connect = 'sqlite_file:MyPedestals.db'

process.PoolDBESSource = cms.ESSource("PoolDBESSource",
    process.CondDB,
    DumpStat=cms.untracked.bool(True),
    toGet = cms.VPSet(cms.PSet(
        record = cms.string('MyPedestalsRcd'),
        tag = cms.string("myPedestal_test")
    )),
)

process.get = cms.EDAnalyzer("EventSetupRecordDataGetter",
    toGet = cms.VPSet(cms.PSet(
        record = cms.string('MyPedestalsRcd'),
        data = cms.vstring('MyPedestals')
    )),
    verbose = cms.untracked.bool(True)
)

# A data source must always be defined. We don't need it, so here's a dummy one.
process.source = cms.Source("EmptyIOVSource",
    timetype = cms.string('runnumber'),
    firstValue = cms.uint64(1),
    lastValue = cms.uint64(1),
    interval = cms.uint64(1)
)

process.path = cms.Path(process.get)
```

Now run `test/test-retrieve_cfg.py`, and you should get:

```
$ cmsRun test/test-retrieve_cfg.py

%MSG-s DataGetter:  EventSetupRecordDataGetter:get@streamBeginRun  25-Jul-2016 18:10:02 CEST Run:
got data of type "MyPedestals" with name "" in record MyPedestalsRcd
```

Success!

How to read/write the data to the offline conditions DB

Now, what next?

1. Write a data object that does what you need.
2. Test it the same way as above with sqlite.
3. Also test it in a more realistic environment by actually using the data in your algorithm.
4. Then try to store/retrieve it through **ORACLE** on `cmsprep`. Using the following parameters in your _cfg.py

   ```
   process.CondDB.connect = "oracle://cmsprep/CMS_COND_TASKNAME"
   ```

   Note: replace "CMS_COND_TASKNAME" with the account name allocated to your task.

   ```
   process.CondDB.DBParameters.authenticationPath = 'where_it_is'
   ```

Note: parameter authenticationPath should point to the location of your authentication.xml. For example, at CERN the AFS location is "/afs/cern.ch/cms/DB/conddb"

5. Try the **Frontier** access on the same. Instruction can be found here
6. Get your data validated (for DB input/output) by the DB core-team. More on oracle production setup can be found here
    - New oracle accounts shall be requested to the DB coordinator
    - Insertion of the **validated** data in the integration/production DB shall be coordinated with the PopCon responsible
    - Use in production shall be coordinated with ALCA coordination (for the global-tag) and release management (for the code)

Various tools

Some command-line tools are presented in SWGuideCondToolCommand

Give me the source

As a closing remark, you can find the whole source (the state at which you should be now) here:

- src.tar.gz: The complete tutorial source.

Review status

Reviewer/Editor and Date (copy from screen)	Comments
VytautasMickus - 2016-07-25	Update to 7_6_5
VincenzoInnocente - 19 May 2010	Update after the release of 3_7_0
FredericRonga - 03 Apr 2007	Update after the release of 1_3_0
ZhenXie - 20 Feb 2007	Review section How to read/write the data to the offline conditions DB
FredericRonga - 16 Feb 2007	First complete version
FredericRonga - 14 Feb 2007	First draft

Responsible: VincenzoInnocente
Last reviewed by: ZhenXie - 20 Feb 2007

This topic: CMSPublic > SWGuideCondObjectsTutorial
Topic revision: r31 - 2017-04-19 - AivarasSilale