POOL Collections

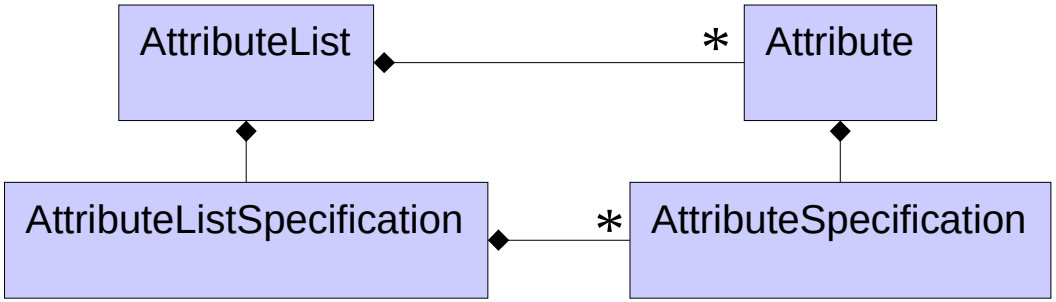
- Collection Overview
- Meta Data Interface
- Design / Functionality
- Implementation Details
- Performance
- MultiCollections
  (Collection of Collections)
- Summary and Outlook

Collection Overview

- Collections keep track of large sets of objects and their description (meta data)
- **Explicit Collections:**
  - Explicit list of object references
  - Allows logical grouping
  - Queryable if associated with meta data
  - Use case: Event collections, fast pre-selection based on event meta data
  - Current implementations: *MySQL* and *Root*
- **Implicit Collections**
  - Defined by the physical containment of the objects
  - Provide user friendly access to all objects in a given database container
- Both collection types use the same interface

Meta Data Interface

- Pool AttributeList component:
  - Type safe user interface to the meta data
  - Used by Pool FileCatalog and Collection component



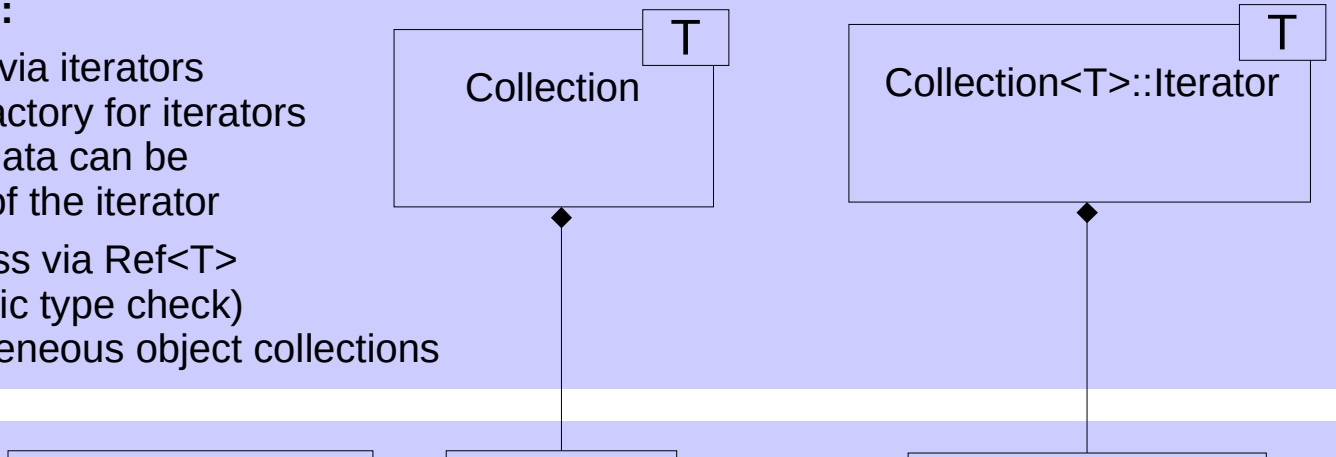
- AttributeList: Row in a table
- Attribute: Individual cell
- AttributeListSpecification: Description of the columns

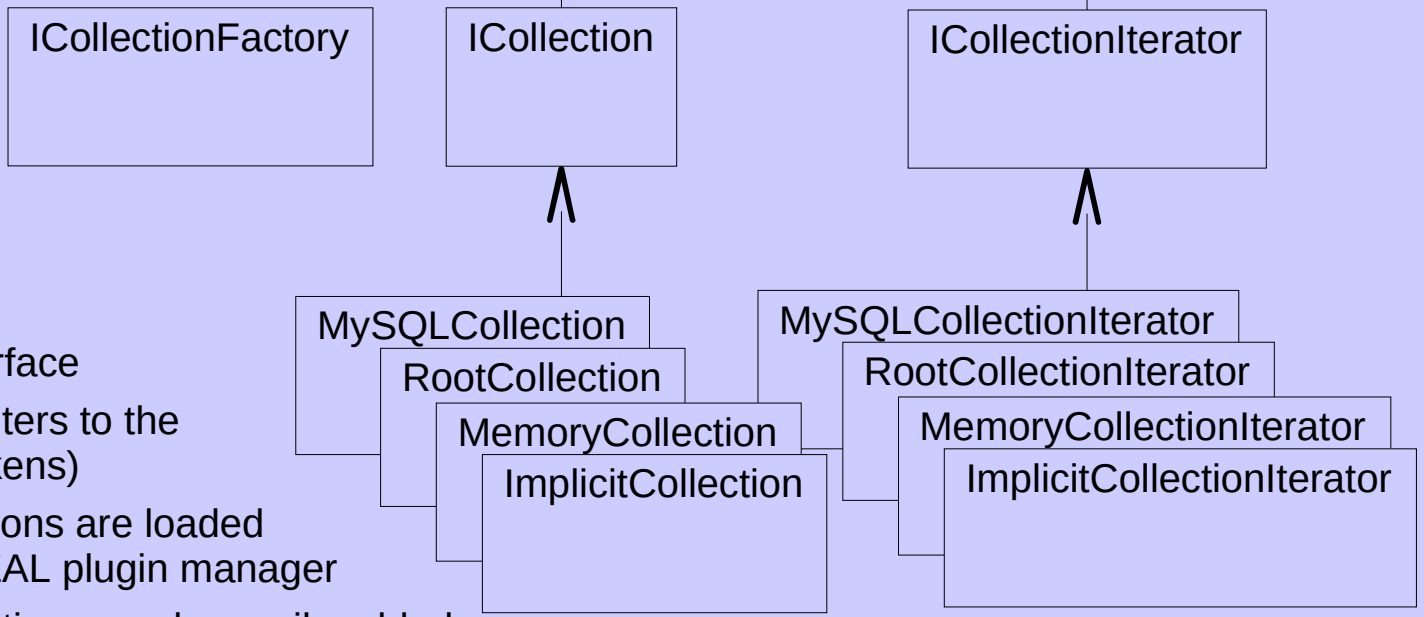
- Attribute:
  - May hold values of any type T
  - Has value semantics and may be used in STL containers
  - Provides type safe access to the stored value of type T
  - Allows for generic stringification without knowledge of type T
  - Currently supported types:
    std::string, double, float, bool, (unsigned) int, ...

Collection Design

Type safe User interface:

- Objects are accessed via iterators
  - Collection acts as factory for iterators
  - Selection on meta data can be applied at creation of the iterator
- Type safe object access via Ref<T> smart pointers (dynamic type check)
  - Support for heterogeneous object collections





Implementation:

- Common abstract interface
- Untyped: Manage pointers to the persistent objects (Tokens)
- Concrete implementations are loaded dynamically via the SEAL plugin manager
- Further types of Collections can be easily added

* A Token uniquely identifies a persistent object by the database FID, container name, classID storage technology and the ID of the object in the container

Collection How-to

```
Collection<Event> srcColl( dataSvc,
                           "MySQLCollection","mysql://test:...",
                           "CosmicRun123",
                           ICollection::READ );

Collection<Event> dstcoll( dataSvc,
                           "RootCollection", "",
                           "MyEvents",
                           ICollection::UPDATE );
```

Server side selection

```
Collection<Event>::Iterator iter = srcColl.select("NHiggs>0");
```

Client side selection

```
// *iter is an Event
while(iter.next()){
   if( mySelection( iter.attributeList() ) ){
      AttributeList myMetaData = analyze( *iter );
      dstcoll.add( *iter, myMetaData );
   }
}
dstcoll.commit();
```

Implementation Details

- Implicit collections
  - User defines database (PFN,LFN,FID), container and object type (via template parameter)
  - A list of tokens is directly retrieved from the container and the objects are made available via Ref<T> if object and user defined type is compatible
  - Read only ; no selection on meta data (due to the nature of this collection type)
- Explicit collections
  - Access modes: CREATE_AND_OVERIDE, CREATE, UPDATE, READ
    Entries cannot be deleted or manipulated once they are committed
  - Meta data schema is fixed at creation
  - MySQL and Root implementation uses different query languages: SQL vs. C++
    A simple parser could be implemented: AND -> && , = -> == , ...
- MySQL explicit collections:
  - Provides two kinds of client iterators:
    - *"FetchOne"*: Fetches server selected entries row by row
    - *"FetchAll"*: Caches information of server selected entries at creation
      Reduces network latency

Implementation Details

- MySQL explicit collections (cont.)
  - For the selected entries all information is retrieved from the server
    - Implementation of a smart access method, which only requests the Token or a subset of the meta data attributes is currently discussed.
  - Tokens are stored as strings:
    ```
    [DB=DAF83A9C-9DC5-D711-9201-000347F31C25][CNT=helm_Root_1000000_100_Container]
    [CLID=4E1F4DBB-1973-1974-1999-204F37331A01][TECH=00000202][OID=00000003-000CFA0F]
    ```
    - Introduction of link tables will reduce the table size significantly. Currently beeing implemented
  - The use of indices improves selection speed dramatically (up to a factor of 100: 10^6 entries, 0.1% cut on one attribute)
    - The setup of indices has to be done directly on the server.
  - MySQL transfers attributes as strings. With the upcoming MySQL release a binary transfer protocol will be provided.
  - Client is based on MySQL++ binding
    - It will be replaced by a more performant and vendor neutral product
    - The utilization OTL binding is currently investigated: Prototypes for Collection and File Catalog

Implementation Details

- ROOT explicit collections
  - Meta data and Tokens are stored in a simple TTree
    (Fully accessible in an interactive Root session)
  - Tokens are stored as strings, but efficiently compressed by Root
  - Selection is performed using TTreeFormula
  - Allows reading of meta data sub-sets
  - Provides remote access via rootd or rfio
  - Tokens and/or metadata are fetched on demand
  - Allows server side selection (very experimental)
    Simple server side C++ script that returns a TEventList to the client side iterator
  - Utilization of the Pool file catalog to locate/register RootCollection
    files will be implemented.

Performance

- Test setup:
  - 10^5 collection entries referencing trivial, identical test objects in one local pool database
  - 100 meta data attributes per object (50 ints, 50 floats randomly distributed)
  - Performed on a PC: P4 1.8 GHz 256 MB (ROOTMARKS = 419.7)
    rootd server: PC of same configuration in the same office
    MySQL server: lxshare070d.cern.ch
- Reading of attribute list using MySQL has improved by a factor of 7 compared to the values presented 3 weeks ago.
- Overall MySQL client performance will improve
- Use of MySQL indices will speed up the selection use case (last row) by a factor of 100, then it will outperform the Root implementation (local file).
- Without indices selection on the MySQL server is unexpectedly slow.
  Even a client side selection on a remotely accessed Root file is faster.
- Accessing all objects takes the same time for implicit and Root explicit collections:
  The time is spent to perform the dynamic type check and
  to provide the objects via Pool DataSvc.

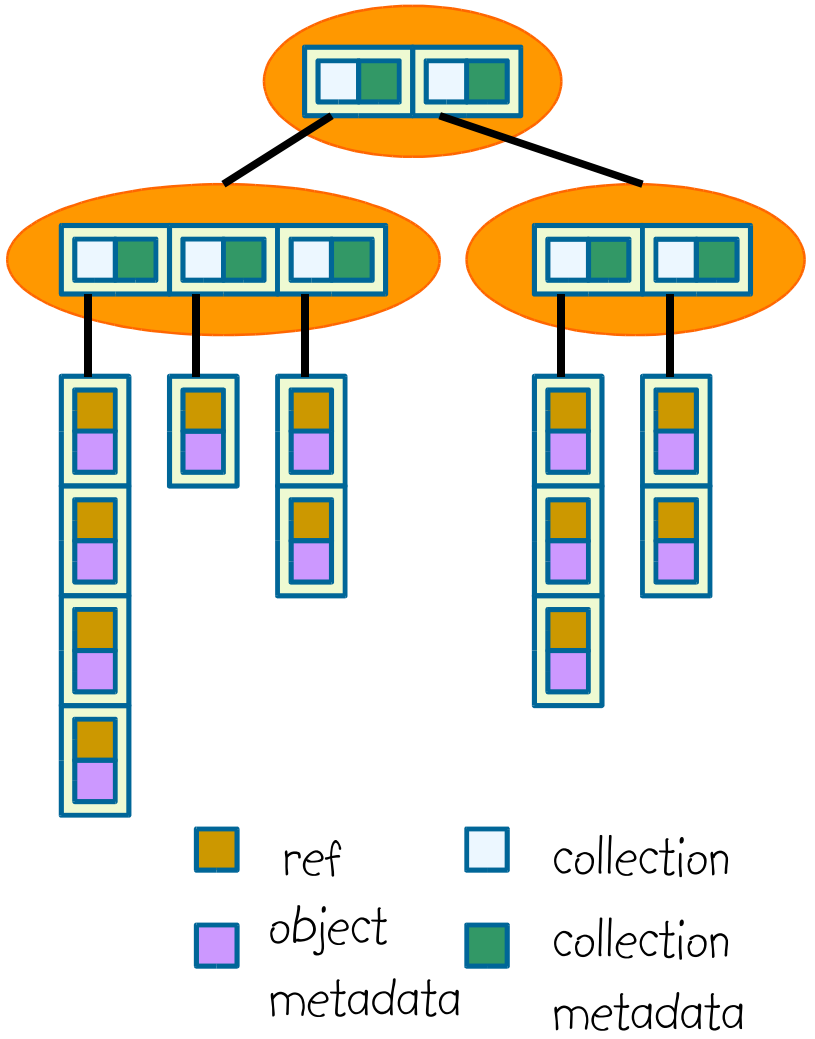
PRELIMINARY !

Real (CPU) mean time [sec] spent to read 10^5 entries	MySQL FetchAll	MySQL FetchOne	Root local file	Root rootd	Implicit
Object access	-	33.0 (32.6)	15.8 (15,7)	17.1 (15.7)	15.6 (15.2)
Read 100 Attributes	-	24.3 (23.6)	3.9 (3.9)	5.7 (3.9)	
Read 2 Attributes	-	-	<0.1	1.0 (0.1)	
Select ~1‰ (cut on one att.) and access objects (91)	3.7 (0.3)	3.3 (0.3)	0.3 (0.3)	1.5 (0.4)	

Multi-Collections

- Collection of Collections
- References to Collections of any type can be managed
  - Explicit Collections (MySQL, Root)
  - Implicit Collections
  - **other MultiCollections**

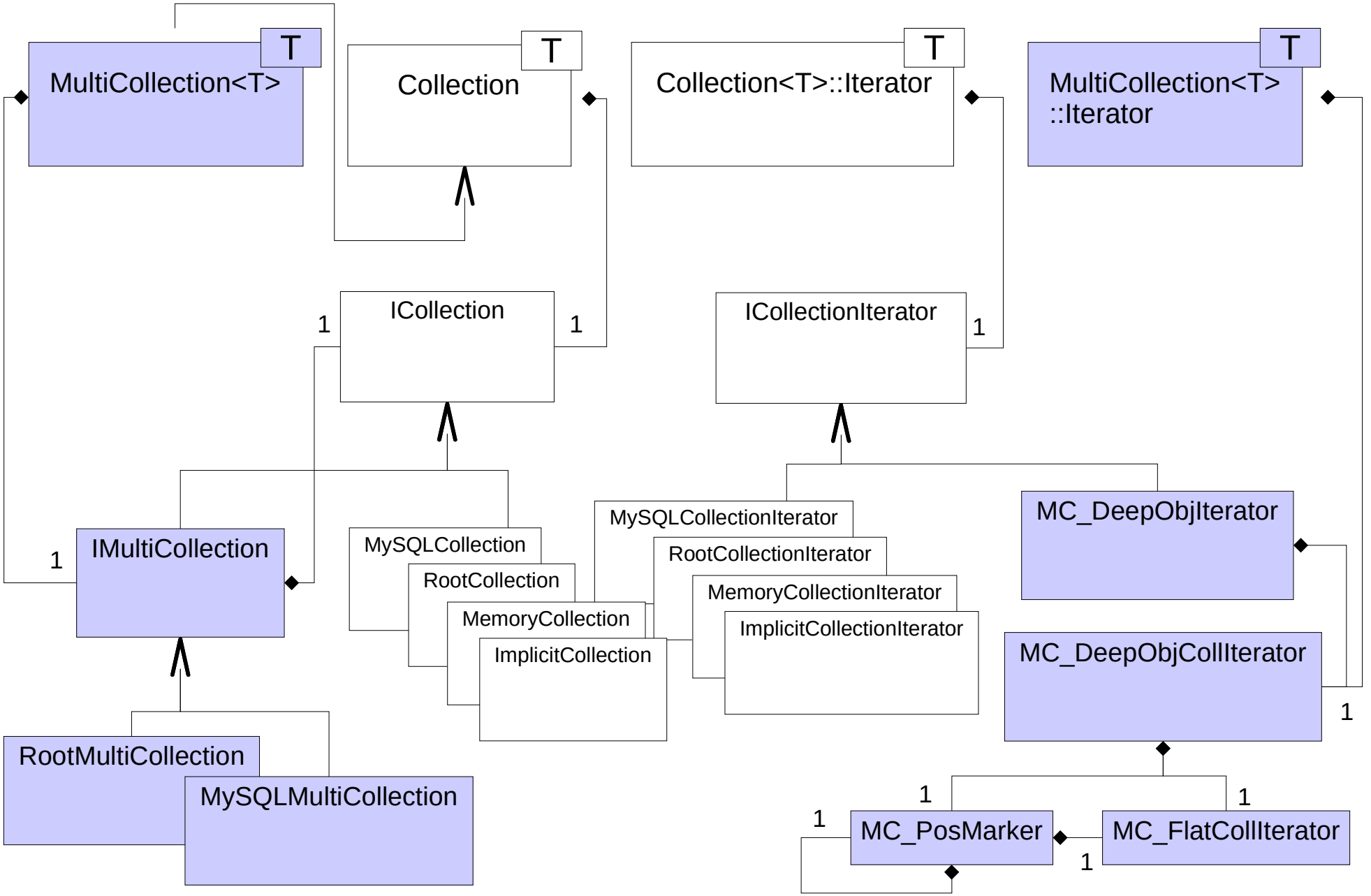
    ⇒ A hierarchical tree of collection can be build
- Allows complex user queries on collection and object meta data
  - all events with >4 selected electrons from runs with working ECAL and selected calibration, alignment setup
- Allows distributed storage of large object collections



Multi-Collection Design

- Implemented on top of the existing Collection implementation layer using the "proxy" pattern
- The MultiCollection proxy utilizes the existing implementations of explicit object collections to store references to collections
    - It writes dummy Tokens
    - It silently adds to the users collection meta data the necessary parameters to create collections via the ICollectionFactory
    - It adds the MultiCollection specific features to the object collection implementation interface
    - It automatically benefits from improvements of the utilized object collection implementations
- Two MultiCollection "implementations" : *MySQLMultiCollection* and *RootMultiCollection*

Multi-Collection Design



MultiCollection How-to

- In read mode MultiCollections offer the same look and feel as flat object collections:

```
Collection<Event> coll( dataSvc,
                        "RootMultiCollection",
                        "root://pcepsft02.cern.ch",
                        "CollisionRuns",
                        ICollection::READ );

typedef Collection<Event>::Iterator EventIter;

EventIter iter = coll.select( "NHiggs>0",
                              "runDate>20030806" );


while(iter.next())   // *iter is an Event
   analyze( *iter);
```

- *Caveat:* Because of the different query languages in Root and MySQL explicit collections, a selection is currently only possible if the MultiCollection is homogenous

Summary / Outlook

- Explicit and implicit collections are ready to use
- Integration with experiments:
  - LHCb plans to utilize Pool collections for their EventSelector component
  - ATLAS already utilizes Pool collections in the AthenaPool EventSelector. Integration of Atlas production bookkeeping (AMI) and Pool Collections is on the way.
  - CMS plans to deploy Pool collections as user-level event collections.
- Future developments:
  - Python Interface
    - Tools for merging, extracting, transforming or extracting a list of needed database files for a specific job
  - Collection registry and management utilities
  - Synchronization of collections containing the same objects but different meta data attributes (like TTree friends)

Documentation / Examples

- General:

  http://lcgapp.cern.ch/project/persist/metadata/index.html

- Collections:

  http://lcgapp.cern.ch/doxygen/POOL/POOL_1_3_3/doxygen/classpool_1_1Collection.html

  http://lcgapp.cern.ch/cgi-bin/viewcvs/viewcvs.cgi/pool/Examples/TinyEventWriter/?cvsroot=POOL

  http://lcgapp.cern.ch/cgi-bin/viewcvs/viewcvs.cgi/pool/Examples/TinyEventReader/?cvsroot=POOL

- MultiCollection:

  http://lcgapp.cern.ch/cgi-bin/viewcvs/viewcvs.cgi/pool/Tests/MultiCollection_BasicFunctionality/?cvsroot=POOL

  http://lcgapp.cern.ch/doxygen/POOL/POOL_1_3_3/doxygen/classpool_1_1MultiCollection.html

- AttributeList (meta data representation):

  http://lcgapp.cern.ch/project/persist/metadata/docs/AttributeList-pool-component.ps

Meta Data How-to

```
// setup the of the specification
AttributeListSpecification spec;
spec.push_back("A","int");
spec.push_back<double>("B");

// setup the AttributeList using the specification
AttributeList list(spec);
list["A"].setValue(int(42.0));       // type safe
list["B"].setValueAsString("5.23");  // generic

// In addition to keyed Attribute access
// there are a couple of iterators (ordered and non ordered)
typedef  AttributeList::const_iterator  Iter;

for( Iter iter = list.begin(); iter != list.end(); ++iter){
  // *iter is an Attribute
  string name = iter->spec()->name();
  string val = iter->getValueAsString();
  int ival;
  iter->GetValue(ival);  // exception is thrown in second
}                        // iteration, because a double is
                         // expected
```

MultiCollection How-to

The MultiCollection<T> interface provides MultiCollection specific features

```
MultiCollection<Event> srcColl( dataSvc,
                                "MySQLMultiCollection","..."
                                "CollisionRuns",
                                ICollection::READ );

MultiCollection<Event> dstColl( dataSvc,
                                "RootMultiCollection",""
                                "MyFavouriteRuns",
                                ICollection::CREATE );

typedef MultiCollection<Event>::Iterator RunIter;

RunIter iter = srcColl.selectObjCollections( "runDate = 20030806" );

float intLumi = 0.0;
while( Iter.next() ){          // *iter is a Collection<Event>
   AttributeList runMD = iter.attributeList();
   dstColl.add( *iter, runMD );
   float lumi;
   runMD["lumi"].GetValue(lumi);
   intLumi += lumi;
}
```