Athena Developers Tutorial, September 2019

Conditions Handling in Athena

Walter Lampl, University of Arizona

- The Conditions Database infrastructure used by ATLAS
- How to use calibration constants in reconstruction
- How to write Conditions Algorithms

What are we talking about?

- Lots of what we call *reconstruction* implies applying calibration constants
- Sometimes we need to apply corrections based on temperatures or voltages at the time of data taking
- Some of these constants change over time, some are very stable
- We have infrastructure to cover all these cases
  - And with the migration to athenaMT, this infrastructure is changing
- In the conditions/database area it is particularly difficult to depreciate stuff - and maintain backward compatibility
  - You'll find lots of stuff that we just keep around to be able to process (reproduce) old data
- **NOT covered here**: How to derive calibration data and populate the databases

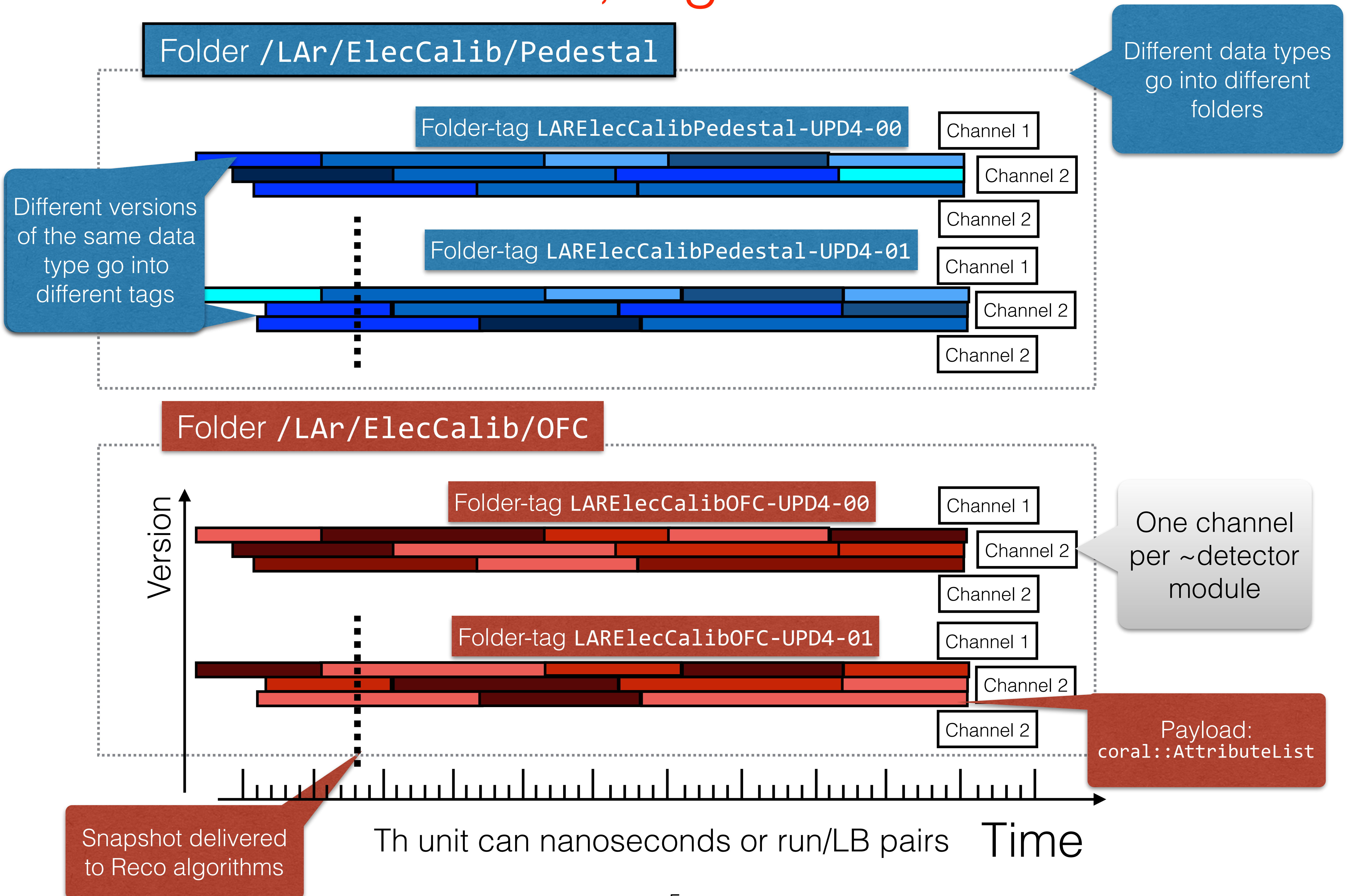
How we store conditions (1/2)

- The current work-horse: The COOL conditions database
  - A software layer (library) that hides most of the database-technicalities
  - Started as common (LHC-exp) project, by now ATLAS is the only user
  - Plan to replace by run 4 with a new product called CREST
  - Back-end: CERN-Oracle (master copy), distribution via layers of caches (“Frontier”)
    - SQLite files also possible (useful for testing)
- COOL is first of all an **Interval-of-Validity** database: Each data-item is associated with a validity-range
  - Versioning of data-sets is also allowed

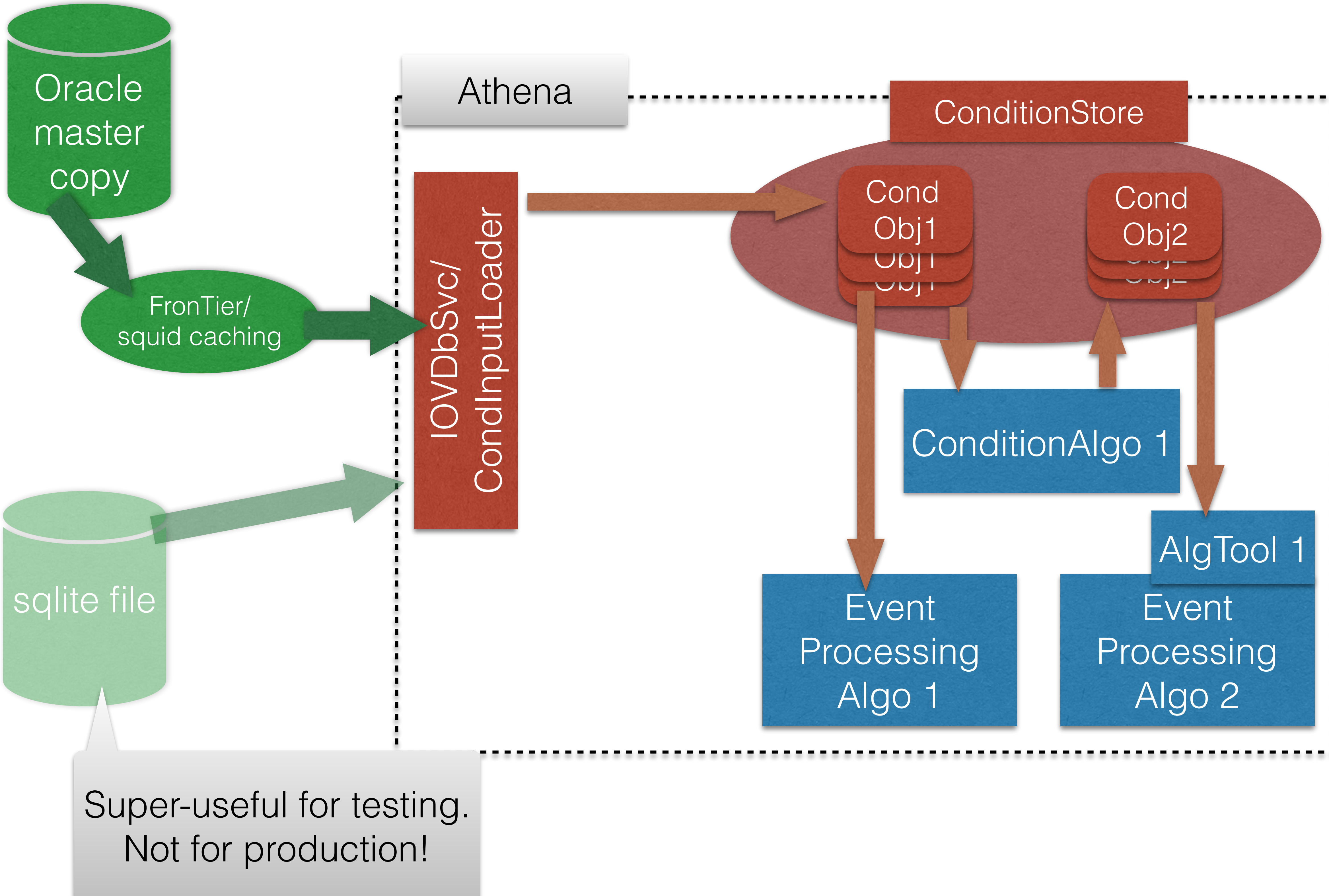
How we store conditions (2/2)

- Data in COOL is organised in **Folders** like a file-system directory structure
  - Each folder holds one kind of data
  - A folder can be sub-divided into '**channels**', each with it's own sequence of IOV
    - Used mostly for Detector-Control data (temperatures, voltages, …)
  - Validity interval can be expressed as time-stamp (nanoseconds) or as run/lumiblock pair
- Different versions of the same data can be distinguished by **Tags**
  - Each tag (can) described a complete set of channels and IOVs spanning the full history of data-taking
  - Concept of hierarchical tags: One global tag specifies which version (tag) of each folder gets used
- We use many different COOL database schemas: Dedicated online/offline database schema for each subsystem.
- The payload in each IOV in each Folder, Tag and Channel are `coral::AttributeList` (a set key-value pairs)
- Historically, we used also conditions stored in POOL file referenced in the COOL-IOV database

COOL Folders, tags & channels



Conditions data flow



ConditionStore and DetectorStore

- The ConditionStore and the DetectorStore are also instances of StoreGate (eg the EventStore) but with a different lifetime policy
- Before athenaMT, we used the DetectorStore for all non-event data
- Today, with athenaMT:
  - ConditionStore: Data with **varying** lifetime
    - Payload objects go into dedicated CondCont that can hold multiple versions. Allow to keep Conditions for multiple IOVs in memory
  - DetectorStore: Data with **infinite** lifetime (geometry …)
  - (StoreGateSvc: Data with lifetime of one event)

Conditions Algorithms

- Purpose: Process raw conditions objects and prepare derived conditions objects
    - Example: Take raw HV values and compute correction factors
- Not necessary if the conditions object in question can be used directly by the client
- Conditions algorithms can have multiple input conditions objects
- Chains of conditions algorithms are possible (and quite common, actually)
    - The chaining is actually done by the scheduler based on data-dependencies
- Historically, conditions handling was done by AlgTools or Services through call-backs (not MT-safe)

How to access conditions: 1/2 (configuration)

Need to specify:

- The **database schema** and the **COOL-Folder** set by jobOptions:
  - old config:
```
conddb.addFolder("LAR_OFL","/LAR/Identifier/OnOffIdMap",
className="CondAttrListCollection")
```
  - new config:
```
result.merge(addFolders(configFlags,"/LAR/NoiseOfl/CellNoise","LAR_OFL",
className="CondAttrListCollection"))
```
  - The data-type you'll get depends on the folder. In practice:
    - AthenaAttributeList for single-channel folders (shallow wrapper around coral::AttributeList)
    - CondAttrListCollection for multi-channel folders (a map of AttributeLists)
    - Whatever is stored in a POOL file for POOL-referenced storage (kind of deprecated)
- The **tag**, eg the version we are using. Typically done via the **global tag**, one of the basic parameters fed into a job.
- The actual pice of information you are getting depends also on the **time-stamp** of the event being processed. That's taken care of by the framework

How to access conditions: 2/2 (C++)

In the Algorithm or AlgTool:

- Declare a SG::Read**Cond**HandleKey<*ConditionObj*> in the header

```
SG::ReadCondHandleKey<ILArPedestal>
m_pedestalKey{this,"PedestalKey","LArPedestal","SG Key of Pedestal CDO”};
```

- Initialise it in the component’s `initialize()` method

```
ATH_CHECK(m_pedestalKey.initialize());
```

- Get a ReadCondHandle out of the ReadCondHandleKey in the execute() method

```
SG::ReadCondHandle<ILArPedestal> pedHdl(m_pedestalKey,ctx);
const ILArPedestal* peds=*pedHdl;
```

One more dereferencing step to get the object valid for the current event

- Use it!

```
const float p=peds->pedestal(id,gain);
```

(Example lines taken from LArRawChannelBuilderAlg)

How to write a ConditionsAlgorithm (and Conditions Data Object) 1/3

- Conditions data is data and conditions algorithms are algorithms with minor changes wrt to regular event processing algorithms
- Ingredients:
  - Base-class remains AthAlgorithm or AthReentrantAlgorithm
  - Dedicated Read/Write Handles
    - `SG::ReadCondHandleKey<ConditionObj>`
    - `SG::WriteCondHandleKey<ConditionObj>`
- Few constraints on output conditions objects: They are transient only (not I/O worries), can be complicated even with algorithmic capabilities (but the obj needs to be const!)
  - To be StoreGate-compliant, they need to have a CLID defined:

```
#include "AthenaKernel/CondCont.h"
CLASS_DEF( LArPedestalMC, 27770770,1)
CONDCONT_DEF(LArPedestalMC,251521960, ILArPedestal);
```

clid obtained by running the script
`lxplus> clid LArPedestalMC`

… and
`lxplus> clid "CondCont<LArPedestalMC>"`

How to write a ConditionsAlgo (2/3)

- **Header:**
  - Declare read/write CondHandleKeys + CondSvc

```
SG::ReadCondHandleKey<ILAruA2MeV> m_lAruA2MeVKey{this,"LAruA2MeVKey","LAruA2MeV","SG key of uA2MeV object"};
SG::ReadCondHandleKey<ILArDAC2uA> m_lArDAC2uAKey{this,"LArDAC2uAKey","LArDAC2uA","SG key of DAC2uA object”};
SG::WriteCondHandleKey<LArADC2MeV>m_ADC2MeVKey{this,"LArADC2MeVKey","LArADC2MeV","SG key of the resulting LArADC2MeV object"};
ServiceHandle<ICondSvc>m_condSvc{this,”CondSvc","CondSvc"};
```

- **`initialize()`**:
  - Initialize all Read/WriteHandles and CondSvc (like any other Service)

```
ATH_CHECK(m_condSvc.retrieve());
ATH_CHECK(m_lAruA2MeVKey.initialize());
ATH_CHECK(m_lArDAC2uAKey.initialize());
//Write handle
ATH_CHECK( m_ADC2MeVKey.initialize() );
```

  - Register the output key with the CondSvc

```
ATH_CHECK(m_condSvc->regHandle(this,m_ADC2MeVKey).isFailure());
```

- Example taken from LArADC2MeVCondAlg

How to write a ConditionsAlgo (3/3)

- **`execute()`**:
  - Get the output handle and check if it is already valid:

```
SG::WriteCondHandle<LArADC2MeV> writeHandle{m_ADC2MeVKey,ctx};
 if (writeHandle.isValid()) {
  ATH_MSG_DEBUG("Found valid write handle");
  return StatusCode::SUCCESS;
}
```

  - Get the input data objects and determine their IOV range

```
EventIDRange rangeIn,rangeOut;

SG::ReadCondHandle<ILAruA2MeV> uA2MeVHdl{m_lAruA2MeVKey,ctx};
const ILAruA2MeV* laruA2MeV{*uA2MeVHdl};
if (!uA2MeVHdl.range(rangeOut)){ … error reporting }

SG::ReadCondHandle<ILArDAC2uA> DAC2uAHdl{m_lArDAC2uAKey,ctx};
const ILArDAC2uA*larDAC2uA{*DAC2uAHdl};
if (!DAC2uAHdl.range(rangeOut)){ … error reporting }

rangeOut=EventIDRange::intersect(rangeOut,rangeIn);
```

Please be careful here!
Getting the range wrong will lead to hard-to-debug problems:
Wrong calibration applied for some events

  - Create the result the result:

```
std::unique_ptr<LArADC2MeV>lArADC2MeVObj=std::make_unique<LArADC2MeV>(m_larOnlineID, … );
```

  - and record it

```
ATH_CHECK(writeHandle.record(rangeOut,std::move(lArADC2MeVObj)));
```

Questions?