POOL 1.7.0 User guide

27 July 2004

Table of Contents

About this document

The POOL project [http://pool.cern.ch] has released version 1.7.0 of the LCG persistency framework.

This User Guide addresses mainly framework developers on the experiment side who are involved in the integration of POOL into their existing software systems. It is structured into an architectural overview, a chapter addressing C++ developer view of the POOL system, and finally a chapter from the point of view of deployment, covering issues like software installation, build system integration, and management of POOL file catalogs.

1. Introduction

The POOL project has been created to implement a common persistency framework for the LHC Computing Grid (LCG) application area. POOL can store multi-Petabyte experiment data and metadata in a distributed and grid enabled way. The project follows a hybrid approach combining C++ Object streaming technology, such as ROOT I/O, for the bulk data with a transaction safe relational database (RDBMS) store, such as MySQL. POOL is based a strict component approach - as laid down in the LCG persistency and blue print RTAG documents - providing navigational access to distributed data without exposing details of the particular storage technology.

1.1. Persistency Framework for LCG

Data processing at LHC [1] will impose significant challenges on the computing of all LHC experiments. The very large volume of data ? some hundred Petabytes over the lifetime of the experiments ? requires that traditional approaches, based on explicit file handling by the end user, be reviewed. Furthermore, the long LHC project lifetime results in an increased focus on maintainability and change management for the experiment computing models and for core software such as data handling. It has to be expected that, during the LHC project lifetime, several major technology changes will take place and experiment data handling systems will be required to adapt quickly to the changes in the environment or in the physics research focus.

In the context of the LHC Computing Grid (LCG [2]), a common effort to implement a persistency framework underlying the different experiment frameworks has been started in April 2002. Since that time, project POOL [3] (acronym for POOL Of persistent Objects for LHC) has ramped up to about 10 FTE from the IT/DB group at CERN and from the experiments located at CERN and at outside in-

stitutes.

For POOL as a project, the strong involvement of the experiments from the earliest stages is very important to guarantee that the experiments' requirements are injected and implemented by the project, without introducing too much distance between software providers and users. Many of the POOL developers are part of an experiment software team and will be directly involved in the integration of POOL into their experiments' software framework.

1.2. Component Architecture

POOL as a LCG Application Area project follows closely the overall component base architecture laid down in the LCG Blueprint RTAG report [4]. The aim is to follow as much as possible a technology neutral approach. POOL therefore provides a set of service APIs - often via abstract component interfaces - and isolates experiment framework user code from the details of a particular implementation technology. As a result, the POOL user code is not dependent on the implementation API or header files. POOL applications do not directly depend on implementation libraries. Even though POOL implements object streaming via ROOT-I/O [10] and uses MySQL [11] as an implementation for relational database services, there is no link time dependency on the ROOT or MySQL libraries. Back end component implementations are instead loaded at runtime via the SEAL [5] plug-in infrastructure. The main advantage of this approach is that changes required to adapt to new back end implementations are largely contained inside the POOL project, rather than affecting the much larger code base of the experiment frameworks or end user code. Achieving this goal and still keeping the system open for new developments is only possible by constraining very consciously the concepts exposed by POOL. The project has made a significant effort to identify a minimal API that is just sufficient to implement the data management requirements, but that still can be implemented using most implementation technologies that are available today.

1.3. Hybrid Technology Store

The POOL system is based on a hybrid technology approach. POOL combines two main technologies with quite different features into a single consistent API and storage system. The first technology includes so-called object streaming packages (e.g. ROOT I/O) that deal with persistency for complex C++ objects, such as event data components. Often this data is used in a write-once, read-many mode, and concurrent access to the data can therefore be constrained to the simple read-only case. In particular, this simplifies the deployment, as no central services are required to implement transaction or locking mechanisms. The second technology class provides Relational Database (RDBMS) services, such as distributed, transaction consistent, concurrent access to data that still can be updated. RDBMS based stores also provide facilities for efficient server side query evaluation. The aim of this hybrid approach is to allow users to be able to choose the most suitable storage implementation for different data types, use cases, and deployment environments. In particular, RDBMS based components are currently used heavily in the area of catalogs, collections, and their metadata, while streaming technology is used for the bulk data.

1.4. Navigational Access

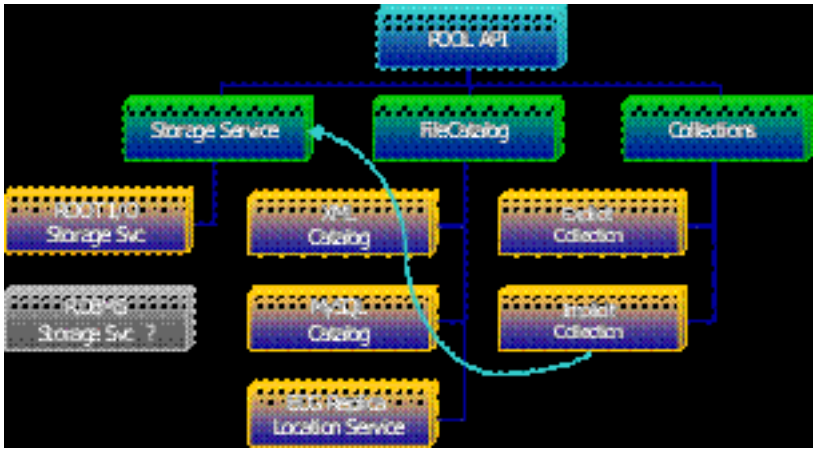
POOL implements a distributed store with full support for navigation between individual data objects. References between objects are transparently resolved ? meaning that referred-to objects are brought into the application memory automatically by POOL as required by the application. References may connect objects in either the same file or spanning file and even technology boundaries. Physical details such as file names, host names, and the technology that holds a particular object are not exposed to reading user code. These parameters can therefore easily be changed, which allows optimizing the computing fabric with minimal impact on existing applications.

2. POOL by examples

3. POOL Architecture

3.1. Project breakdown into packages

The internal structure of POOL follows closely a domain decomposition that has been previously described largely in the report of the Persistency RTAG [7] that preceded the POOL project. In this paper, we give only a brief overview of the overall project structure and the main responsibilities and collaboration between its main components. A more detailed description of component implementations can be found in [8] and [9]. Component design documents are available [3].

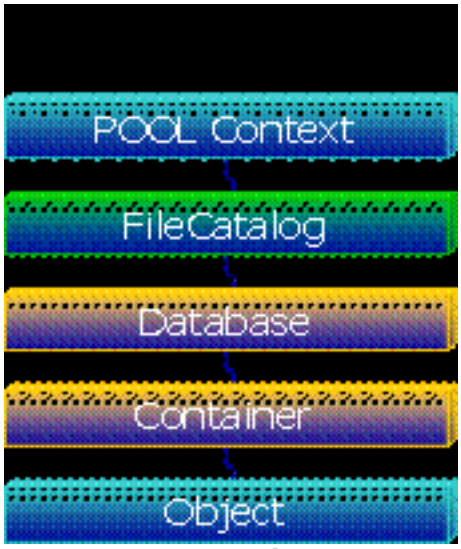


POOL breakdown in components

3.2. Storage hierarchy

The storage hierarchy exposed by POOL consists of several layers (shown in Fig. 2), each dealing with finer granularity objects than the layers above. The entry point into the system is the POOL context, which holds all objects that have been previously obtained. Each context may reference objects from any entry in a given File Catalog. Currently POOL supports a single File Catalog at a time. This may be extended in later releases. By specifying the file catalog for a particular application, one can determine the scope of objects that the application can see.

Since V1.1, the context is also the granularity of user level transactions that POOL provides. All objects that have been marked for writing in a context will be written together at the context transaction commit. The persistency service subcomponent of the storage service keeps a list of open database connections and issues individual low level commits on the database level as required.



POOL Storage Hierarchy

Each POOL database (entry in the POOL file catalog) has a well-defined major storage technology. Currently only one major technology is supported, namely ROOT I/O files, but the RDBMS storage manager prototype will be a first extension to prove that such independence has indeed been achieved.

POOL databases are internally structured into containers, which are used to group persistent objects in the database. POOL containers in the same database may differ in their minor technology type but not in their major type (e.g. a single ROOT I/O database file may hold containers of ROOT-tree and ROOT-keyed type).

Some storage service implementations may constrain the choice of data types that can be kept in a container simultaneously. For example, a ROOT tree based container does not allow storing arbitrary combinations of unrelated types in the same container, while a ROOT directory based container does allow this.

3.3. File catalogue

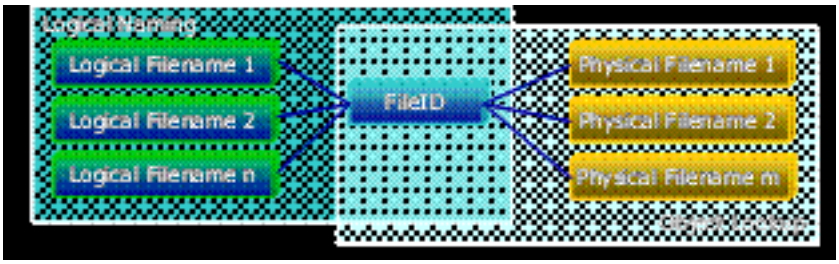
The main responsibility of the File Catalog is to keep track of all POOL databases (usually files that store objects) and to resolve file references into physical file names, which are then used by lower level components, such as the storage service, to access file contents. More recently, the POOL file catalog has been extended to allow simple metadata to be attached to each file entry. This infrastructure is shared with the collection implementation.

When working in a Grid environment, a File Catalog component based on the EDG Replica Location Service (RLS) is provided to make POOL applications grid aware. In this case, file resolution and catalog metadata queries are forwarded to grid middleware requests.

For environments not connected to the grid, MySQL and XML based implementations of the component interface use a dedicated database server in the local area network (e.g. isolated production catalog servers) or local file system files (e.g. disconnected laptop use cases).

Files are referred to inside POOL via a unique and immutable file identifier (FileID), which is assigned at file creation time. This concept of a system generated FileID has been added by POOL to the standard grid model of many-to-many mapping between logical and physical file names to provide for stable inter-file references in an environment where both logical and physical file names may change after data has been written. The stable FileID allows POOL to maintain referential consistency between multiple files that contain related objects without requiring any data update (e.g. to fix up changes in logical or physical file names).

In addition, the particular FileID implementation that has been chosen for POOL is based on so-called Universally or Globally Unique Identifiers (UUID/GUID [12]). It provides another benefit, namely that GUID based unique FileIDs can be generated in complete isolation, without a central allocation service. This greatly simplifies the distributed deployment of POOL, as POOL files can be created without a network connection and later be integrated in larger store catalogs without any risk of clashes.



POOL File Catalog Mapping

3.4. Storage service and Conversion

The storage technology information from the File Catalog is used to dispatch any read or write operation to a particular storage manager. The task of the storage manager component is to translate (stream) any transient user object into a persistent storage representation that is suitable for subsequently reconstructing an object in the same state. The complex task of mapping individual object data members and the concrete type of the object relies on the LCG Object Dictionary component developed by the SEAL project. For each persistent class, this dictionary provides detailed information about the internal data layout, which is then used by the storage service to configure the particular backend technology (e.g. ROOT I/O) to perform I/O operations.

In addition to the existing storage service, which supports objects in ROOT trees and objects in ROOT directories, a prototype implementation of a RDBMS base store is underway. As the POOL program interface hides the details of their internal implementation, the user can easily adapt to new requirements or technologies with very little change to the application code.

During the process of writing an object, a unique object identifier is defined, which can later be used to locate the object anywhere within a POOL store.

3.5. Object cache and references

Once an object has been created in application memory by a POOL read operation or a user write operation, the object is maintained in an object cache (also called Data Service). This speeds up repeated accesses to the same object and controls the object lifetime. The implementation provided with POOL uses a templated smart pointer type (pool::Ref<T>) that implements - close to the ODMG standard - object loading on demand and automatic cache management via reference counting on any cached object.

Alternatively, an experiment may decide to clean all objects from the cache explicitly via an API or to replace the POOL object cache with its own implementation using the cache interface defined in POOL.

As the inter-object references can be stored as part of a persistent object, and as POOL will transparently load objects on demand, the Ref is also the main building block to construct persistent associations between objects. These may be local to a single file or may cross-file and technology boundaries. Object lifetime management and object caching is coupled closely to the user implementation language ? currently C++ for LHC offline code. This POOL component therefore acts as a C++ binding of POOL and encapsulates most functional changes that would be required in case native support of an additional language should become a requirement.

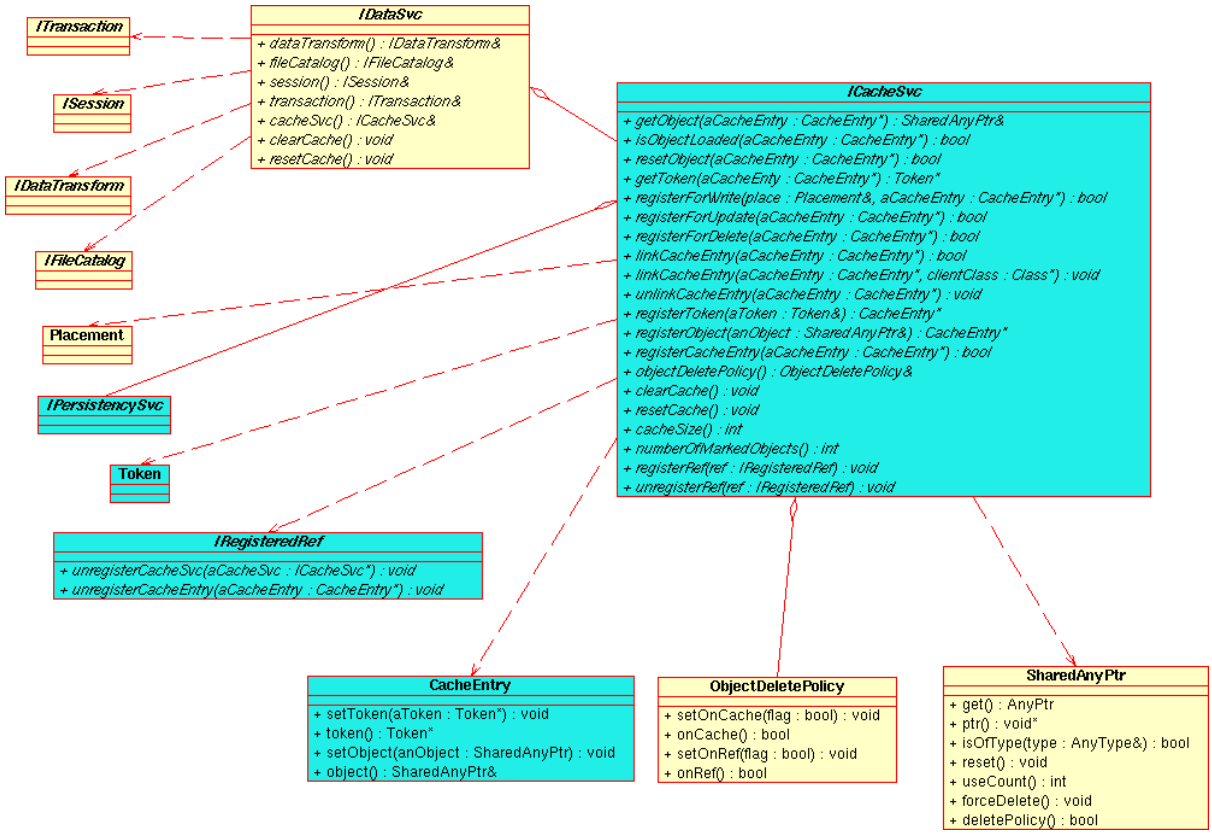
3.6. Collections

The collection support provided by POOL allows maintaining large-scale object collections (e.g. event collections) and should not be confused with the standard C++ container support that is provided by the POOL storage service. POOL collections can be optionally extended with metadata (currently only simple lists of attribute-value pairs) to support user queries that select only collection elements that fulfill a query expression. POOL supports several different collection implementations based either on the RDBMS back end or on the ROOT/IO streaming layer. Collections can be defined explicitly ? via adding each contained objects explicitly ? or as an implicit collection, which refers to all objects in a given list of databases or containers. As the different collection implementations adhere to a common collection component interface, the user can easily switch from a collection using ROOT trees in local files to a collection using a database implementation that allows distributed access and server side query evaluation.

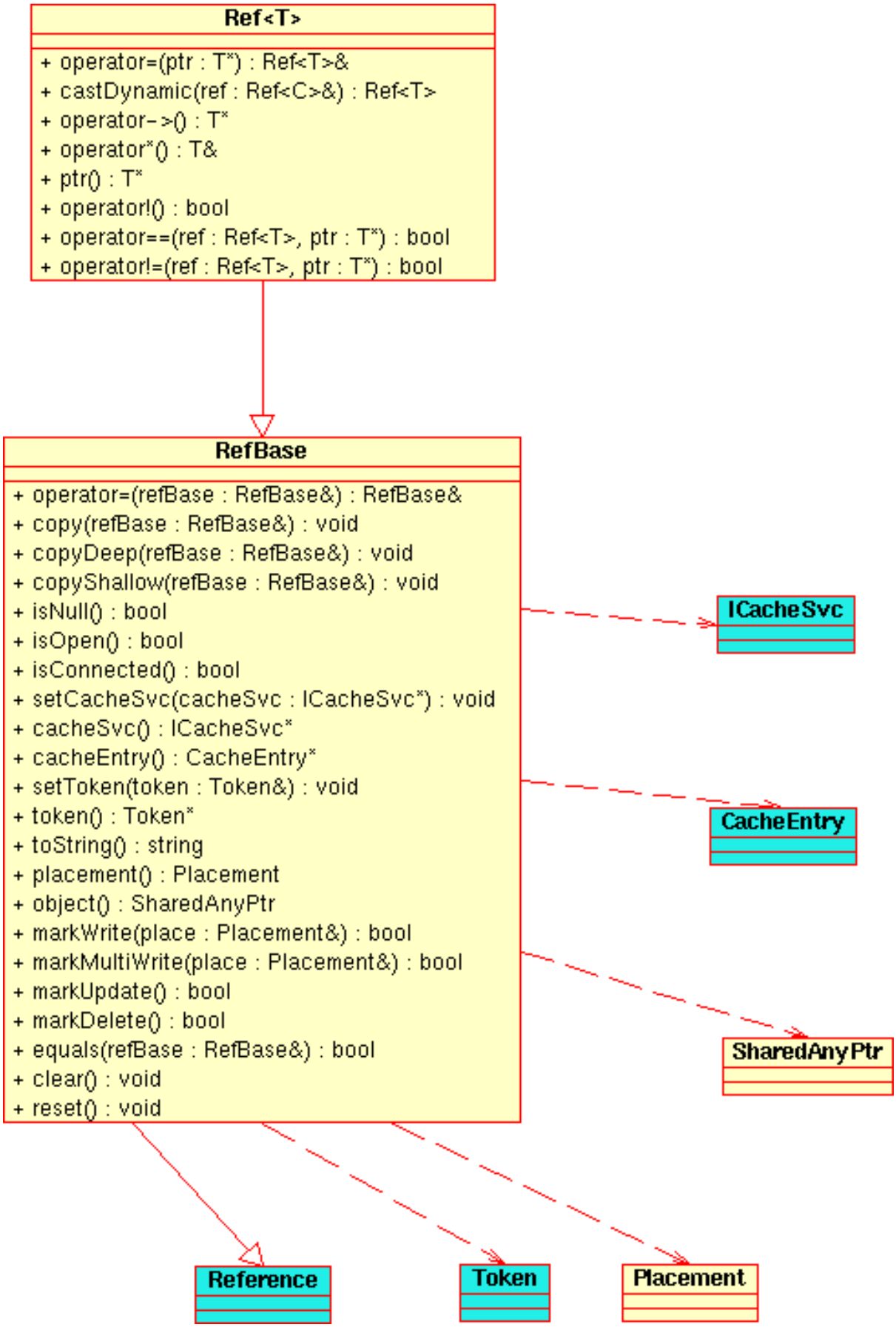
4. DataSvc component: User level semantics

4.1. Public classes UML diagram

The following UML class diagrams are describing the public interfaces of the DataSvc component, respectively for the DataSvc and the Ref API. With cyan-filled boxes are represented the classes which are not intended for the End-User access, but only for Developer-User.



UML class diagram of the DataSvc public interfaces



UML class diagram of the Ref public interfaces

4.2. DataSvc component User level interface

The DataSvc API is mainly based on two classes: the Ref<T> class, which handles the persistency of the individual user object, and the IDataSvc interface, which defines the storage system access point. This two classes exposes the methods to exploit the main POOL object storage functionaties for the

general user. Here follows a brief description of the usage of the API.

4.2.1. Setting up of a POOL storage system environment

The POOL storage system environment is mainly defined by a File Catalogue, which is described by the IFileCatalog interface. The Catalogue, which can be implemented in different technologies depending on the use case context, contains the physical locations of the 'files' (as generalized concept) composing the storage system.

Optionally, two more element can be specified in the POOL setting up (see PersistenySvc documentation for details):

- A set of functions defininig customized class-specific trasformations between transient and persistent shapes. The set has to be provided as a registry implementing the IDataTransform interface.
- A function to re-locate the objects in the storage system, implementing the ITokenValidator interface.

4.2.2. Database handling: sessions, connections, and transactions

The IDataSvc interface exposes to the end user the database handling API of the PersistencySvc component, which can be summarized as follows (for details one can refer to the related documentation):

- The Session object provides functions to handle the overall database operation: transactions, policy for the implicit access, explicit access.
- The Connection does not need to be handled explicitly: once a database needs to be accessed (for both writing or reading), a connection is open in the required mode, applying the policy previously defined by the user (see examples below).
- The Transaction is applied to the global storage system and needs to be explicitly handled by the user, for both READ and UPDATE mode. Changes in the UPDATE mode are applied at the commit time in the data storage.

4.2.3. Object Navigation using Ref<T>

The POOL framework manages the persistency at level of the individual object.

User-created objects can be made persistent creating a corresponding entry in a specific storage system, and with the selected database technology. Conversely, persistent item in a database can be reloaded in memory creating instances of their original classes. The mapping between objects in the user application and items in the storage system can be convenienlty described using the Ref class.

The pool::Ref<T> class is a template wrapping of a generic object pointer of class T. It provides access to the embedded object member, maintaining the basic syntax (in many respects) of a C++ pointer. In addition, it confers persistency capability on the enclosed object.

A Ref class is fully operating when associated to a database access point, specified through an DataSvc instance. The DataSvc object provides the necessary connection to operate on the storage system, handling the object I/O through a caching mechanism.

The object cache provides a centralized mapping between the transient and persistent descriptions, implementing a bookkeeping mechanism with reference counting for the object in use. The Ref instances referencing persistent objects are client of this object cache. Each Ref is associated to a corresponding entry in the object cache. Operation on the database are executed through the according to the underlying transactional scheme, described above. Within this scheme, the Ref class can operate in two modes:

-UPDATE. The Ref instance is constructed by specifying a pointer to the object and the IDataSvc pointer. Within an UPDATE transaction, the embedded object can be made persistent, updated, or de-

leted from the storage system.

-READ. The Ref instance is constructed by specifying a Token (object describing the location of a persistent item in the storage system) and the IDataSvc instance. Within a READ transaction, the transient object will be created 'on demand', as soon as it is accessed.

The Ref class also provides the semantics to define associations between persistent objects. A Ref instance can be declared as an attribute of a given class, defining in the correspondent object an embedded reference to another item in the storage system. The Ref class is recognized as a special type by the Storage Manager in both the writing and reading procedures. During a write procedure, the Storage Manager must resolve the Token object corresponding to the embedded Ref instance to make a persistent description of the association. During a read procedure, the Storage Manager creates the object containing an empty Ref attribute, which has to be properly initialized in order to point to the associated entry in the storage system. In both cases, the Storage Manager applies procedures provided by the cache service on the top through a call back mechanism.

As for any other read operation, the loading of the object referenced by an embedded Ref instance only happens when the object is accessed.

With this model, the navigation among associated objects is achieved in a transparent way, without specifying any details about the locations or the storage technology.

4.2.4. Pointer ownership

In agreement with common smart pointer concepts, the Ref class has been mainly designed to take over from the user the ownership of the enclosed pointer. However, the actual policy applied has been left somewhat configurable, allowing adaptations according to the particular integration context.

The default policy implements a shared ownership with reference counting - close to the one in boost::shared_ptr. In this model, when multiple Ref instances point to the same object, they share a single cache entry object, which acts as a proxy to the related object pointer. When all the Ref instances disappear, the cache entry is automatically deleted. This implies the destruction of the object, if no other actor is referencing it.

The alternative policy simply leaves the ownership of the object pointers to the user. The reference counting mechanism only holds for the cache entries shared among the Ref instances. In this case, however, the destruction of a cache entry does not effect the lifetime of the related object pointer.

4.2.5. Restriction of use of Ref<T>

In the current implementation, a Ref<T> object can be constructed on a given pointer C* only if the two classes are in the dictionary. The handling of class inheritance and object lifetime relies on the seal::reflection::Class instance corresponding to the involved classes.

4.2.6. Multi cache access

For some particular use case, it can be required to access objects of different categories from a given storage system through different object caches. The specific use case can bring up to two complicated configurations:

- Association between objects in different caches.

  As explained above, the embedded references are by default handled in the same object cache of the embedding object. However, it is possible to specify explicitly the cache where the referenced object should be handled. As explained in the examples, some special construct allows to write objects with reference to other objects in different caches, or to read them back creating the corresponding instances in the original caches.

- More caches sharing the same objects.

This use case has to be treated carefully, in particular when the ownership of the object pointers relies to the caches. The lifetime of the objects is controlled by a smart pointer with shared ownership, such that the reference counting is correctly handled, provided all the assignment of reference are executed through the smart pointer (and the 'naked' pointer is never used directly). The restrictions are explained further in the examples.

4.3. Example of usage

4.3.1. Construction of a DataSvc instance

- - Specifying the entire POOL environment in the DataSvc context:

  The parameter to specify are: File Catalogue (a technology-specific instance is constructed with the dedicated factory), Data Transform (register containing the shape transformation), Token Validator (function for token translations), Object Delete policy.

```
pool::DataSvcContext                                                ctx;
//          IFileCatalog*          myFileCatalog          =          ...
ctx.setFileCatalog(myFileCatalog);
//        IDataTransform*        myDataTransform        =        ...
ctx.setDataTransform(myDataTransform);
//         ITokenValidator*         myTokenValidator=         ...
ctx.setTokenValidator(myTokenValidator);
```

  The previous parameters are used for the construction of the PersistencySvc instance. If not provided, default values are assumed (see related documentation).

```
ObjectDeletePolicy                                          myDeletePolicy;
myDeletePolicy.setOnCache(true);    //    delete    on    the    cache
myDeletePolicy.setOnRef(true);     //   delete   on   the   'free'   ref
ctx.setObjectDeletePolicy(myDeletePolicy);   //   default   is   DO_DELETE
pool::IDataSvc*     myDatasvc     =     pool::DataSvcFactory::create(ctx);
```

- - Specifying only the IFileCatalog instance:

```
//              IFileCatalog*              myFileCatalog...
pool::IDataSvc*  myDataSvc  =  pool::DataSvcFactory::create(myFileCatalog);
```

  For the other parameters of the POOL environment are used default values: no shape transformation, no token re-construction, DELETE object delete policy.

- - Specifying the IPersistencySvc instance:

```
//         IPersistencySvc*         myPersistencySvc         =...
```

  The IPersistencySvc instance defining the whole POOL environment. Object delete policy as default.

```
pool::IDataSvc* myDatasvc = pool::DataSvcFactory::create(myPersistencySvc);
```

4.3.2. Operation on the storage system with Ref:

- -UPDATE - write:

```
pool::IDataSvc*  dataSvc  =  ...;  //  using  pool::DataSvcFactory(...)

MyClass* ptr = new MyClass; //user-create object, registered in the Dictionary
```

Prepare placement hint

```
pool::Placement                              place;
place.setDatabase(dbName,pool::DatabaseSpecification::PFN             );
place.setContainerName(contName);
place.setTechnology(pool::ROOTKEY_StorageType);
```

Instantiate Ref with the previous DataSvc instance

```
pool::Ref<MyClass>                                        ref(dataSvc,ptr);
```

Start transaction in UPDATE mode, mark the object for writing and commit changes

```
dataSvc->transaction().start(pool::ITransaction::UPDATE);
ref.markWrite(place);
dataSvc->transaction().commit();
```

- -UPDATE - update:

```
pool::IDataSvc*  dataSvc  =  ...;  //  using  pool::DataSvcFactory(...)
```

A Ref is obtained from a READ operation on the storage system:

```
pool::Ref<MyClass>                ref                =                ...
```

Start transaction in UPDATE mode, modify the object, mark for update and commit changes

```
dataSvc->transaction().start(pool::ITransaction::UPDATE);
ref->myMethod();  //  call  a  non  const  method  changing  object  state
ref.markUpdate();
dataSvc->transaction().commit();
```

- -UPDATE - delete:

```
pool::IDataSvc*  dataSvc  =  ...;//  using  pool::DataSvcFactory(...)
```

A Ref is obtained from a READ operation on the storage system:

```
pool::Ref<MyClass>                ref                =                ...
```

Start transaction in UPDATE mode, mark for delete and commit changes

```
dataSvc->transaction().start(pool::ITransaction::UPDATE);
ref.markDelete();
dataSvc->transaction().commit();
```

- -READ:

```
pool::IDataSvc*  dataSvc  =  ...;//  using  pool::DataSvcFactory(...)
```

A READ operation always requires a Token to be specified. This object is internally exchanged in the Storage Manager, and should be never constructed explicitly.

```
pool::Token*     myToken     =     ...;     //     the     Token

//    instantiate    Ref    with    the    previous    DataSvc    instance
pool::Ref<MyClass>                                  ref(dataSvc,*myToken);
```

Start transaction in READ mode, access the object to load it and close the READ-ONLY transaction

```
dataSvc->transaction().start(pool::ITransaction::READ);
ref->myMethod();
dataSvc->transaction().commit();
```

In all the examples, the operation within the transaction boundaries may be executed in a nested scope (or more), remaining valid regardless to the chosen object delete policy:

```
// the transaction is started outside the scope where Ref instance exists
dataSvc->transaction().start(pool::ITransaction::UPDATE);
{
                     pool::Ref<MyClass>                    ref(dataSvc,ptr);
                           ref.markWrite(place);
}
// the object is written even if the Ref instance has been destructed already!
dataSvc->transaction()->commit();
```

4.3.3. Object association with Ref:

- -SINGLE (implicit) cache:

  Define the class for the embedding object

```
struct                    MyObject                    {
                          pool::Ref<MyRelated>                          rel;
};
```

  Declare embedding object and referenced object

```
MyObject*            obj            =            new            MyObject;
MyRelated*           relObj           =          new           MyRelated;
```

  Define and set up the placement hints for the two objects

```
pool::Placement                          placeObj;
pool::Placement                          placeRel;
```

  Construct the Ref instances for the two objects

```
pool::Ref<MyObject>                                    refObj(dataSvc,obj);
pool::Ref<MyRelated>                                 refRel(dataSvc,relObj);
```

Start transaction in UPDATE mode, mark the two objects for writing, set the association between the two objects and commit

```
dataSvc->transaction().start(pool::ITransaction::UPDATE);
refObj.markWrite(placeObj);
refRel.markWrite(placeRel);
refObj->rel = relObj; // equivalent to refObj->rel = refRel.ptr();
dataSvc->transaction().commit();
```

Start transaction in READ mode to retrieve back the objects, access both objects and commit.

```
dataSvc->transaction().start(pool::ITransaction::READ);
pool::Ref<MyRelated> refRelNew = refObj->rel; // this will load MyObject class
refRelNew->myMethod();    //    this    will    load    MyRelated    class
dataSvc->transaction().commit();
```

In both the write and the read operations, the Storage Manager assumes that the MyRelated object embedded in MyObject is associated to an object in the same cache (dataSvc instance).

- -MULTI (explicit) cache:

Construct static Info object to associate a specific cache to a defined class

```
struct                                CCInfo                                {
               static               pool::IDataSvc*               ccCache;
};
```

Extend Ref class in order to use the DataSvc from the corresponding Info object

```
template <class T> class CCRef : public pool::Ref<T>, virtual public CCInfo {
                                                                      public:
            //            empty           (default)           constructor
            CCRef():            pool::Ref<T>(CCInfo::ccCache)           {}
                     //                    copy                   constructor
      CCRef(const      CCRef<T>&      aCCRef):      pool::Ref<T>(aCCRef){}
                 //                 assignment                  operators
        CCRef<T>&        operator=(const        CCRef<T>&        aCCRef){
                                      pool::Ref<T>::operator=(aCCRef);
                                return                                *this;
                                                                      }
        CCRef<T>&       operator=(const       pool::Ref<T>&       aRef){
                                        pool::Ref<T>::operator=(aRef);
                                return                                *this;
                                                                      }
};
```

Define embedding object, instantiate objects

```
struct                    MyObject                    {
     //    association    defined    through    the    extended    Ref
                         pool::CCRef<MyRelated>                        rel;
};

MyObject*            obj            =            new            MyObject;
MyRelated*           relObj           =           new           MyRelated;
```

Prepare placement hints

```
pool::Placement                                                placeObj(...);
pool::Placement                                                placeRel(...);
```

Set the cache for embedded object and instantiate the Refs

```
pool::CCInfo::ccCache                        =                        dataSvc2;
//        here        both        can        be        'normal'        Ref!
pool::Ref<MyObject>                                    refObj(dataSvc1,obj);
pool::Ref<MyRelated>                                refRel(dataSvc2,relObj);
```

start transaction in UPDATE mode, mark the two objects for write, set the association and commit

```
dataSvc->transaction().start(pool::ITransaction::UPDATE);
refObj.markWrite(placeObj);
refRel.markWrite(placeRel);
refObj->rel = refRel; // the ref instance is 'deeply' copied (the cache is propagated)
dataSvc->transaction().commit();
```

Start READ transaction to retrieve the objects, access the top level object and the embedded object, close the transaction.

```
dataSvc->transaction().start(pool::ITransaction::READ);
pool::Ref<MyRelated> refRelNew = refObj->rel; // this will load MyObject class in dataSvc
refRelNew->myMethod(); // this will load MyRelated class in dataSvc2!!
dataSvc->transaction().commit();
```

The wrapped CCRef class allows to set at construction time the cache specific for the objects of this class, stored in a static variable.

4.3.4. Ownership handling

- -Selection of the policy:

  DataSvc specific selection:

  Set up POOL environment

```
pool::DataSvcContext                                                ctx;
ctx.setFileCatalog(myFileCatalog);
//                                                                   ...
```

  Set DONOT_DELETE policy

```
ctx.setObjectDeletePolicy(pool::ObjectDeletePolicy::DONOT_DELETE); //for the non-deleting
pool::IDataSvc*     myDatasvc     =     pool::DataSvcFactory::create(ctx);
```

  Global selection:

  The GLOBAL_DEFAULT policy is applied to all the Ref and DataSvc instances.

```
pool::ObjectDeletePolicy::GLOBAL_DEFAULT                =
                                  pool::ObjectDeletePolicy::DONOT_DELETE;
```

- -Garbage collection with DO_DELETE

- Case 1 (simple out-of-scope):

  Declaration of user-defined class

```
MyClass*            ptr            =            new            MyClass;
{
         //       ref       declared       in       nested       scope
                   pool::Ref<MyClass>                 ref(dataSvc,ptr);
}              //              ptr             is             deleted!
```

- Case 2 (marked for write/update/delete):

  Declaration of user-defined class

```
MyClass*            ptr            =            new            MyClass;
```

  Start transaction in UPDATE mode

```
dataSvc->transaction()->start(pool::ITransaction::UPDATE);
{
         //       ref       declared       in       nested       scope
                   pool::Ref<MyClass>                 ref(dataSvc,ptr);
   ref.markWrite(place); // without this call ptr will be deleted
} // ptr is not deleted yet! (as with markUpdate, and markDelete)
dataSvc->transaction()->commit();
//             now             ptr             is             deleted!
```

- Case 3 (ownership shared among caches):

  Declaration of user-defined class, used in two caches

```
MyClass*            ptr            =            new            MyClass;
{
      //    ref    declared    in    nested    scope    in    cache    1
                  pool::Ref<MyClass>                ref1(dataSvc1,ptr);
                                                                       {
          //     inner     nested     scope,     ptr     in     cache     2
                     pool::Ref<MyClass>                ref(dataSvc2,ptr);
           }       //       ptr       is       not       deleted!
}          //           now          it          is          deleted!
```

5. DataSvc component: Reference Manual

5.1. class Ref<T>

5.1.1. Introduction

The ref class template is a wrapper around a generic object pointer, providing persistency capability.

The ref class meets the CopyConstructible and Assignable requirements of the C++ Standard Library, and so can be used in standard library containers.

The ref class handles pointer ownership following a user-defined policy, but meets the general requirements of actors in a shared-ownership environment (see chapter on ownership policy).

The class template is parameterized on T, the type of the object pointed to. ref and its member functions place few requirements on T; it must be a complete type, and non-void. Member functions that do place additional requirements (...) are explicitly documented below.

ref<T> can be implicitly converted to ref<U> whenever T* can be implicitly converted to U*. In particular, refwith <T> is implicitly convertible to ref<const T>, to ref<U> where U is an accessible base of T (see chapter on polymorphic behavior).

5.1.2. Members

- **Constructors**

```
ref();                    //                    never                    throws
```

Effects: Constructs an empty, cache-unbound ref. The default ownership policy is assumed.

Postconditions: ptr() == 0.

Throws: nothing.

```
explicit      Ref(DeletePolicy      policy);      //      never      throws
```

Effects: Constructs an empty, cache-unbound ref. The specified delete policy will be used to handle pointer ownership.

Postconditions: ptr() == 0.

Throws: nothing.

```
explicit    Ref(IDataSvc*    dataSvc);    //    deprecate.    Never    throws
explicit      Ref(IDataSvc&      dataSvc);      //      never      throws
```

Effects: Constructs an empty ref bound to the specified dataSvc object. The default ownership policy is assumed.

Postconditions: ptr() == 0.

Throws: nothing.

```
explicit      Ref(ICacheSvc&      cacheSvc);      //      never      throws
```

Effects: Constructs an empty ref bound to the specified cacheSvc object. The default ownership policy is assumed.

Postconditions: ptr() == 0.

Throws: nothing.

```
Ref(IDataSvc*   dataSvc,   T*   obj);   //   deprecated.   Never   throws
Ref(IDataSvc&     dataSvc,     T*     obj);     //     never     throws
```

Requirements:T must be a complete type.

Effects: Constructs a ref that stores in the specified dataSvc cache the pointer obj. The registration in the cache calls the method ICacheSvc::registerObject. If dataSvc is a null pointer, the T* pointer is stored locally in the ref instance.

Postconditions: ptr() == obj. If the dataSvc pointer is not null, the ref instance references a specific entry of the cache, containing the object pointer obj. The cache entry involved is either re-used

(if the pointer has been registered already), or created (if the pointer was not in the cache before).

Throws: nothing.

```
Ref(ICacheSvc&  cacheSvc,  T*  obj);  //  never  throws
```

Requirements:T must be a complete type.

Effects: Constructs a ref that stores in the specified cacheSvc cache the pointer obj. The registration in the cache calls the method ICacheSvc::registerObject. If cacheSvc is a null pointer, the T* pointer is stored locally in the ref instance.

Postconditions: ptr() == obj. If the dataSvc pointer is not null, the ref instance references a specific entry of the cache, containing the object pointer obj. The cache entry involved is either re-used (if the pointer has been registered already), or created (if the pointer was not in the cache before).

Throws: nothing.

```
Ref(IDataSvc* dataSvc, const Token& token); // deprecated. Never throws
Ref(IDataSvc&  dataSvc,  const  Token&  token);  //  never  throws
```

Effects: Constructs a ref pointing to an object identified in the storage system by the token instance. If the dataSvc pointer is null, the instance is virtually equivalent to an empty, cache-unbound ref. Otherwise, the ref instance references a specific entry of the cache, containing the token pointer. The cache entry involved is either re-used (if the token value has been registered already), or created (if the token value was not in the cache before). If the token instance is valid, the object is created in a second time, as soon as the object pointer has to be accessed through the ref instance. The object pointer is stored in the same cache entry hosting the token pointer.

Postconditions: ptr() returns a pointer to a transient instance of the object stored in the database with the specified token identifier. A null pointer is returned if the token is invalid.

Throws: nothing.

```
Ref(ICacheSvc&  cacheSvc,  const  Token&  token);  //  never  throws
```

Effects: Constructs a ref pointing to an object identified in the storage system by the token instance. If the cacheSvc pointer is null, the instance is virtually equivalent to an empty, cache-unbound ref. Otherwise, the ref instance references a specific entry of the cache, containing the token pointer. The cache entry involved is either re-used (if the token value has been registered already), or created (if the token value was not in the cache before). If the token instance is valid, the object is created in a second time, as soon as the object pointer has to be accessed through the ref instance. The object pointer is stored in the same cache entry hosting the token pointer.

Postconditions: ptr() returns a pointer to a transient instance of the object stored in the database with the specified token identifier. A null pointer is returned if the token is invalid.

Throws: nothing.

```
Ref(IDataSvc* dataSvc, const std::string& tokenString); // deprecated. Never throws
Ref(IDataSvc& dataSvc, const std::string& tokenString); // never throws
```

Effects: Constructs a ref pointing to an object identified in the storage system by the tokenString instance. If the dataSvc pointer is null, the instance is virtually equivalent to an empty, cache-unbound ref. Otherwise, the ref instance references a specific entry of the cache, containing the token pointer. The cache entry involved is either re-used (if the tokenvalue has been registered already), or created (if the tokenvalue was not in the cache before). If the token instance is valid, the object is created in a second time, as soon as the object pointer has to be accessed through the

ref instance. The object pointer is stored in the same cache entry hosting the token pointer.

Postconditions: ptr() returns a pointer to a transient instance of the object stored in the database with the specified tokenString identifier. A null pointer is returned if the tokenString is invalid.

Throws: nothing.

```
Ref(ICacheSvc& cacheSvc, const std::string& tokenString); // never throws
```

Effects: Constructs a ref pointing to an object identified in the storage system by the tokenString instance. If the dataSvc pointer is null, the instance is virtually equivalent to an empty, cache-unbound ref. Otherwise, the ref instance references a specific entry of the cache, containing the token pointer. The cache entry involved is either re-used (if the tokenvalue has been registered already), or created (if the token value was not in the cache before). If the token instance is valid, the object is created in a second time, as soon as the object pointer has to be accessed through the ref instance. The object pointer is stored in the same cache entry hosting the token pointer.

Postconditions: ptr() returns a pointer to a transient instance of the object stored in the database with the specified tokenString identifier. A null pointer is returned if the tokenString is invalid.

Throws: nothing.

```
Ref(const Refwith <T>& r); // never throws
```

Effects: Constructs a ref instance with the same features of the specified r: the same cache (if any) will be used, the same cache entry (if any) will be shared, increasing the related reference counting. See details about cache entry sharing.

Postconditions: ptr()==r.ptr()

Throws: nothing.

```
template <class C> Ref(const Ref<C>& r); // never throws
```

Requirements: C must be a complete type. T must be a public base of C - otherwise a compile-time error will be reported.

Effects: Constructs a ref instance with the same features of the specified r: the same cache (if any) will be used, the same cache entry (if any) will be shared, increasing the related reference counting. See details about cache entry sharing.

Postconditions: ptr() == static_cast_cast<T*>(r.ptr())

Throws: nothing.

- **destructor**

```
~ref(); // never throws
```

Effects: If *this is cache-unbound the embedded SharedAnyPtr instance (sharing the ownership of the object pointer) is destroyed. If no other SharedAnyPtr instances have been created around the same object pointer, the specified delete policy is applied. Otherwise, if *this holds a pointer to a cache, the reference to the cache entry (if any) is released. This might trigger the deletion of the cache entry, if no other ref are using it. The cache entry deletion might in turns imply the deletion of the object and the token associated.

Throws: nothing.

pointer assignment

```
Refwith <T>& operator=(T* p);
```

Effects: If *this is cache-unbound the pointer p is assigned to the embedded SharedAnyPtr instance. The reference to the previously stored pointer is released. Otherwise, if *this holds a pointer to a cache, the reference to the cache entry (if any) is released. This might trigger the deletion of the cache entry, if no other ref are using it. The cache entry deletion might in turns imply the deletion of the object and the token associated.

Postconditions: ptr() == p.

Returns: *this.

- **assignment**

```
Refwith <T>& operator=(const Refwith <T>& ref);
```

Effects: A default copy operation (link) with no type checking is performed.

Postconditions: ptr() == ref.ptr()

Returns: *this.

- **extended assignment**

```
template <class C> Refwith <T>& operator=(const Ref<C>& ref);
```

Requirements: C must be a complete type. T must be a public base of C - otherwise a compile-time error will be reported.

Effects: A default copy operation (link) with no type checking is performed. Type checking is ensured at compile time.

Postconditions: ptr() == static_cast<T*>(ref.ptr()).

Returns: *this.

- **ref<C> dynamic casting**

```
template <class C> Refwith <T>& castDynamic(const Ref<C>& ref);
```

Requirements: C must be a complete type.

Effects: A default copy operation (link) with type checking is performed. Type compatibility is verified at run time.

Postconditions: ptr() == dynamic_cast<T*>(ref.ptr()).

Returns: *this.

- Indirection

```
T* operator->() const;
```

Effects: In the case of cache-bound ref, the load-on-demand mechanism is activated.

Postconditions: isOpen() == true.

Returns: the pointer associated, stored locally (for cache-unbound refs) or in the related cache entry.

Throws: RefException when the pointer returned is null.

```
T&                              operator*()                              const;
```

Effects: In the case of cache-bound ref, the load-on-demand mechanism is activated.

Postconditions: isOpen() == true.

Returns: a reference to the object pointed to by the stored pointer.

Throws: RefException when the pointer stored is null.

- **conversion**

```
operator                    T*                    ()                    const;
```

Effects: In the case of cache-bound ref, the load-on-demand mechanism is activated.

Postconditions: isOpen() == true.

Returns: the pointer associated, stored locally (for cache-unbound refs) or in the related cache entry.

Throws: nothing.

- **ptr**

```
T*                                ptr()                                const;
```

Effects: In the case of cache-bound ref, the load-on-demand mechanism is activated.

Postconditions: isOpen() == true.

Returns: the pointer associated, stored locally (for cache-unbound refs) or in the related cache entry.

Throws: nothing.

- **equality operators**

```
template                                <class                                  C>
bool         operator==(const          Ref<C>&         aRef)         const;
```

Returns: a bool value resulting from the call RefBase::equals(aRef). (link).

```
template                                <class                                  C>
bool         operator!=(const          Ref<C>&         aRef)         const;
```

Returns: a bool value resulting from !operator==(aRef)

5.1.3. Free Functions

- **comparison**

```
inline friend bool operator==(const Ref& aRef, T* aPtr);
inline friend bool operator==(T* aPtr, const Ref& aRef);
```

Returns: aRef.ptr() == aPtr.

Throws: nothing.

```
inline friend bool operator!=(const Ref& aRef, T* aPtr);
inline friend bool operator!=(T* aPtr, const Ref& aRef);
```

Returns: !(aRef == aPtr).

Throws: nothing.

5.1.4. RefBase functions

- **default copy**

```
void copy(const RefBase& aRef);
```

Effects: A specific copy policy is applied:

- if aRef is bound to a cache, *this will be bound to the same cache and cache entry (deep copy)

- else if aRef is cache-unbound, the cache of *this (if any) will be kept. The object pointer associated to aRef (if any), will be registered in the cache of *this (shallow copy).

In both case, the compatibility between *this embedded type and the type of the object pointer is verified. If the types are unrelated, *this will only copy the cache pointer of aRef (when available) but will be not associated to any object pointer.

PostCondition: (*this == aRef)

Throws: nothing

- **deep copy**

```
void copyDeep(const RefBase& aRef);
```

Effects: The two cache-related parameters (cache and cache entry references) of aRef are copied into *this. The compatibility between *this embedded type and the type of the object pointer related to aRef is verified. If the types are unrelated, *this will only copy the cache pointer of aRef (when available) but will be not associated to any object pointer.

PostCondition: (*this == aRef); cacheSvc()==aRef.cacheSvc();

Throws: nothing.

- **shallow copy**

```
void copyShallow(const RefBase& aRef);
```

Effects: The cache reference in *this is not modified. The object pointer associated to aRef is assigned to *this. The compatibility between *this embedded type and the type of the object pointer related to aRef is verified. If the types are unrelated, *this will be not associated to any object pointer.

PostCondition: (*this == aRef);

Throws: nothing.

- **isNull**

```
bool                            isNull()                            const;
```

Effects: If *this is connected to a cache, this call may trigger the object loading from the storage syste

m. **Returns**: ptr() == 0;

PostCondition: isOpen() == true;

Throws: nothing.

- **isOpen**

```
bool                            isOpen()                            const;
```

Effects: Verifies if a transient object corresponding to *this has been loaded in memory by reading from the storage system. Always true for cache-unbound instances.

Throws: nothing.

- **isConnected**

```
bool                          isConnected()                         const;
```

Returns: cacheSvc()!=0;

Throws: nothing.

- **setCacheSvc**

```
void          setCacheSvc(ICacheSvc*          aCacheSvc)          const;
```

Effects: Connect the ref instance to the specified cache service object. If *this is already connected to a cache, the old conncetion is previously closed (including a reference to any cache entry) If *this is not connected to any cache, the connection to the specified cache will cause the registering of the locally-stored pointer (if any) in this cache. As a conseguence, the local pointer storage wil be empty. If *this is connected to the same cache svc specified in the call, no action is taken.

PostCondition: cacheSvc() == aCacheSvc;

Throws: nothing.

- **cacheSvc**

```
ICacheSvc*                      cacheSvc()                          const;
```

Returns: the pointer to the ICacheSvc object the instance is connected to. 0 if no connection is available.

Throws: nothing

- **cacheEntry**

```
const CacheEntry* cacheEntry() const;
```

Returns: the pointer to the CacheEntry object referenced. 0 if no CacheEntry is referenced.

Throws: nothing.

- **setToken**

```
void setToken(const Token& aToken);
```

Effects: Associate the ref instance to the specified token object. If *this is not connected to a cache, no action is taken. If *this is already connected to a cache, any reference to a cache entry is previously dropped. The new cacheentry referenced will be the result of the specified token registration. No action is taken if old and new CacheEntry are the same.

PostCondition: token() == aToken;

Throws: nothing.

- **token**

```
const Token* token() const;
```

Returns: the pointer to the Token object associated. 0 if no CacheEntry, and so no Token is referenced.

Throws: nothing.

- **toString**

```
const std::string toString() const;
```

Returns: Given: Token* t = token(); the method returns t->toString() if t>0, otherwise an empty string ("").

Throws: nothing.

- **placement**

```
Placement placement(); // never throws
```

Returns: Given: Token* t = token(); when t>0, the method returns a Placement instance with the values described by t; otherwise an empty Placement object..

Throws: nothing.

- **object**

```
const SharedAnyPtr& object() const;
```

Returns: the object pointer referenced, encapsulated in a SharedAnyPtr object.

Effects: the object is read from the database if it is not cached when the function is called.

PostCondition: isOpen() == true;

Throws: nothing.

- **register for write, update and delete**

```
bool markWrite(const Placement& place) const;
```

Effects: - Free ref (no cache svc, no cache entry): no effect.

- Cache bound ref: if the associated cache entry does not contain a token, a token instance is assigned to it according to the placement parameters. Otherwise, and in particular if the associated cache entry is already marked, the call has no effect. The cache entry, when marked, is kept in the cache until the commit time, regardeless to the lifetime of the ref.

Returns: -Free ref: always false.

- Cache bound ref:true if already marked for write, or previously not associated to a token. False if already associated to a token, and not already marked for write.

PostCondition: isOpen() == true;

Throws: CacheSvcException if the Class describing the object is not represented in the dictionary.

```
bool markMultiwrite(const Placement& place) const;
```

Effects: - Free ref (no cache svc, no cache entry): no effect.

- Cache bound ref: if no token is associated to the associated cache entry, the effect is the same as markWrite. Otherwise, a new cache entry is associated to the ref, containing a new token instance created according to the placement parameters. The old cache entry reference, when present, is overwritten. The cache entries, when marked, are kept in the cache until the commit time, regardeless to their reference counting. **Returns**: -Free ref: always false.

- Cache bound ref: always true.

PostCondition: isOpen() == true;

Throws: CacheSvcException if the Class describing the object is not represented in the dictionary.

```
bool markUpdate() const;
```

Effects: - Free ref (no cache svc, no cache entry): no effect.

- Cache bound ref: if no token is assigned to the ref, the call has no effect. If a token is assigned and the associated cache entry is already marked for write, update or delete, the call has no effect. Otherwise, the entry in the storage system corresponding to the assigned token is updated according to the present object content.

Returns: -Free ref: always false.

- Cache bound ref: false if no token is assigned, or the associated cache entry is already marked for write or delete. True otherwise.

PostCondition: isOpen() == true;

Throws: CacheSvcException if the Class describing the object is not represented in the dictionary.

```
bool                markDelete()                const;
```

Effects: - Free ref (no cache svc, no cache entry): no effect.

- Cache bound ref: if no token is assigned to the ref, the call has no effect. If a token is assigned and the associated cache entry is already marked for write, update or delete, the call has no effect. Otherwise, the entry in the storage system corresponding to the assigned token is deleted..

Returns: -Free ref: always false.

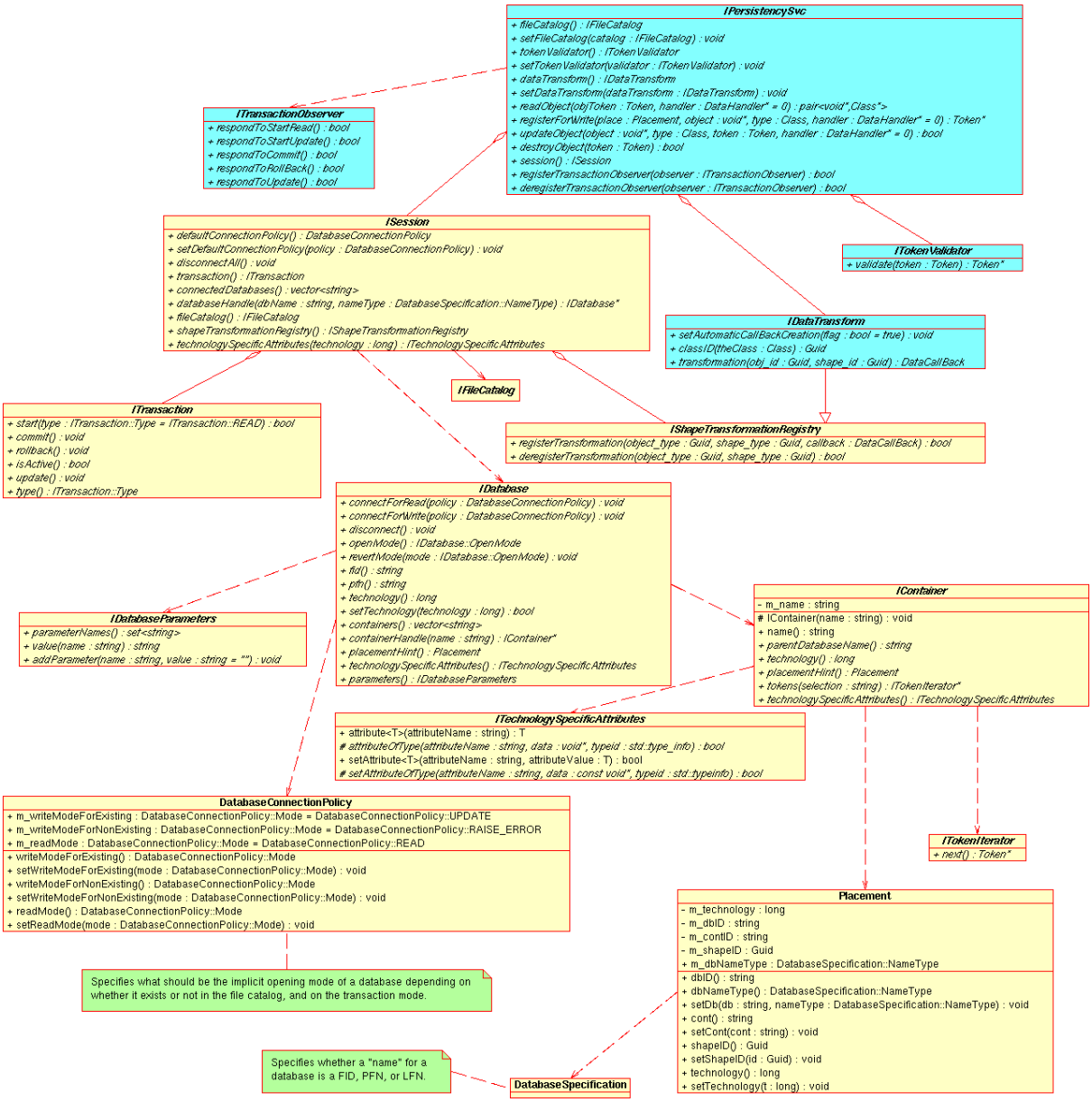
- Cache bound ref: false if no token is assigned, or the associated cache entry is already marked for write or update. True otherwise.

Throws: nothing.

6. PersistencySvc component: User level semantics

6.1. Public classes UML diagram

The public interfaces of the component are depicted in the following UML class diagram. In this diagram, with cyan-filled boxes are represented the classes that are not expected to be accessed by a user who uses the POOL framework with the DataSvc package. These are the *developer level* interfaces that are used either by other POOL packages (like the DataSvc), or by the core framework packages of the experimental software. In the classes the full operation signature appears as well.



UML class diagram of the PersistencySvc public interfaces

6.2. User and Developer level top level interfaces

The top level abstract interface in the PersistencySvc package is *IPersistencySvc*. It provides all the methods for technology-independent object I/O, making use only of the dictionary information for a given object.

An IPersistencySvc object is always created in association to an IFileCatalog object. Moreover, it is owned, managed and used directly only by services providing cache and object reference management, such as the POOL DataSvc package. It is therefore considered to be a developer level interface, since users never interact with it directly.

The public top level interface which is exposed to the user is *ISession* which provides access to the *ITransaction* object, the policy for the implicit accessing of databases, and methods for explicitly opening or closing them. In a typical POOL application the ISession instance (one per IPersistencySvc) is retrieved from the DataSvc object.

6.3. The scheme of global transactions

In POOL every I/O operation is done within a transactional context. There is one context per PersistencySvc instance, and therefore one per DataSvc instance.

The transactional context is controlled with calls to the ITransaction interface. The latter allows the

user to start, commit or roll back a transaction. Transactions can be started either in *UPDATE* or in *READ* mode depending whether at least one object will be ever written, updated or deleted within the active transaction.

A database file is explicitly or implicitly opened by the PersistencySvc through the appropiate calls to the underlying *IStorageSvc* instances (there is one such instance per major technology type). For every connected database file there is a *micro-transaction* defined in the StorageSvc package. Every object I/O operation is performed within such a micro-transaction. The PersistencySvc global/user transaction, which is controlled by the ITransaction interface, has the responsibility of dispatching the relevant start/commit/rollback calls to all the micro-transactions of all the connected database files.

Any client of the IPersistencySvc object implementing the *ITransactionObserver* interface may subscribe itself to follow the transactional operations. Such a client is a class from the DataSvc package which makes sure that the object cache is cleared whenever a transaction is committed.

6.4. Implicit and explicit opening of databases

6.4.1. Implicit connections, placement hints and implicit connection policies

An IPersistencySvc object keeps track of the databases which are open at any given moment, for every technology in use. Whenever needed, a new or existing database is opened for reading or writing automatically. This may happen when:

- The physical location where objects are written is determined by the *Placement* class which is used to specify the following:

- *DatabaseSpecification*

- During a READ transaction all databases are opened in read mode. The check whether a database exists or not the system queries the FileCatalog. In case of an UPDATE transaction, there are three cases where the system needs to know how it should behave:

-
- It is possible to define a different global behaviour by constructing a *DatabaseConnectionPolicy* object and setting its relevant fields accordingly. The global implicit connection policy is changed whenever the ISession is passed a new DatabaseConnectionPolicy object.

6.4.2. Explicit connections

Databases can be accessed also explicitly. The ISession interfaces may act as a factory for *IDatabase* objects specifying the name of the database and the corresponding DatabaseSpecification, similarly to the case of Placement objects. If the given database is not already connected the user may explicitly connect it for reading or writing. The user may do that using either the current global DatabaseConnectionPolicy, or specifying a new one that will be valid only for this particular database.

6.5. Exploring databases and containers

If a user obtains an IDatabase object and explicitly connects to the corresponding database, it is possible to retrieve its technology independent attributes such as PFN, FID, technology identifier and the list of the names of its containers. Technology specific attributes can be obtained by name using the *ITechnologySpecificAttributes* interface returned by the IDatabase, while annotation-like parameters can be set or retrieved through the *IDatabaseParameters* interface.

The IDatabase interface acts also as a factory class for *IContainer* objects specifying the container name. An IContainer object may be used to start an iteration over the tokens of the objects that it holds using an *ITokenIterator* object. These low level interfaces are the implementation basis of the ImplicitCollection package.

The IDatabase and IContainer interfaces expose all the necessary information like system names and technology identifiers to allow the backwards navigation in the hierarchy Technology Domain / Database / Container. At any given level of this hierarchy it is possible to set and retrieve technology specific attributes via calls to the *ITechnologySpecificAttributes* interface. The latter can be retrieved from an IContainer, an IDatabase and an ISession (specifying the technology identifier) object.

6.6. Customizable operations

6.6.1. Customized streamers

The public interfaces of PersistencySvc allow the customization of the streaming behaviour of the system when reading on writing objects. Very often the need appears that the representation of an object in the persistent world is different from the one in the transient world. A typical case is when there are some data members of a class, that are used only for bookkeeping purposes or they have a temporary functional scope (like a boolean flag indicating whether a class has been initialized or not with a call to an initialize() method). It is possible to instruct the system not to store such variables by declaring them as *transient* when creating their dictionary.

In more complicated use cases the layout of the persistent class is so different from the corresponding transient class, where simply declaring some data members as transient does not suffice. This is the case for example where a data member is of a different type or has a different name in the two representations. In order to achieve this the following steps have to be followed:

1.
2. *DataCallback*

3. *ShapeTransformationRegistryIDataTransform*
When storing an object the user instructs the system to use a particular persistent shape by declaring its class identifier to the corresponding field in the Placement object which is used to steer the writing. The *Token* of the persistent object holds this class identifier and it is used to trigger the customized conversion when reading it back into an object of a different transient type.

6.6.2. Token validators

It is possible that an object is written into the system at some moment in time and then it has been relocated to a different contrainer/database/technology. In POOL references are always unidirectional, and therefore reference integrity is not garranteed by the system is a user decides to relocate an object which may be referenced by other objects. In order to make sure that old references are redirected to the correct object, it is possible to install an implementation of the *ITokenValidator* interface into the system via the IPersistencySvc interface. This interface as a factory for new tokens given an existing one. The ITokenValidator in use is always invoked to validate a token during each read operation. The default token validator of the system returns always a token identical to the input.

6.7. Example of usage

Creating an IPersistencySvc object (done by DataSvc)

```
pool::IPersistencySvcFactory* psfactory = pool::IPersistencySvcFactory::get();
std::auto_ptr< pool::IPersistencySvc > persistencySvc( psfactory->create( "PersistencyS
                                                                          aFileCatalogP
```

Starting and committing transactions

```
pool::ISession& session = (from the DataSvc or an IPersistencySvc object)
    pool::ITransaction&     transaction     =     session.transaction();
        transaction.start(          pool::ITransaction::UPDATE          );
                                                                        .
                  some              write/update              operations
                                                                        .
                                              transaction.commitAndHold();
                                                                        .
                  more              write/update              operations
                                                                        .
                                                        transaction.commit();
```

Explicit opening for reading of a database file specifying the PFN, and extracting its containers

```
pool::ISession& session = (from the DataSvc or an IPersistencySvc object)
    pool::ITransaction&     transaction     =     session.transaction();
         transaction.start(          pool::ITransaction::READ          );
std::auto_ptr< pool::IDatabase > db( session.databaseHandle( "myFile.pool",
                                                             pool::DatabaseSpecif
                                                      db->connectForRead();
  std::vector<    std::string    >    containers    =    db->containers();
                                                      transaction.commit();
```

Retrieving the FID and the file size of a database

```
pool::ISession& session = (from the DataSvc or an IPersistencySvc object)
    pool::ITransaction&     transaction     =     session.transaction();
if ( ! transaction.isActive() ) transaction.start( pool::ITransaction::READ );
std::auto_ptr< pool::IDatabase > db( session.databaseHandle( "myFile.pool",
                                                             pool::DatabaseSpecif
 if  (db->openMode()  ==  pool::IDatabase::CLOSED  )  db->connectForRead();
       const        std::string&        fid        =        db->fid();
unsigned int sizeInkB = db->technologySpecificAttributes().attribute<int>( "FILE_
                                                      transaction.commit();
```

Further examples of usage can be considered:

•

All the above exist in the POOL CVS repository and appear in every POOL release

7. PersistencySvc component: Reference Manual

7.1. Signatures of public interfaces

7.1.1. User-level interfaces

- *ISession* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1ISession.html
- *ITransaction* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1ITransaction.html
- *IShapeTransformationRegistry* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IShapeTransformationRegistry.html
- *IDatabase* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IDatabase.html [http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IIDatabase.html]
- *IDatabaseParameters* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IDatabaseParameters.ht

ml

- *DatabaseConnectionPolicy* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1DatabaseConnectionPolicy.html
- *IContainer* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IContainer.html
- *ITechnologySpecificAttributes* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1ITechnologySpecificAttributes.html
- *Placement* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1Placement.html
- *ITokenIterator* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1ITokenIterator.html

7.1.2. Developer-level interfaces

- *IPersistencySvc* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IPersistencySvc.html
- *ITransactionObserver* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1ITransactionObserver.html
- *IDataTransform* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IDataTransform.html
- *ITokenValidator* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1ITokenValidator.html

7.2. Command line tools

`pool_insertFileToCatalog [-c fileCatalog] [-t technologyType] file1 file2 ...`
Opens the database files specified (PFN values), retrieves their File ID, and inserts them in the file catalog. If the technology type is not specified, it is assumed that the database files have the ROOT format.

NOTE: This tool is only meant to be used by experienced users for development and debuging purposes. **For the purposes of production data handling, one should resort to the consistent usage of the relevant FileCatalog tools**.

8. AttributeList component: User level semantics

8.1. Introduction

AttributeList specifies an API and does not provide command line tools. C++ users usually assume that attribute list object are already created by other pool components (file catalogs or collections). In the codelets below we assume the following convention:

```
// alist  - is an exsiting attribute list object (for example obtained from a file catalog)
AttributeList & alist = get_it_from_somewhere();
```

8.2. Setting values of AttributeList

Assume that attribute list has 3 columns A,B (int), X (string). Setting new values is as easy as:

```
                                                    alist["A"].setValue(10);
                                                    alist["B"].setValue(20);
                                        alist["X"].setValue(std::string("xxx"));
```

If you try to set a string to column B (which is of type int) then an exception pool::attribute_bad_type will be thrown:

```
                                                                      try
                                                                        {
                                         alist["B"].setValue(std::string("xxx");
                                                                        }
                              catch(pool::attribute_bad_type                       x)
                                                                        {
                        std::cerr           <<           "Type           mismatch";
                                                                        }
```

"Type mismatch" will be printed in this case.

8.3. Reading values of AttributeList

Reading back the values is equally easy. Following the previous example:

```
                                        int                                    A,B;
                                    std::string                                  X;

                                                     alist["A"].getValue(A);
                                                     alist["B"].getValue(B);
                                                     alist["X"].getValue(X);

    std::cout << "A = " << A << " B = " << B << " X = " << X;
```

will print something like:

```
A = 10 B = 20 X = xxx
```

Again an exception pool::attribute_bad_type is thrown if the types of attribute mismatches the variable in which you try to read the value.

8.4. Getting attribute type and name: AttributeListSpecification

So far so good. But how do we know what is attribute name and type? Every attribute has a specification which defines precisely this.

```
//           prints           "My            name           is           A".
std::cout     <<    "My    name    is    "    <<    alist["A"].spec().name();

//                                 returns                                "int"
alist["A"].spec().type_name()
```

String indicating the type ("int") is a cross-platform and cross-compiler name for the attribute type.

AttributeList has an AttributeListSpecification which contains specifications for all attributes in the list:

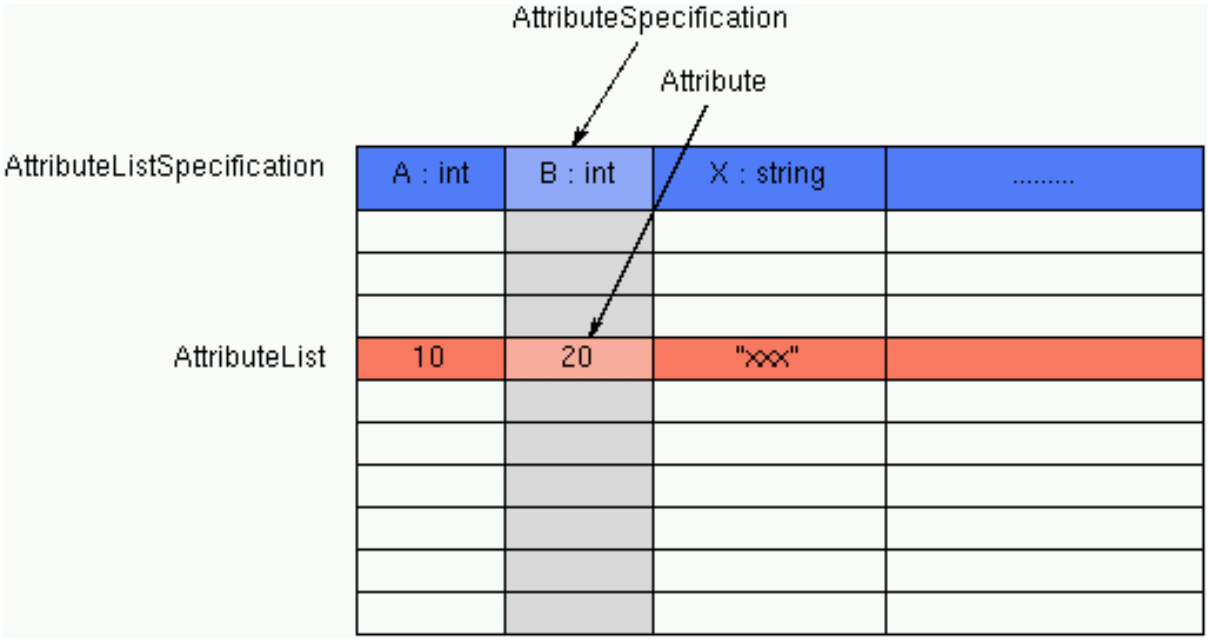
```
alist.attributeListSpecification()
```

AttributeListSpecification has methods to access individual specifications and iterators to loop over them.

8.5. The difference between AttributeList and AttributeListSpecific-

ation

You may think of AttributeList as a row in a table where Attribute objects correspond to individual cells. AttributeListSpecification is a description of columns in a table where each AttributeSpecification describes one column.



A view of AttributeList table created in the examples.

8.6. Iterating over the elements in AttributeList

AttributeList provides an iterator to iterate over individual attributes in the list. The following loop prints all attributes in the list:

```
for(pool::AttributeList::const_iterator     it     =     alist.begin();
                                           it    !=    alist.end();    ++it)
          //        *it        is        a        current        Attribute
                                              it->print(std::cout);
```

Important: order of attributes is specified by the AttributeListSpecification. If you have two AttributeLists A,B,C and A,C,B they will be iterated in different order, according to their specification. All this means that attributes are NOT sorted in the list.

8.7. Converting Attributes to strings and vice-versa

Attributes may be converted to and from string even if we do not know the real type of an attribute. This may be very useful for general applications when you do not know the attribute type statically and you need to discover it at runtime. Nevertheless you still want to set and get the value of such attributes. The following example prints the value and type of all attributes in the list:

```
for(pool::AttributeList::const_iterator     it     =     alist.begin();
                                           it    !=    alist.end();    ++it)
                                                                        {
          //        *it        is        a        current        Attribute
           std::string        vs        =        it->getValueAsString();
           std::string        ts        =        it->spec()->type_name();
    std::cout  <<  "value  =  "  <<  vs  <<  "type  =  "  <<  ts  <<  std::endl;
                                                                        }
```

You may also set the value of an attribute as a string:

```
                                          alist["A"].setValueAsString("10");
```

If the argument string cannot be converted to the type expected by the attribute , then the exception is

thrown (FIXME: not yet implemented).

8.8. Comparing AttributeList objects

You may compare two AttributeLists using the operator ==. AttributeLists are equal if they have the same number of columns with same type and name and their values are equal.

Important: Order of columns does not matter. A,B,C is equal to C,A,B.

Important: In the future we may drop operator == for explicit methods like isEqual. The reason is that several comparison strategies are possible and we may want to support many of them at the same time.

AttributeListSpecification is also compared with operator ==. All the remarks are the same.

8.9. Pitfals with setting attributes and implicit type conversion

Some of the methods, such as Attribute::setValue<T> are templates and therefore they may support almost any type. However this also means that implicit conversions are sometimes not performed on their arguments. For example, if you have a boolean Attribute, then

```
attribute.setValue(false)
```
works fine. But

```
attribute.setValue(1)
```
fails because 1 is of type int and you get the attribute_bad_type exception. Of course you may always do the conversion explicitly like this:

```
attribute.setValue(bool(1))
```
This is perfectly valid, because typically you know the static type of the attribute anyway. For a type-less way of setting attributes see the section "Converting Attributes to strings".

9. AttributeList component: Reference Manual

Refer to Doxygen and LXR pages available from http://pool.cern.ch

10. FileCatalog component: User level semantics

10.1. Public classes UML diagram

10.2. Composite Catalog concepts

Starting from POOL_1_6_0, composite catalog features are supported. The file catalog the user operates on consists of one master catalog which is read/writable and any number of read-only catalogs. One can specify the master catalog and add read-only catalogs using the contact strings.

The writing operations on the catalog are automatically performed on the master catalog; while the lookup operations lookup first in the master catalog and then the read-only ones in the order defined by the user. For bulk lookups, results found in all the leaf catalogs are returned. For lookup operations which expect a single result, the search stops when the first result is found and returned.

10.3. How to construct the catalog contact string

To obtain the connection to the catalog, a contact string of the format:

[prefix_][protocol]://[username]:[password]@[host]:[port]/[path]

or

[prefix_]file:path

The **[prefix_]** field is used to distinguish different catalog implementations. In case of absence, a local XMLCatalog will be used.

The supported prefix are: **xmlcatalog_** , **XMLFileCatalog_**, **mysqlcatalog_**, **edgcatalog_**

The supported protocols are: **mysql** for MySQL catalog; **http** for XML and EDG catalog; **ftp** for XML catalog; **file** for XML catalog.

Some examples of the contact strings for different catalogs are shown as follows:

- **MySQL:**

  mysqlcatalog_mysql://@lxshare070d.cern.ch:3306/testFCdb

  For the MySQL catalog, the [path] field represents the database name. The [username] field should be the username of the database. In case of absence, the login name of the user will be taken. The default value for the [port] field is 3306.

- **XML:**

  xmlcatalog_file:/tmp/FileCatalog.xml

  file:/tmp/FileCatalog.xml

  xmlcatalog_http://pc01.cern.ch/file001, if the catalog is at remote site and read only

- **EDG:**

  edgcatalog_http://rlstest.cern.ch:7777/edg-replica-location/services/edg-local-replica-catalog

10.4. How to construct the query string

The component supports query on the file metadata. In the current release, the query is a plain string consists of the attribute, "=" or "like" predicates and the desired value of the attribute. The wildcard "%" on the attribute value is allowed. Due to the string implementation of the XML and EDG catalogs, numerical queries are not supported in this release. All the string values must be quoted within a pair of single quotes. Example of some query strings: "jobid='sim101'", "owner like '%me%'"

The query strings can be passed to the command-line tools using the ?q option or passed to the catalog API as argument of the lookup methods. The query attribute can be either the metadata or 'pfname', 'lfname' and 'guid'.

FileID(GUID) is and should not be explicitly defined as an attribute because it is implicitly defined when the metadata schema is created. It is invisible to the user.

In this release only 'AND' logic is supported by all implementations, e.g. "jobid='sim101' AND owner line '%me%'".

10.5. How to use command-line tools of the component

The command-line tools provided by the File Catalog are in the /pool/Utilities/FileCatalog repository.

General options:

-h print help message

-u the catalog contact string. If absent, the contact string is picked up from the environment variable

POOL_CATALOG. The contact string specified by ?u option overrides that taken from the environment variable. To specify more than one catalogs in lookup operations, one should separate the contact string of each leaf catalog with a white space and close the entire string with double quotes.

-l LFN

-p PFN

-m customized cache size when using the catalog container, if this option is not given, the default cache size 1000 is assumed.

1. Register PFN

   FCregisterPFN -p pfname [-u uri -t filetype -h]

   Registers a PFN and assigns a unique FileID to it.

   Warning: You should only run this command for testing purpose, otherwise your real FileID will be lost. A file can be registered only from inside the job.

2. Register LFN

   FCregisterLFN -p pfname -l lfname [-u uri -h]

   Register the LFN (specified by **-l** option) to the PFN (specified by **-p** option).

3. Register a replica file name

   FCaddReplica -p pfname -r replica [-u uri -h]

   Register a replica file PFN (specified by **-r** option) to the master file PFN (specified by **-p** option).

4. Lookup PFNs

   FClistPFN [-l lfname -q query -m cachesize -u uri -t -h]

   -l option: list all PFNs with given LFN

   -q option: list all PFNs satisfy the query

   If no option is given, all PFNs are displayed.

   -t option: print PFNs in long format: PFN, filetype

   -u option: a contact string containing multiple catalogs separated by whitespaces are accepted. E.g. "mysqlcatalog_mysql://user@localhost/mycat1 xmlcatalog_file:mycat2.xml" .

   The first catalog is searched first.

5. Lookup LFNs

   FClistLFN [-p pfname -q query -m cachesize -u uri -h]

   -p option: list all LFNs with given PFN

   -q option: list all LFNs satisfy the query

   If no option is given, all LFNs are displayed.

   -u option: a contact string containing multiple catalogs separated by whitespaces are accepted. E.g. "mysqlcatalog_mysql://user@localhost/mycat1 xmlcatalog_file:mycat2.xml" .

   The first catalog is searched first.

6. Lookup Meta Data

   FClistMetaData [-l lfname -p pfname -q query -u uri -m maxcache -h]

   -l option: list metadata associated with the file with given PFN.

   -p option: list metadata associated with the file with given LFN.

   -q option: list all MetaData entries satisfy the query

   If no option is given, all metadata entries are displayed.

   -u option: a contact string containing multiple catalogs separated by whitespaces are accepted. E.g. "mysqlcatalog_mysql://user@localhost/mycat1 xmlcatalog_file:mycat2.xml" .

   The first catalog is searched first.

7. Describe the file meta data definition

   FCdescribeMetaData [-u uri -h]

   Describe meta data in the catalog.

   Format of the output:

   ((attribute1_name, attribute1_type), (attribute2_name, attribute2_type))

8. Define the meta data specification

   FCcreateMetaDataSpec [-F -m metadatadefinition -u uri -h]

   Create meta data specification specified by the ?m option.

   Format of the input:

   "(attribute1_name, attribute1_type), (attribute2_name, attribute2_type) "

   -F if a meta data schema already exists, drop the old one and create a new one.

9. Update the meta data specification

   FCupdateMetaDataSpec [-F -m metadatadefinition -u uri -h]

   Update meta data specification specified by the ?m option.

   Format of the input:

   "(attribute1_name, attribute1_type), (attribute2_name, attribute2_type) "

   -F if a meta data schema already exists, drop the old one and create a new one.

10. Insert meta data

    FCaddMetaData [-p pfname -l lfname -m metadata -u uri -h]

    Insert file meta data specified by the ?m option associated with file with given PFN or LFN. Format of the input:

    "((attribute1_name, attribute1_value), (attribute1_name, attribute2_value))"

    -p insert metadata associated to the PFN specified by this option.

    -l insert metadata associated to the LFN specified by this option.

11. Delete the selected PFN entry

FCdeletePFN [-p pfname -q query -u uri -h]

-p delete PFN entry specified by this option.

-q delete PFN entries selected by the query specified by this option.

If the pfn is the last one in the catalog, the entire mapping is deleted which has the same functionality of the command FCdeleteEntry -p

12. Delete the selected LFN entry

    FCdeleteLFN [-l lfname -q query -u uri -h]

    -l delete LFN entry specified by this option.

    -q delete LFN entries selected by the query specified by this option.

13. Delete the selected MetaData entry

    FCdeleteMetaData [-p pfname -l lfname -q query -u uri -h]

    -p delete MetaData entry associated with the PFN specified by this option.

    -l delete MetaData entry associated with the LFN specified by this option.

    -q delete MetaData entries selected by the query specified by this option.

14. Delete the selected PFN-LFN-MetaData mapping

    FCdeleteEntry [-p pfname -l lfname -q query -u uri -h]

    Delete the PFN-LFN-MetaData mapping of a file.

    -p delete the mapping associated with the PFN specified by this option.

    -l delete the mapping associated with the LFN specified by this option.

    -q delete the mapping selected by the query specified by this option.

15. Drop all the metadata and its specification

    FCdropMetadata [-u uri -h]

16. Extract a fragment from the source catalog and attach it to the destination catalog

    FCpublish -d destinationcatalog [-q query -s metadatadef -m cachesize -u sourcecatalog -h]

    The destination catalog is specified by the **-d** option.

    -u option: a contact string containing multiple catalogs separated by whitespaces are accepted. E.g. "mysqlcatalog_mysql://user@localhost/mycat1 xmlcatalog_file:mycat2.xml"

    -q option: extract/publish catalog fragment selected by given query.

    -s option: redefine destination catalog metadata schema. If an empty string is given to this option, no metadata will be imported to the destination catalog.

    If no query is specified, the entire source catalog will be appended to the destination catalog. The operation is atomic.

17. Rename PFN

    FCrenamePFN -p pfname -n newpfname [-u sourcecatalog -h] Rename the PFN (specified by the **-p** option) to the new one (specified by the **-n** option).

10.6. C++ API of the component

Class IFileCatalog is the interface of the component. It provides functions of the following types:

- Composite Catalog manipulations
- Connection and transaction control functions
- Cross catalog operations

The master catalog in the composite catalog is set by setWriteCatalog(). The addReadCatalog() method is used to add read-only catalogs into the composite catalog. The iteration of the leaf catalogs is achieved by getReadCatalog(), nReadCatalogs() and getWriteCatalog().

The catalog has two transaction states: in transaction and between transactions. Transaction starts with start() and ends with commit() or rollback(). Methods start() and commit() or rollback() should always be called in pairs. Exceptions will be thrown if these methods are not in pairs. Commit() methods takes IFileCatalog::CommitMode as arugment. REFRESH mode indicates the XML parser (for the XML catalog) will be reinitialised at the next start() method while ONHOLD mode indicates that parser states will not be changed at the next start() method. The default value of the argument is the REFRESH mode.

Between connect() and start(), start() and disconnect() are the between transaction states. Exceptions will be thrown when catalog operations are called at the between transaction states. Methods connect() and disconnect() also should be called in pairs. Exceptions will be thrown if connect() or disconnect() method is called twice in a row.

User can import a fragment of another catalog into the current catalog through IFileCatalog::importCatalog() method. The catalog fragment to be imported is selected by query.

Class IFCAction is the interface for catalog operations. The interface of this class is designed to be used by other POOL components. Its subclasses FCregister, FClookup and FCAdmin are responsible for general user register, lookup and schema manipulations. All methods in IFCAction class are also available in the subclasses. Each action instance is associated with one composite catalog by the method IFileCatalog::setAction().

The component supports associating metadata with the guid. The purpose of the metadata is to ease the file lookup and the catalog fragment selection. The metadata schema can be created, updated or droped. When dropping the metadata schema, all the file metadata will be lost as well.

The interface provides method to delete LFN, PFN and metadata entries in the catalog. However, one should use these methods with caution, if the selected PFN is the last one in the catalog, the entire mapping is deleted.

Class IFCContainer provides an interface to iterate on catalog entries. Only sequential iteration is supported through the method hasNext() and Next(). For scalability reason the results are retrieved first into a cache with default size of 1000 entries. The cache is used repeatedly until all results are retrieved, new batch of entries will overwrite the old entries in the cache. The iterating state can be reset through the reset() method. Each container is bound to a given filecatalog when created. There are four types of containers: PFNContainer, LFNContainer, MetaDataContainer and GuidContainer.

10.7. Example of usage

An example of application code is shown below:

```
IFileCatalog*                                                   mycatalog;
mycatalog->setWriteCatalog("file:test.xml");
IFileCatalog::FileID                                                  fid;
FCregister                                                              a;
```

```
mycatalog->setAction(a);
mycatalog->start();
a.registerPFN("aPFN","fileformat",fid);
a.registerLFN("aPFN","lfn:aPFN");
mycatalog->commit(IFileCatalog::ONHOLD);
std::string                          bestpfn,                          filetype;
mycatalog->start();
a.lookupBestPFN(fid, IFileCatalog::READ, IFileCatalog::SEQUENTIAL, bestpfn, filetype)
mycatalog->commit(IFileCatalog::ONHOLD);
FClookup                                                                    l;
mycatalog->setAction(l);
PFNContainer                       mypfs(mycatalog,                       100);
mycatalog->start();
l.lookupPFNByQuery("",mypfs);
while(mypfs.hasNext()){
                                        std::cout<<mypfs.Next()<<std::endl;
}
mycatalog->commit();
mycatalog->disconnect();
```

10.8. Python interface

The component provides a Python interface which allows the catalog operations to be called from a Python script instead of a compiled C++ application. The Python extension of the component consists of the following modules PyFileCatalog.py, PyAction.py, PyFCContainer.py, PyFileCatalogError.py and the C++ binding module of the backend implementation FileCatalog.so. To import these modules, $POOLProject/$arch/lib, $POOLProject/$arch/bin must be set in the $PYTHONPATH. To load backend catalog implementations, one should also set $SEAL_PLUGINS correctly and have all the external libraries the backend depends on set in $LD_LIBRARYPATH.

Following the common style of Python extension modules, this module is self-documenting. One can use dir and help functions to see the usage of the method. All the methods provided by the C++ API of IFileCatalog, IFCAction and IFCContainer class are available in Python with the same name. Besides, the python component has one extra method PyFClookup::lookupEntryByQuery() which retrieves the entire PFN-LFN-MetaData mapping by query.

The python component defines the following constants in the global scope which behave as enum type in C++ to be used as function arguments:

REFRESH , ONHOLD (argument of the commit() method)

NO_DELETE, DELETE_REDUNDANT (argument of the updateMetaDataSpec() method)

LFN, PFN, META, GUID (arguments of the PyFCContainer() constructor)

SEQUENTIAL, RANDOM, PRANDOM (access-mode arguments of the lookupBestPFN() method)

READ, WRITE, UPDATE (open-mode arguments of the lookupBestPFN() method)

Due to the language difference, the Python methods support default arguments and keyword arguments: e.g. the following calls are also legal

pfns=PyContainer(a_catalog_instance, PFN) (default cache size=1000)

pfns=PyContainer(a_catalog_instance, "cache size"=120, "container type"=PFN)

Examples of the python component can by found in the component unit test area:

/pool/PyFileCatalog/tests

10.9. Catalog schema migration

The main schema of the file catalog has been changed from POOL_1_3_x releases to POOL_1_4_x releases. In the later releases, the PFN attributes "job_status" and "file_status" are removed. For the

old catalogs produced by POOL_1_3_x to be readable by POOL_1_4_x software, user has to update the schema of the old catalog using the migration tools included in the POOL_1_4_x releases.

For the XML catalog, use the command:

XMLmigrate_POOL1toPOOL1.4 -u oldcatalog.xml -d newcatalog.xml

Note: here the protocol name "file:" should not be included in the catalog name.

For the MySQL catalog, use the script in src/ Scripts/FileCatalog/mysqlcatalog_migrate_POOL1toPOOL1.4.sql

mysql -u username -h hostname dbname< mysqlcatalog_migrate_POOL1toPOOL1.4.sql

For the EDG catalog, use the command:

EDGmigrate_POOL1toPOOL1.4 rlstest.cern.ch:7777

Note: updating of the EDG catalog should be performed only by the administrator of the rls service of the VO. Single user should not attempt to update the EDG catalog.

10.10. Detailed C++ API of the component

FileCatalog component depends on the following POOL components

/pool/POOLCore

/pool/AttributeList

All classes are defined in the pool namespace.

10.10.1. Public interfaces

IFileCatalog Class

connect

This method establishes the connection to the catalog backend. Exception is thrown in case of problems.

Syntax: void connect()

disconnect

This method disconnect from the catalog backend.

Exception is thrown in case of problems.

Syntax: void disconnect()

start

This method starts the catalog transaction. Exception is thrown in case of problems

Syntax: void start()

commit

This method commits the catalog transaction. Exception is thrown if the operation cannot be committed.

CommitMode can be IFileCatalog::REFRESH or IFileCatalog::ONHOLD. The default value is REFRESH.

Syntax: void commit(const CommitMode)

rollback

This method rolls back the catalog transaction. Exception is thrown if the opertation cannot be rolled back.

Syntax: void rollback()

addReadCatalog

This method adds the read-only catalog specified by the contact string to the composite catalog

Syntax: void addReadCatalog(const std::string& contact)

addReadCatalog

This method adds the read-only catalog instance to the composite catalog

Syntax: void addReadCatalog(FCLeaf* r)

setWriteCatalog

This method sets the read/writable catalog specified by the contact string

Syntax: void setWriteCatalog(const std::string& contact)

setWriteCatalog

This method sets the read/writable catalog instance in the composite catalog

Syntax: void setWriteCatalog(FCLeaf* w)

getWriteCatalog

This method returns the read/writable catalog

Syntax: IFileCatalog* getWriteCatalog()

getReadCatalog

This method returns the instance of the read-only catalog specified by the idex

Syntax: IFileCatalog* getReadCatalog(const unsigned long& idx)

nReadCatalogs

This method returns the number of read-only catalogs

Syntax: const size_t nReadCatalogs() const

setAction

This method associates an action with the catalog

Syntax: void setAction(IFCAction&)

importCatalog

This method appends a fragment of the given source catalog to the current catalog. The fragment is selected by query on the source catalog metadata. If the query string is empty, entire source catalog is appended to the current catalog. One can specify the default cache size for this operation. The default value is 1000.

Syntax: void importCatalog(IFileCatalog* fc, const std::string& query, unsigned int

cachesze=FCDEFAULT_CACHE_SIZE) const

isReadOnly

This method tells if the catalog is read-only

Syntax: bool isReadOnly() const

IFCAction class

isWritableEntry

This method tells if the given guid is from the read/writable catalog

Syntax: bool isWritableEntry(IFileCatalog::FileID& fid)

registerPFN

This method registers a file with the given PFN and file type; returns the corresponding FileID from the argument list. Exception is thrown if PFN is already registered. This operation is performed on the read/writable catalog.

Syntax: void registerPFN(const std::string& pfn, const std::string& filetype, IFileCatalog::FileID& fid)

lookupFileByPFN

This method returns the FileID and file type with given PFN

Syntax: void lookupFileByPFN(const std::string& pfn, FileID& fid, std::string& filetype)

lookupFileByLFN

This method returns the FileID with given LFN

Syntax: void lookupFileByLFN(const std::string& lfn, FileID& fid) getMetaDataSpec This methods returns the metadata schema definition of the catalog

Syntax: void getMetaDataSpec(MetaDataEntry& spec)

lookupBestPFN

This method returns a PFN associated with the given FileID. The first PFN found is returned. The file type is also returned. FileOpenMode and FileAccessPattern are passed as a hint to the Grid components for file transfer.

Syntax: void lookupBestPFN(const FileID& fid, const FileOpenMode& omode, const FileAccessPattern& amode, std::string& pf, std::string& filetype)

visitFCLeaf

This methods allows the leaf catalog to register itself

Syntax: void visitFCLeaf(IFileCatalog* cat)

visitFCComposite

This methods allows the composite catalog to register itself

Syntax: void visitFCComposite(IFileCatalog* cat)

FCregister class

registerLFN

This method registers a LFN associated with the given PFN.

Syntax: void registerLFN(const std::string& pfn, const std::string& lfn)

addReplicaPFN

This method adds the PFN of a replica to its master copy PFN.

Syntax: void addReplicaPFN(const std::string& pfn, const std::string& rpf)

renamePFN

This method replaces a given PFN with a new PFN.

Syntax: void renamePFN(const std::string& pfn, const std::string& newpfn)

registerMetaData

This method inserts metadata values of a file with given FileID.

Syntax: void registerMetaData(const IfileCatalog::FileID& fid, MetaDataEntry& attrs)

FClookup class

lookupPFN

This method returns PFNs associated with given FileID.

Syntax: void lookupPFN(const IFileCatalog::FileID& fid, PFNContainer& pfs)

lookupLFN

This method returns LFNs associated with given FileID.

Syntax: void lookupLFN(const IFileCatalog::FileID& fid, LFNContainer& lfs)

lookupPFNByQuery

This method returns PFNs satisfy the query.

Syntax: void lookupPFNByQuery(const std::string& query, PFNContainer& pfs)

lookupLFNByQuery

This method returns LFNs satisfy the query.

Syntax: void lookupLFNByQuery(const std::string& query, LFNContainer& lfs)

lookupMetaDataByQuery

This method returns meta data selected by the query.

Syntax: void lookupMetaDataByQuery(const std::string& query, MetaDataContainer& metas)

lookupPFNByLFN

This method returns PFNs associated with given LFN

Syntax: void lookupPFNByLFN(const std::string& lfn, PFNContainer& pfs)

lookupLFNByPFN

This method returns LFNs associated with given PFN

Syntax: void lookupLFNByPFN(const std::string& pfn, LFNContainer& lfs)

lookupFileByQuery

This method returns all FileIDs selected by the query.

Syntax: void lookupFileByQuery(const std::string& query, GuidContainer& fids)

FCAdmin class

deleteLFN

This method deletes the specified LFN.

Syntax: void deleteLFN(const std::string& lfn)

deletePFN

This method deletes the specified PFN. If the PFN is the last one associated with a file, all associated LFN and metadata are deleted as well.

Syntax: void deletePFN(const std::string& pfn)

deleteMetaData

This method deletes the metadata associated with the FileID.

Syntax: void deleteMetaData(const IFileCatalog::FileID& fid)

dropMetaDataSpec

This method drops the metadata and its definition. Syntax: void droptMetaDataSpec()

createMetaDataSpec

This method creates the metadata definition of the catalog.

Syntax: void createMetaDataSpec(MetaDataEntry& spec)

updateMetaDataSpec

This method updates metadata definition in the catalog or create one if catalog has no metadata defined. The default value of the FCMetaUpdateMode argument is NO_DELETE which only adds new attributes. DELETE_REDUNDANT mode will delete the old attributes which are absent from the new metadata definition. deleteEntry This method deletes the entire mapping associated with the given FileID.

Syntax: void deleteEntry(const IFileCatalog::FileID& fid)

template<typename Item>IFCContainer, PFNContainer, LFNContainer,MetaDataContainer, GuidContainer IFCContainer

The constructor creates an instance of the container bounded to a given catalog and initialised with given cachesize.

Syntax: IFCContainer(IFileCatalog* catalog, const size_t cachesize=1000)

reset

This method resets the iterator state to its initial values.

Syntax: void reset()

hasNext

This method tells if there is next entry in the container.

Syntax: bool hasNext()

Next

This method retrieves the next item from the container.

Syntax: Item& Next()

max_size

This method tells cache capacity of the container

Syntax: size_t max_size() const

11. FileCatalog component: Reference Manual

11.1. Signatures of public interfaces

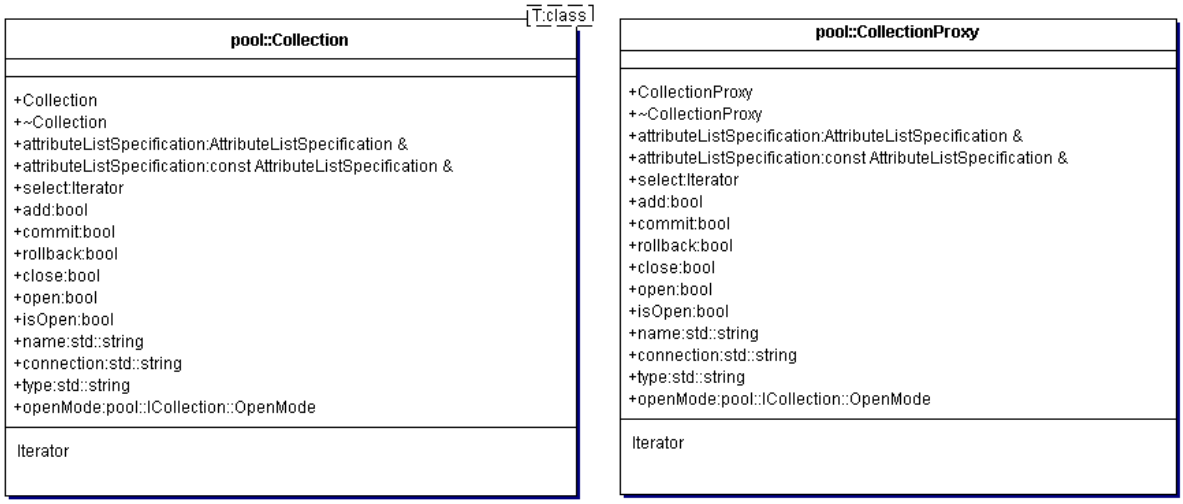
```
[Method/code                                                    fragment]
```

11.2. Command line tools

```
[Method/code                                                    fragment]
```

12. Collection component: User level semantics

12.1. Public classes UML diagram



Class diagram showing public methods of Collection package

12.2. User interfaces

This section describes the most common user interfaces to the Collection package. A detailed listing of all public methods available to the user from this package is presented in section Section 13.

12.2.1. Collection creation

The recommended way for a user to create a new collection of known data type is the following:

```
pool::Collection<     MyDataObjType     >     myCollection(
                            myDataSvc,
                          <collection                          type>,
                  <collection               location>,
                                                    myCollectionName,
                       pool::ICollection::CREATE                    );
```

where MyDataObjType is the type of data object to be stored in the collection, myDataSvc is a pointer to an instance of the POOL data service, <collection type> is the specific type of POOL collection being created (e.g. MySQLCollection or RootCollection), <collection location> is the location of the database containing the data object references (in the form of Token identifiers) and associated metadata, and myCollectionName is the persistent name of the collection. The last argument specifies that the collection is to be created if and only if a collection of the same name does not already exist. If the collection is to overwrite an existing collection of the same name then this argument should be replaced with pool::ICollection::CREATE_AND_OVERWRITE. If a collection of the same name exists and the user would like to append more data to it then this argument should be pool::ICollection::UPDATE. If the user does not wish to specify the type of data object being stored upon collection creation then the following implementation should be used instead:

```
pool::CollectionProxy            myCollection(
                            myDataSvc,
                          <collection                          type>,
                  <collection               location>,
                                                    myCollectionName,
                       pool::ICollection::CREATE                    );
```

where the input arguments have the same meaning as above.

If the collection itself is to be stored in an ensemble of other collections then a MultiCollection object also needs to be opened to contain this ensemble:

```
pool::MultiCollection<     MyDataObjType     >     myMultiColl(
                                                        myDataSvc,
                        <multi-collection                    type>,
                <multi-collection              location>,
                                                 myMultiCollectionName,
                       pool::ICollection::CREATE                    );
```

where <multi-collection type> is the type of multi-collection being created (e.g. MSQLMultiCollection or RootMultiCollection), <multi-collection location> is the database location of the persistent multi-collection and myMultiCollectionName is the name of the persistent multi-collection.

12.2.2. Attribute list creation

To create an attribute list for a given event the user must first obtain the attribute list specification object for the collection containing the event:

```
pool::AttributeListSpecification&      myAttribSpec      =
                          myCollection.attributeListSpecification();
```

This object is then used to construct a new attribute list object:

```
pool::AttributeList      myAttribList(      myAttribSpec      );
```

This object is then filled with the desired metadata using the setValue method of the Attribute class for each metadata member in the list specification. The syntax for each entry is of the form:

```
myAttribList[ <metadata name> ].setValue( <metadata value> );
```

where <metadata name> is the name of the given attribute in the list specification and <metadata value> is the value of this attribute for the given event.

12.2.3. Adding data objects and associated metadata to a collection

To add data to a collection the user must first create a reference to the given data object as follows:

```
pool::Ref< MyDataObjType > myDataObjRef(
                    myDataSvc,
            new              MyDataObjType()          );
```

Where MyDataObjType is the type of data object being stored. Then this object must be marked for writing by the persistency service:

```
myDataObjRef.markWrite( placement );
```

where placement is an object of type pool::Placement which specifies to the persistency service the file and container to which the data object is to be persistified and the type of storage technology used (e.g. ROOT I/O). Then the data object reference and its associated attribute list are added to the collection as follows:

```
myCollection.add( myDataObjectRef, myAttribList );
```

After all data has been added to the collection, these changes must be committed and the collection should be closed to complete the persistency process:

```
myCollection.commit();
myCollection.close();
```

The collection itself may be added to a multi-collection as follows:

```
myMultiColl.add( myCollection, myCollAttribList );
```

where myCollAttribList is a list of metadata associated with the collection itself.

12.2.4. Performing queries on a collection

To perform a query on an existing explicit collection of known type, the collection should be opened as follows:

```
pool::Collection< MyDataObjType > collection(
                    myDataSvc,
                  <collection type>,
          <collection location>,
                    myCollectionName,
              pool::ICollection::READ );
```

where the arguments have the same meanings as in section Section 12.2.1. On the other hand, if the collection is implicit then the collection should be opened using the following arguments instead:

```
pool::Collection<     SimpleTestClass     >     collection(
                                                      myDataSvc,
                                                ImplicitCollection,
                                <data                      location>,
                               <container                      name>,
                          pool::ICollection::READ                  );
```

where <data location> specifies the database file containing the persistent data objects and must take the form <file name type>:<file name> where <file name type> must be one of the following: PFN (physical file name), LFN (logical file name) or FID (file identifier) and <container name> is the container where the data is stored within the file.

Next, the user needs to construct an iterator over the events in the collection. If a server side query is desired then the iterator should be constructed as follows:

```
pool::Collection<    myDataObjType    >::Iterator    myIter    =
       collection.select(     myPrimaryQuery,     mySecondaryQuery,
                             myIteratorOptions                    )
```

where myPrimaryQuery consists of a string of predicates applied to the attributes of the collection. The syntax of these predicates depends on the type of collection being queried. For instance, the MySQL query 'myAttribute > 2 AND myAttribute < 10' must take the form 'myAttribute > 2 && myAttribute < 10' for ROOT (note that both forms happen to work for MySQL). The argument mySecondaryQuery is set to "" by default but can be useful when the collection consists of an ensemble of collections (created via a MultiCollection object). This argument could then consist of a string of predicates applied to the attributes of the collection ensemble itself. The argument myIteratorOptions is a space separated selection of the following arguments:

```
                -                    MySQLCollection:
 -  "FetchOne":  The  default;  Selected  rows  are  not  cached.
- "FetchAll": Selected rows are cached at initialization time.
 -   "SELECT   <attribute   list>":   where   <attribute   list>
  is a comma separated list of attributes to be read in from
  persistence; "SELECT *" selects all attributes; "SELECT" with
    no   arguments   means   only   the   Token   is   selected.
- "NoToken": Flag to exclude Token selection from persistence.
                         -                                RootCollection:
- "SELECT <attribute list>" or "ATTRIBUTE_LIST <attribute list>":
  where <attribute list> is a comma or space separated list of
  attributes to be read in from persistence; "SELECT *" selects all
  attributes; "SELECT" with no arguments means only the Token is
                                                          selected.
- ROOT_SELECTION_SERVER: Performs selection on a remote server
                                                  (experimental):
      e.g.       "ROOT_SELECTION_SERVER       pcepsft01.cern.ch"
   On   the   remote   server   the   simple   server   script
  RootCollection/scripts/server.C  has  to  be  started.  Type
                          "root                        server.C+".
```

Inside the event loop the user can, among other things, retrieve a reference to the data object. This reference can then be used to obtain more information about the data object. For instance, to retrieve the identifier of the database containing the given data object in persistent form, one could write the following:

```
std::string     dbFileId     =     myIter.ref().token()->dbID();
```

If a client side query is desired instead, then the iterator is created with an empty query argument and the cuts on the attributes are made inside the event loop itself:

```
Pool::Collection<    myDataObjType    >::Iterator    myIter    =
     collection.select(     "",     "",     myIteratorOptions    );
```

Alternatively, one may use a CollectionProxy object to perform the query. The syntax is similar to that of Collection<T> except that the user does not need to know the collection type to open the col-

lection and obtain an iterator for the query:

```
pool::CollectionProxy             collection(
                        myDataSvc,
                    <collection                              type>,
          <collection                location>,
                                              myCollectionName;
                pool::ICollection::READ                       );

pool::CollectionProxy::Iterator        myIter        =
collection.select(    myQuery,    "",    myIteratorOptions    );
```

However, this time if a reference to a given event is needed the object type must be specified in the call to the iterator's ref() method. So for instance, to get the identifier of the persistent data object's database the user would need to implement the following:

```
std::string             dbFileId             =
myIter.ref<       MyDataObjType       >().token()->dbID();
```

12.2.5. Exceptions Generated

Like the other POOL components, the collections and metadata classes are designed to throw exceptions in the event of unexpected program behavior but to let decisions about subsequent actions be handled outside the scope of POOL. In general, the collection packages throw exceptions of type seal::Exception. Each thrown exception will contain a brief description of the exception, the method which threw it, and the SEAL error status of the exception (e.g. seal::Status::FATAL). Additionally, some collection code may still throw exceptions of type std::exception.

12.3. Examples of usage

Three of the most general usage examples of this component consist of the following:

1. Declare an object of type Collection<T> or CollectionProxy using an open mode that either creates a new collection or overwrites an existing one. Define the collection's list of metadata types via the AttributeListSpecification class. Then write event data references and their associated metadata into the collection.

2. Declare an object of type Collection<T> or CollectionProxy using an open mode that reads an existing collection. Then declare an object of type CollectionIterator<T>::Iterator or CollectionProxy::Iterator to perform queries on the collection via cuts on its associated metadata. The query may be performed on either the client side or the server side. In the latter case, the query predicates are used as arguments to the iterator's constructor.

3. Declare an object of type Collection<T> or CollectionProxy using an open mode that updates an existing collection. Retrieve the collection's list of metadata types in the form of an AttributeListSpecification object. Then write additional event data references and their associated metadata into the collection.

The following sections illustrate three specific examples using the collections and metadata packages: 1) Writing an explicit collection 2) Reading an explicit collection 3) Updating an explicit collection. The complete implementations of these examples can be found in the collection integration tests Collection_Write, Collection_Read, and Collection_Update in the POOL CVS repository: *POOL Integration Tests* [http://lcgapp.cern.ch/cgi-bin/viewcvs/viewcvs.cgi/pool/Tests/]

12.3.1. Writing an Explicit Collection

First, an explicit collection is created to store references to data objects of type SimpleTestClass (which is defined in the POOL package Tests/Libraries/TestDictionary) as follows:

```
pool::Collection<     SimpleTestClass     >     collection(
                                                      dataSvc,
                          <collection                   type>,
                         <collection                 location>,
                          <collection                     name>,
                     pool::ICollection::CREATE                );
```

where dataSvc is a pointer to an instance of the POOL data service, collection type is the specific type of collection being created (e.g. MySQLCollection or RootCollection), <collection location> is the location of the collection.s data object references and associated metadata and <collection name> is the collection.s persistent name (e.g. a MySQL table name). The last argument is the open mode of the collection. If the collection already exists and is to be overwritten, then this argument should be replaced by pool::ICollection::CREATE_AND_OVERWRITE. These arguments presumably come from outside the application code. For example, in ATLAS they would probably be specified in an Athena job options file. The metadata schema of the collection is also defined:

```
  pool::AttributeListSpecification&       attribSpec       =
                          collection.attributeListSpecification();
     attribSpec.push_back(          "run",          "int"          );
    attribSpec.push_back(         "event",          "int"          );
 attribSpec.push_back(      "aUInt",      "unsigned      int"      );
   attribSpec.push_back(        "aShortInt",        "short"        );
attribSpec.push_back(    "aUShortInt",    "unsigned    short"     );
   attribSpec.push_back(         "aLongInt",         "long"        );
attribSpec.push_back(    "aULongInt",    "unsigned    long"      );
   attribSpec.push_back(         "aFloat",         "float"        );
   attribSpec.push_back(        "aDouble",        "double"        );
    attribSpec.push_back(         "aBool",          "bool"        );
   attribSpec.push_back(        "aString",        "string"        );
```

Note: As of POOL_1_6_0 Token attributes are supported in addition to the primitive types shown above. This allows the reference to another object or file to be stored as part of the collection metadata. See the Collection_Write integration test for an example of usage: *Collection_Write Test* [http://lcgapp.cern.ch/cgi-bin/viewcvs/viewcvs.cgi/pool/Tests/Collection_Write]

Then a loop over runs and events is performed and for each event a new SimpleTestClass object is created and marked for persistency, this object and its associated list of metadata are given values and finally a reference to the SimpleTestClass object along with its associated metadata are added to the collection:

```
for     (int     run     =     1;     run     <=     3;     run++)
                                                              {
    for     (int     evt     =     1;     evt     <=     10;     evt++)
                                                              {
                       pool::Ref<SimpleTestClass>
           simpleTestClass(dataSvc,     new     SimpleTestClass());

              simpleTestClass.markWrite(         placement         );

                                    simpleTestClass->setNonZero();

              pool::AttributeList         attributeList(attribSpec);
            attributeList[      "run"      ].setValue(      run      );
           attributeList[     "event"     ].setValue(     evt     );
            attributeList[      "aUInt"      ].setValue(      8      );
          attributeList[    "aShortInt"    ].setValue(    -4    );
          attributeList[    "aUShortInt"    ].setValue(    6    );
           attributeList[    "aLongInt"    ].setValue(    -9    );
           attributeList[    "aULongInt"    ].setValue(    5    );
            attributeList[     "aFloat"     ].setValue(     3.5     );
           attributeList[     "aDouble"     ].setValue(     9.7     );
            attributeList[      "aBool"     ].setValue(     true     );
       attributeList[   "aString"   ].setValue(   "test   string"   );

             collection.add(simpleTestClass,        attributeList);
                                                              }
```

```
}
```

The input argument to the markWrite method is an object of type pool::Placement which is defined by the POOL persistency service and contains all the information necessary to persistify the data object. After references to all data objects and their associated metadata have been added to the collection these changes are committed and the collection is closed:

```
collection.commit();
collection.close();
```

12.3.2. Reading an Explicit Collection

An existing explicit collection of objects of type SimpleTestClass is reopened in order to perform queries on its data:

```
pool::Collection<     SimpleTestClass     >     collection(
                                                      dataSvc,
                          <collection                   type>,
                         <collection                 location>,
                           <collection                   name>,
                    pool::ICollection::READ                 );
```

where dataSvc, <collection type>, <collection location> and <collection name> have the same meanings as above and they would again come from some outside application code. Then a server side query is defined and used to obtain an iterator to the collection as follows:

```
pool::Collection<   SimpleTestClass   >::Iterator   collIter   =
            collection.select(          <primary          query>,
                                  <secondary              query>,
                             <iterator          options>          )
```

where <primary query> consists of the desired predicates (e.g. 'event_number > 2 AND event_number < 5' for a MySQL query), <secondary query> can contain a secondary set of predicates (which, as mentioned in section Section 12.2.4, is presently only implemented for multi-collections and has a default value of ""), and <iterator options> presently has two possible values: FetchAll and FetchOne (the default). FetchAll stores all of the collection.s data object references and associated metadata in cache during iterator construction, while FetchOne retrieves each collection element upon request without caching. If a client side query is preferred then <primary query> should be set to "" so that the predicates can be applied inside the iterator loop instead. Next, the query is performed by incrementing the resulting iterator over the desired set of events and reading some information for each event that satisfies the query:

```
while          (          collIter.next()          )
                                                      {
      collIter->streamOut(            std::cout            );

   pool::AttributeList         attributeList         =
                                    collIter.attributeList();

      attributeList.print(            std::cout            );

      std::string           dbFileId           =
                             collIter2.ref().token()->dbID();
                  std::string                       bestPFN;
                 std::string                      fileType;
            filecatalog->lookupBestPFN(
                                                 dbFileId,
                                  pool::IFileCatalog::READ,
             pool::IFileCatalog::SEQUENTIAL,
                              bestPFN,
                            fileType                        );
```

```
std::cout << "Physical file name is " << bestPFN
                                  <<                    std::endl;
 std::cout    <<    "File    type    is    "    <<    fileType
                                  <<                    std::endl;
                                                                }
```

In this example, the data object (i.e. a SimpleTestClass object) itself is accessed first and its data is printed to standard output. Then the object's list of associated metadata is also accessed and printed to standard output. Finally, the persistent reference to the data object is accessed and used to obtain the name of the physical file in which the data object is stored in its persistent form. After all desired queries have been performed the collection is closed:

```
                                                    collection->close();
```

12.3.3. Updating an Explicit Collection

An existing explicit collection of objects of type SimpleTestClass is reopened in order to add new data objects and associated metadata to the collection:

```
pool::Collection<     SimpleTestClass     >     collection(
                                                        dataSvc,
                        <collection                       type>,
                       <collection                     location>,
                         <collection                       name>,
                   pool::ICollection::UPDATE                   );
```

where the meanings of dataSvc, <collection type>, <collection location> and <collection name> are the same as described in the write and read examples above. Then the metadata list specification is obtained from the collection:

```
 pool::AttributeListSpecification&       attribSpec       =
                          collection.attributeListSpecification();
```

A loop is performed over runs and events and for each event a new SimpleTestClass object reference and its associated metadata are added to the collection in the same way as described for the write example given above:

```
for     (int     run     =     1;     run     <=     3;     run++)
                                                                {
    for     (int     evt     =     1;     evt     <=     10;     evt++)
                                                                {
                        pool::Ref<SimpleTestClass>
             simpleTestClass(dataSvc,    new    SimpleTestClass());

              simpleTestClass.markWrite(         placement         );

                                    simpleTestClass->setNonZero();

                pool::AttributeList         attributeList(attribSpec);
            attributeList[      "run"      ].setValue(      run      );
            attributeList[     "event"     ].setValue(     evt     );
             attributeList[      "aUInt"      ].setValue(      3      );
            attributeList[    "aShortInt"    ].setValue(    -6    );
            attributeList[    "aUShortInt"    ].setValue(    7    );
            attributeList[    "aLongInt"    ].setValue(    -2    );
            attributeList[    "aULongInt"    ].setValue(    10    );
             attributeList[     "aFloat"     ].setValue(     4.6     );
            attributeList[     "aDouble"     ].setValue(     8.2     );
            attributeList[     "aBool"     ].setValue(     false     );
      attributeList[ "aString" ].setValue( "another test string" );

             collection.add(simpleTestClass,         attributeList);
```

```
                                                                    }
                                                }
```

After all additional data objects and their associated metadata have been added to the collection these changes are committed and the collection is closed:

```
collection.commit();
collection.close();
```

12.3.4. Solutions to Common Problems

1. If a process that writes data object references and their associated attributes to a collection is aborted before the process has a chance to commit changes made to the file catalog, then a subsequent attempt to create and overwrite the existing collection will fail if the name of the database file that contains the collection's persistent data objects (e.g. TestDbFile.pool in the Collection_Write integration test) was not already added to the local file catalog in a previous run. This is because the storage service knows that the file exists but the file catalog does not. The present solution to this problem is to simply delete the database file before rerunning. A more permanent solution is to rewrite one's code to commit and restart the file catalog after the first event has been marked for writing. This will assure that the data file name is registered in the catalog despite any possible crashes during production.

2. The attribute list specification of a collection is not meant to be changed during the lifetime of the collection and an attempt to do so will result in a run time error.

3. Presently, there is not sufficient access restriction on the MySQL database used by POOL to store data object Tokens and their associated metadata (i.e. the persistent collections) to prevent tables from being overwritten. This will be changed eventually but it does not seem to pose a major inconvenience at the moment because the tables are primarily used for regression testing.

13. Collection component: Reference Manual

13.1. Signatures of public interfaces

This section gives a description of the public methods of the collection classes available to the user.

13.1.1. Collection<T>

A type-safe class used to store, retrieve and update collections of data objects and their associated metadata. The class is templated according to the type of data object that it stores. It contains the following public methods:

Collection()	Connects to the database and creates a concrete object (e.g. MySQLCollection or RootCollection)
	
	Syntax:
	
	`Collection( IDataSvc* dataS`
	`std::string collectionType,`
	`std::string co`
	`std::string coll`
	`ICollection::OpenMode openMo`

~Collection()	Default destructor.

	Syntax: `Not impleme`

attributeListSpecification()	Returns the metadata list specification of the collection.
	Syntax:
	`const AttributeListSpecifica`
	`attributeListSpecification() c`

select()	Returns an iterator over a subset of the collection data satisfies a set of predicates on the data's associat metadata. Iterator options determine whether caching is u to store the collection data before the query (no caching the default).
	Syntax:
	`Iterator select( std::string primaryQuery,`
	`std::string secondaryQ`
	`std::string IteratorOptions`

add()	Adds an event object and its associated metadata to collection.
	Syntax:
	`bool add( const Ref<T>&`
	`const AttributeList& attrList`

commit()	Persistifies last changes made to the collection. **Syntax:** `bool comm`
rollback()	Aborts last changes made to collection before committing. **Syntax:** `bool rollba`
close()	Closes collection explicitly, aborting uncommitted changes. **Syntax:** `bool clo`
open()	Reopens collection explicitly after it was closed.

	Syntax: `bool`

isOpen()	Checks if collection is open. **Syntax:** `bool isOpen()`
name()	Returns persistent name of collection. **Syntax:** `std::string name()`
connection()	Returns location of persistent collection. **Syntax:** `std::string connection()`
type()	Returns type of concrete collection being (MySQLColletion, RootCollection, etc.). **Syntax:** `std::string type()`
openMode()	Returns the mode used to open the collection READ, etc.) **Syntax:** `pool::ICollection::OpenMode openMode()`

13.1.2. Collection<T>::CollectionIterator

An iterator class to perform queries on Collection<T> objects. Depending on how the iterator is created it can either perform server side queries on a subset of the collection entries or client side queries on the whole collection. The iterator also provides an option to store the results of the query in cache. The public methods of this class are:

Iterator()	Copy constructor. **Syntax:** `Iterator( const Iterator&`

~Iterator()	Default destructor. **Syntax:** `Not impleme`

operator*()	Data object dereference operator.
	Syntax:
	`T& operato`
	`const T& operator*() c`

operator->()	Data object pointer operator.
	Syntax:
	`T* operator`
	`const T* operator->() c`

next()	Retrieves the next data object in the collection. **Syntax:** `bool ne`

attributeList()	Returns the list of associated metadata for the data object. **Syntax:** `const AttributeList& attributeList() c`

ref()	Returns the underlying reference to the data object.
	Syntax:
	`Ref<T>& r`
	`const Ref<T>& ref() c`

isValid()	Checks if iterator is valid. **Syntax:** `bool isValid() c`

13.1.3. CollectionProxy

A class used to store, retrieve and update collections of data objects and their associated metadata. It is similar to the Collection<T> class except that the iterator of the CollectionProxy class does not

automatically perform a type check on each data object that it references, thus reducing iteration time. Instead, such type checking is left as an option for the user via the introduction of a new Boolean method of the iterator class called isCurrentObjectType(). These differences allow a CollectionProxy object to be created without specifiying the type of its data, while its ref() method must be type dependent to resolve this ambiguity when data objects are recorded in or retrieved from persistency. As a result, the public interface to the CollectionProxy class is identical to that of the Collection<T> class and the class CollectionProxy::Iterator is almost identical to Collection<T>::Iterator except that it does not define any overloaded dereferencing operators and it contains the following additional or modified methods:

isCurrentObjectOfType()	Checks whether the data object to which the iterator is currently pointing **Syntax:** `template<class T> bool isCurrentObje`
ref()	Retrieves a reference to the current data object based on its type. **Syntax:** `template<class T> Ref<T> ref()`

See the Collection_Read collection integration test for a query example using the CollectionProxy class: *POOL Collection Read Test* [http://lcgapp.cern.ch/cgi-bin/viewcvs/viewcvs.cgi/pool/Tests/Collection_Read/]

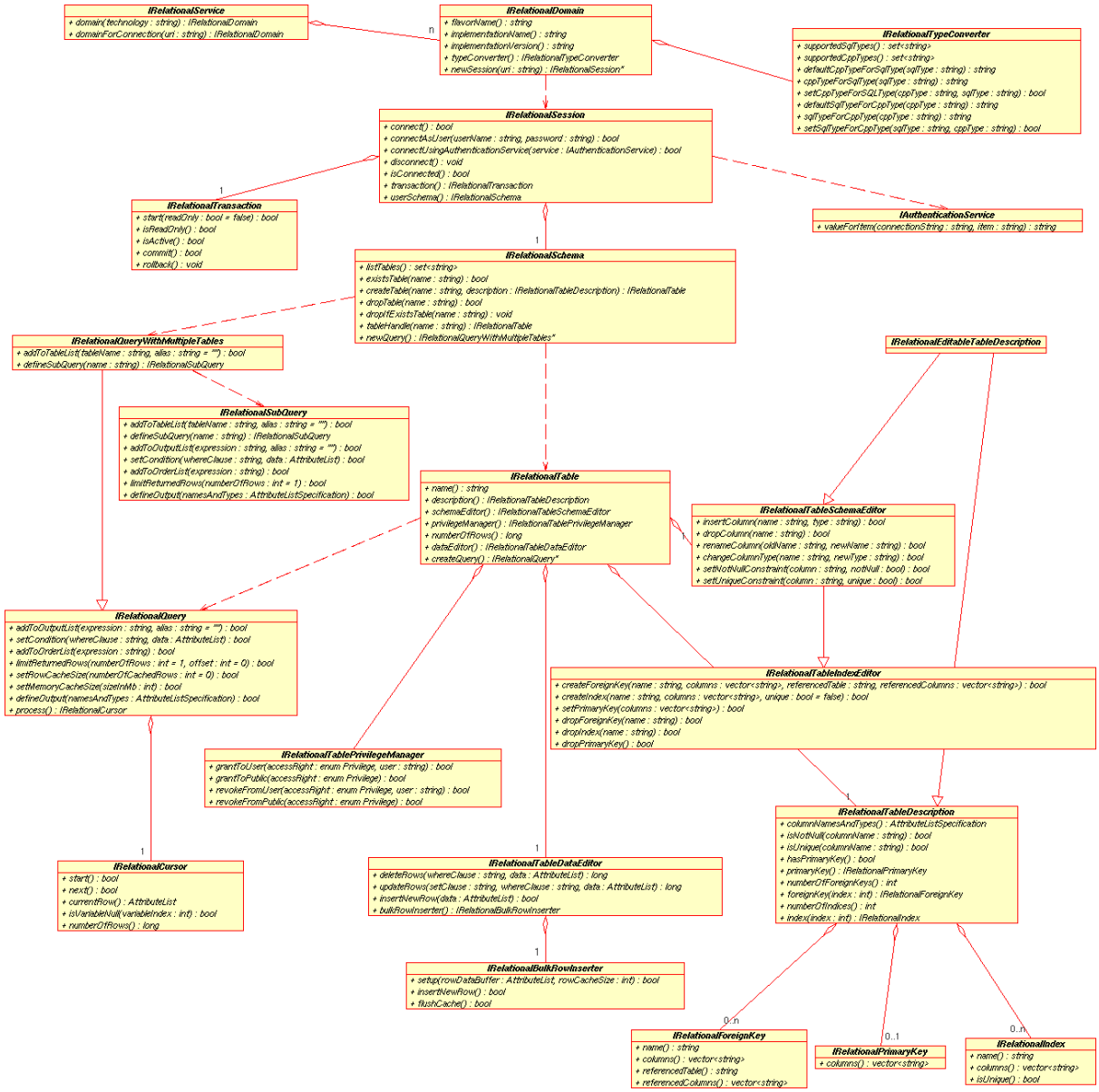
13.2. Command line tools

`[Method/code fragment]`

14. RelationalAccess component: User level semantics

14.1. UML class diagram of the RelationalAccess public interfaces

The public interfaces of the component are depicted in the following UML class diagram.



UML class diagram of the RelationalAccess public interfaces

14.2. Exceptions

Most of the methods of the abstract interfaces return a boolean variable indicating the success of the operation. In case false is returned there is usually an error message in the form of a SEAL message stream that is generated by the implementation plugin.

Methods that normally return a reference to an object throw a C++ exception which is always derived from the base **RelationalException** class, which in turn is an extension of the SEAL Exception. Refer to the documentation of a specific interface for more information on the exceptions that it may throw.

14.3. Accessing a database

14.3.1. The Relational Service and Technology Domains

The software entry point for accessing a relational database through the RelationalAccess package is the **IRelationalService** interface. The package provides a default implementation as a *SEAL component* with the label "POOL/Services/RelationalService".

The IRelationalService interface provides access to **IRelationalDomain** objects for the various different technology types. The IRelationalDomain interface provides information about the technology type, the implementation of the corresponding plugin module and its version.

An IRelationalDomain object is asked by the IRelationalService specifying either the technology type, or a connection string (where the technology type is encoded). The IRelationalDomain is the base class for the corresponding SEAL component of a technology plugin. The component should be named according to the technology name with the following convention: If the technology type is "MyRDBMS", then the IRelationalService will be searching for a plugin labeled as "POOL/RelationalPlugins/MyRDBMS".

14.3.2. The C++ <-> SQL type conversion mechanism

From the IRelationalDomain interface one may obtain a reference to the corresponding **IRelationalTypeConverter** object. The latter is responsible for all the C++ <-> SQL type conversions that are performed internally by the technology-specific plugin. Its presence in the design is required by the requirement that the rest of the RelationalAccess interfaces expose only C++ types for the description of the schema of a table or the result set of a query.

The IRelationalTypeConverter object holds the default mappings and the initial list of supported types predefined by the plugin (see the documentation for each specific plugin). The user may update the current type mapping and insert additional recognizable SQL types and new conversion rules. The C++ <-> SQL type conversion is not 1-1. This allows for example multiple SQL types, like CHAR(n), VARCHAR(n), etc to be recognized as std::string. Moreover when a C++ -> SQL type conversion is defined, the corresponding SQL -> C++ conversion is NOT automatically added, and vice versa.

14.3.3. Connection strings

The IRelationalDomain interface acts as a factory for **IRelationalSession** objects which are used for the actual connection to a relational database. Such an object is obtained by specifying a connection string which has to have the format

```
technology_protocol://hostName:portNumber/databaseOrSchemaName:sidNumber
```

where technology can be oracle, mysql, odbc, etc. (all small letters) and refers to the actual plugin implementing the RelationalAccess abstract interfaces. The protocol is optional and can be file, http, etc. It is used for server-less, file-based databases, like SQLite. The port and sid numbers are optional as well.

14.3.4. The authentication mechanism

The IRelationalSession interface allows a user connect to a database specifying a user name and a password. Alternatively a reference to an **IAuthenticationService** object may be passed explicitly or implicitly looking for such a registered service anywhere in the corresponding context tree.

The IAuthenticationService is a simple interface which returns for a given connection string the connection parameters (typically user name and password). It has been introduced in the design of the component so that authentication parameters need not appear in the connection string, which may have to be shared by users with different access rights. Refer to the relevant documentation for the semantics of the existing implementations of this interface.

14.3.5. Transactional context

Every operation with a relational database using the RelationalAccess component is executed within a transactional context. There is a single transactional context for each session. A reference to the **IRelationalTransaction** interface can be retrieved by an IRelationalSession object once a connection has been established.

A user can start a transaction in an update (default) or read-only mode, commit and roll back the changes.

14.3.6. Connectivity example

.

```
.
.
#include                                "RelationalAccess/IRelationalService.h"
#include                                 "RelationalAccess/IRelationalDomain.h"
#include                                "RelationalAccess/IRelationalSession.h"
#include                             "RelationalAccess/IRelationalTransaction.h"
#include                                                 "POOLCore/POOLContext.h"
#include                                                   "SealKernel/Context.h"
.
.
.
pool::POOLContext::loadComponent(    "POOL/Services/RelationalService"    );
.
.
.
//   Retrieving   a   handle   to   the   registered   relational   service
seal::IHandle<pool::IRelationalService> serviceHandle = pool::POOLContext::context()->query

//   Accessing   the   domain   object   for   the   "myrdbms"   technology.
std::string     connectionString     =     "myrdbms://mydbhost/mydbschema";
pool::IRelationalDomain& domain = serviceHandle->domainForConnection( connectionString );

// Connecting to the database, specifying explicitly a user name and a password
std::auto_ptr<pool::IRelationalSession> session( domain.newSession( connectionString ) );
session->connectAsUser(        "my_username",        "my_password"        );

//                   Starting                    a                   transaction
session->transaction().start();
.
.
.
//                 Committing                 the                 transaction
session->transaction().commit();

//                                                              Disconnecting
session->disconnect();
.
.
.
```

14.4. Managing schemas

14.4.1. Schemas, tables, descriptors and editors

Once connected to a relational database, the user works with the data defined under the schema defined for the table collection (MySQL database, Oracle user schema, SQLite file, etc.) specifyied in the connection string. The **IRelationalSchema** interface which is retrieved by the IRelationalSession object allows the user to list, create and drop tables in the working schema.

The IRelationalSchema can be used to retrieve a reference to an **IRelationalTable** object specifying its name. This interface allows the user to

- retrieve the description of the corresponding table via the **IRelationalTableDescription** interface
- set the access privileges using the **IRelationalTablePrivilegeManager** interface
- alter the table definition using the **IRelationalTableSchemaEditor** interface
- perform data manipulation on the table through the **IRelationalTableDataEditor** interface
- execute queries with the data of the table

The IRelationalTableDescription object allows the user to retrieve the following information from a table:

- The column names and types (the C++ equivalents)

- The NULLness and the UNIQUEness of a column
- The definition (column names) of its primary key, in case there is one, using the **IRelationalPrimaryKey** interface
- The definitions (key name, column names, referenced table, referenced columns) of its foreign keys using the **IRelationalForeignKey** interface
- The definitions (column names, uniqueness) of its indices using the **IRelationalIndex** interface

In order to create a new table in the schema the user has to construct an IRelationalTableDescription object and pass it to the IRelationalSchema. The component provides an implementation of the **IRelationalEditableTableDescription** interface for this purpose.

By default, for most technologies, a table is created without any access right to anybody but the table owner. It is therefore a good practice for tables that will be exposed to other database users to use the IRelationalTablePrivilegeManager interface to grant the relevant access rights immediatelly after the table creation.

Note that the semantics of the interfaces assume that the names of the tables, columns, keys and indices are case sensitive. Given though that some databases do not preserve the case, it is highly recommended to capitalize all such system names, especially if the RAL is expected to be used for cross-populating databases of different technologies.

14.4.2. Example of retrieving the schema definition

```
.
.
.
#include                                "RelationalAccess/IRelationalSchema.h"
#include                                 "RelationalAccess/IRelationalTable.h"
#include             "RelationalAccess/RelationalEditableTableDescription.h"
#include                              "RelationalAccess/IRelationalPrimaryKey.h"
#include                              "RelationalAccess/IRelationalForeignKey.h"
#include                                   "RelationalAccess/IRelationalIndex.h"
#include                                        "AttributeList/AttributeList.h"
.
.
.
std::set<  std::string  >  tableNames  =  session->userSchema().listTables();
std::cout    <<    "Tables    in    schema    :    "    <<    std::endl;
for ( std::set< std::string >::const_iterator iTableName = tableNames.begin();
         iTableName    !=    tableNames.end();    ++iTableName    )    {
     std::cout    <<    "      "    <<    *iTableName    <<    std::endl;
  const pool::IRelationalTable& table = session->userSchema().tableHandle( *iTableNam
  const pool::IRelationalTableDescription& tableDescription = table.description();
  const pool::AttributeListSpecification& columnNamesAndTypes = tableDescription.colu
  for ( pool::AttributeListSpecification::const_iterator iColumn = columnNamesAndType
           iColumn    !=    columnNamesAndTypes.end();    ++iColumn    )    {
    std::cout << "       " << iColumn->name() << " : " << iColumn->type_name();
       if    (    tableDescription.isNotNull(    iColumn->name()    )    )    {
                std::cout          <<          "          NOT          NULL";
                                                                              }
       if    (    tableDescription.isUnique(    iColumn->name()    )    )    {
                    std::cout             <<             "             UNIQUE";
                                                                              }
                       std::cout                  <<                  std::endl;
                                                                              }
         if       (       tableDescription.hasPrimaryKey()       )       {
    const pool::IRelationalPrimaryKey& primaryKey = tableDescription.primaryKey();
     std::cout   <<   "         ...   has   primary   key   for   column(s)   :";
    for ( std::vector< std::string >::const_iterator iColumn = primaryKey.columns().b
            iColumn    !=    primaryKey.columns().end();    ++iColumn    )    {
             std::cout        <<        "        "        <<        *iColumn;
                                                                              }
                       std::cout                  <<                  std::endl;
                                                                              }

      int     numberOfIndices     =     tableDescription.numberOfIndices();
   for  (  int  iIndex  =  0;  iIndex  <  numberOfIndices;  ++iIndex  )  {
```

```
    const pool::IRelationalIndex& index = tableDescription.index( iIndex );
          std::cout       <<       "              ...       defined       ";
       if    (    index.isUnique()    )    std::cout    <<    "unique    ";
     std::cout  <<  "index  \""  <<  index.name()  <<  "\"  for  column(s)  :";
    for ( std::vector< std::string >::const_iterator iColumn = index.columns().begin();
             iColumn    !=    index.columns().end();    ++iColumn    )    {
             std::cout       <<       "        "       <<       *iColumn;
                                                                    }
                    std::cout                <<                std::endl;
                                                                    }

    int   numberOfForeignKeys   =   tableDescription.numberOfForeignKeys();
  for ( int iForeignKey = 0; iForeignKey < numberOfForeignKeys; ++iForeignKey ) {
    const pool::IRelationalForeignKey& foreignKey = tableDescription.foreignKey( iForeignKe
    std::cout << "       ... defined foreign key \"" + foreignKey.name() + "\" for columns(
    for ( std::vector< std::string >::const_iterator iColumn = foreignKey.columns().begin()
             iColumn   !=   foreignKey.columns().end();   ++iColumn    )    {
             std::cout       <<       "        "       <<       *iColumn;
                                                                    }
    std::cout << " referencing table \"" + foreignKey.referencedTable() + "\" for columns(s
    for ( std::vector< std::string >::const_iterator iColumn = foreignKey.referencedColumns
          iColumn  !=  foreignKey.referencedColumns().end();  ++iColumn  )  {
             std::cout       <<       "        "       <<       *iColumn;
                                                                    }
                    std::cout                <<                std::endl;
                                                                    }
}
.
.
.
```

14.4.3. Example of creating a new table

```
.
.
.
#include             "RelationalAccess/RelationalEditableTableDescription.h"
#include               "RelationalAccess/IRelationalTablePrivilegeManager.h"
#include                                      "SealKernel/MessageStream.h"
.
.
.
seal::MessageStream log( pool::POOLContext::context(), "UserSchema_Test" );
.
.
.
session->userSchema().dropIfExistsTable(         "MySimpleTable2"         );
session->userSchema().dropIfExistsTable(         "MySimpleTable1"         );

//     Creating     the     description     for     the     first     table
std::auto_ptr< pool::IRelationalEditableTableDescription > description1( new pool::Relation

description1->insertColumn( "id", pool::AttributeStaticTypeInfo<int>::type_name() );
description1->insertColumn( "x", pool::AttributeStaticTypeInfo<float>::type_name() );
description1->insertColumn( "y", pool::AttributeStaticTypeInfo<double>::type_name() );
description1->insertColumn( "z", pool::AttributeStaticTypeInfo<double>::type_name() );
description1->setNotNullConstraint(                  "id"                  );
description1->setNotNullConstraint(                  "x"                   );
description1->setNotNullConstraint(                  "y"                   );
description1->setNotNullConstraint(                  "z"                   );
description1->setUniqueConstraint(                   "z"                   );
description1->setPrimaryKey(  std::vector<  std::string  >(  1,  "id"  )  );

//      Create      a      unique      index      for      x      and      y
std::vector<            std::string             >            idxColumns(2);
idxColumns[0]                                =                                "x";
idxColumns[1]                                =                                "y";
description1->createIndex(   "MySimpleTable1_IDX",   idxColumns,   true   );
//       Create       an       index       for       y       and       z
idxColumns[0]                                =                                "y";
idxColumns[1]                                =                                "z";
description1->createIndex(     "MySimpleTable1_idx2",     idxColumns     );

//             Create             the             first             table
std::cout   <<   "Creating   table   \"MySimpleTable1\""   <<   std::endl;
```

```
pool::IRelationalTable& table1 = session->userSchema().createTable( "MySimpleTable1",

//          Grant          read          access          to          everybody
table1.privilegeManager().grantToPublic( pool::IRelationalTablePrivilegeManager::SELE

//     Creating     the     description     for     the     second     table
std::auto_ptr< pool::IRelationalEditableTableDescription > description2( new pool::Re

description2->insertColumn( "fid", pool::AttributeStaticTypeInfo<int>::type_name() );
description2->insertColumn( "tx", pool::AttributeStaticTypeInfo<float>::type_name() )
description2->insertColumn( "ty", pool::AttributeStaticTypeInfo<double>::type_name()
description2->setNotNullConstraint(                  "fid"                 );
description2->setNotNullConstraint(                  "tx"                  );
description2->setNotNullConstraint(                  "ty"                  );

//       Create       an       index       for       tx       and       ty
std::vector<            std::string            >            idxColumns2(2);
idxColumns2[0]                              =                           "tx";
idxColumns2[1]                              =                           "ty";
description2->createIndex(     "MySimpleTable2_IDX",     idxColumns2     );

//         Make         fid         be         a         foreign         key
std::vector<     std::string     >     fkColumns(     1,     "fid"     );
std::vector<     std::string     >     fkRColumns(     1,     "id"     );
description2->createForeignKey(                          "MySimpleTable2_FK",
                                                                 fkColumns,
                                                        "MySimpleTable1",
                                                fkRColumns                );

//             Create             the             second             table
std::cout   <<   "Creating   table   \"MySimpleTable2\""   <<   std::endl;
pool::IRelationalTable& table2 = session->userSchema().createTable( "MySimpleTable2",

//          Grant          read          access          to          everybody
table2.privilegeManager().grantToPublic( pool::IRelationalTablePrivilegeManager::SELE

std::cout << "Successfully created tables \"" << table1.name() << "\" and \"" << tabl

.
.
.
```

14.5. Performoming data manipulation

14.5.1. Supported operations

The IRelationalTableDataEditor allows the privileged user perform DML operations on a table. In particular it is allowed to

- Insert a new row. The input row is fed with an AttributeList with an AttributeListSpecification that is a subset of the table's columns and types.
- Modify existing rows. In this case the SET and WHERE clauses of the corresponding SQL statement have to be provided by the user. The user is recommended to use bind variables for the WHERE and SET clauses which can be passed through an AttributeList.
- Delete existing rows. In this case the WHERE clause of the corresponding SQL statement have to be provided by the user. The user is recommended to use bind variables for the WHERE clause.

For fast row insertions with the minimal number of roundtrips to the database server the user is recommended to use the **IRelationalBulkRowInserter** interface. In this case the user specifies the row buffer to be bound and the number of rows to be cached at the client side before the data are sent to the server.

14.5.2. Example DML

```
.
.
.
#include                    "RelationalAccess/IRelationalTableDataEditor.h"
.
.
.
std::auto_ptr< pool::IRelationalEditableTableDescription > description( new pool::Relationa

description->insertColumn( "id", pool::AttributeStaticTypeInfo<int>::type_name() );
description->insertColumn( "x", pool::AttributeStaticTypeInfo<float>::type_name() );
description->insertColumn( "y", pool::AttributeStaticTypeInfo<double>::type_name() );
description->insertColumn( "comments", pool::AttributeStaticTypeInfo<std::string>::type_nam

//                   Create                    the                    table.
std::cout    <<    "Creating    table    \"DataTable\""    <<    std::endl;
pool::IRelationalTable& table = session->userSchema().createTable( "DataTable", *descriptio

//             Costructing             a             row             buffer.
pool::AttributeList   data(   table.description().columnNamesAndTypes()   );

//            Retrieving            the            editor            object.
pool::IRelationalTableDataEditor&    dataEditor    =    table.dataEditor();

//                    Adding                      new                      rows
std::cout   <<   "Adding   five   rows   into   the   table"   <<   std::endl;
data["id"].setValue<int>(                        1                        );
data["x"].setValue<float>(                      1.1                       );
data["y"].setValue<double>(                     1.11                      );
data["comments"].setValue<std::string>(      "A      Table      Row"      );

dataEditor.insertNewRow(                       data                       );

data["id"].setValue<int>(                        2                        );
data["x"].setValue<float>(                      2.2                       );
data["y"].setValue<double>(                     2.22                      );
data["comments"].setValue<std::string>(  "A  second  row  in  the  table"  );

dataEditor.insertNewRow(                       data                       );

data["id"].setValue<int>(                        3                        );
data["x"].setValue<float>(                      3.3                       );
data["y"].setValue<double>(                     3.33                      );
data["comments"].setValue<std::string>(   "This   is   the   third   row."   );

dataEditor.insertNewRow(                       data                       );

data["id"].setValue<int>(                        4                        );
data["x"].setValue<float>(                      4.4                       );
data["y"].setValue<double>(                     4.44                      );
data["comments"].setValue<std::string>(     "Row     number     four."     );

dataEditor.insertNewRow(                       data                       );

data["id"].setValue<int>(                        5                        );
data["x"].setValue<float>(                      5.5                       );
data["y"].setValue<double>(                     5.55                      );
data["comments"].setValue<std::string>(      "Row      five      !!!"      );

dataEditor.insertNewRow(                       data                       );

//                   Deleting                     some                     rows.
std::cout   <<   "Deleting   rows   from   the   table."   <<   std::endl;
pool::AttributeListSpecification                                  spec;
spec.push_back( "idValue", pool::AttributeStaticTypeInfo<int>::type_name() );
spec.push_back( "xValue", pool::AttributeStaticTypeInfo<float>::type_name() );
pool::AttributeList              inputData(                spec                );
inputData["idValue"].setValue<int>(                    4                     );
inputData["xValue"].setValue<float>(                  1.1                    );
long numberOfRowsDeleted = dataEditor.deleteRows( "id < :idValue and x>:xValue",
                                                  inputData        );
std::cout << "Deleted " << numberOfRowsDeleted << " rows from the table." << std::endl;

//                   Updating                     some                     rows
std::cout   <<   "Updating   some   rows   in   the   table."   <<   std::endl;
pool::AttributeListSpecification                                 specForUpdate;
specForUpdate.push_back( "newx", pool::AttributeStaticTypeInfo<float>::type_name() );
```

```
specForUpdate.push_back( "newcomments", pool::AttributeStaticTypeInfo<std::string>::t
specForUpdate.push_back( "idValue", pool::AttributeStaticTypeInfo<int>::type_name() )
specForUpdate.push_back( "xValue", pool::AttributeStaticTypeInfo<float>::type_name()
pool::AttributeList         dataForUpdate(         specForUpdate         );
dataForUpdate["newx"].setValue<float>(                0                 );
dataForUpdate["newcomments"].setValue<std::string>(    "Updated    row"    );
dataForUpdate["idValue"].setValue<int>(                2                 );
dataForUpdate["xValue"].setValue<float>(               5.1               );
long numberOfRowsUpdated = dataEditor.updateRows( "x = :newx, comments = :newcomments
                                                  "id<:idValue or x > :xValue",
                                                      dataForUpdate      );
std::cout << "Updated " << numberOfRowsUpdated << " rows in the table." << std::endl;

std::cout << "There are currently " << table.numberOfRows() << " rows in table \"" <<

//          Fast          inserting          a          few          rows
pool::IRelationalBulkRowInserter& rowInserter = table.dataEditor().bulkRowInserter();
rowInserter.setup(                data,               10                );

for     (     int     i     =     0;     i     <     15;     ++i     )     {
          data["id"].setValue<int>(         i          +          1          );
     data["x"].setValue<float>(    (    i    +    1    )    *    1.1    );
     data["y"].setValue<double>(    (    i    +    1    )    *    1.11    );
                           std::ostringstream                          os;
        os       <<       "Row       "       <<       i       +       1;
          data["comments"].setValue<std::string>(         os.str()         );
                                                rowInserter.insertNewRow();
}

rowInserter.flushCache();
.
.
.
```

14.6. Issuing queries

14.6.1. Queries using a single table

The IRelationalTable interface acts as a factory of **IRelationalQuery** for queries involving the corresponding table. A query is formed by

- *Specifying the output variables*. If no variable is specified a wildcard query is executed. For each variable an alias can be specified. The return type of the output variables or expressions is internally deduced from the table description. The user may fully redefine the output (name and type) by specifying the AttributeListSpecification which is to be used for accessing the result set.
- *Defining the query condition*. The user has to specify the WHERE clause of the corresponding SQL statement. It is highly recommended to use bind variables passing the corresponding AttributeList.
- *Specifying order variables*. By doing so the user instructs the system to append to the SQL query an ORDER BY clause.
- *Limiting the number rows in the result set*. This is a very useful operation when the maximum number of rows that will be retrieved at a query is known.
- *Defining the number of prefetched rows or the size of the buffer on the client side for the rows if the result set*. This operation can be used to reduce the number of roundtrips to the server for fetching the result of a query.

On execution the IRelationalQuery returns a reference to an **IRelationalCursor** object, which acts as an iterator for the result set. This interface is used to access the data of the result providing also information on the NULLness of a variable in a given row.

14.6.2. Example of a query on a table

```
.
.
.
#include                                "RelationalAccess/IRelationalQuery.h"
#include                               "RelationalAccess/IRelationalCursor.h"
.
.
.
//  Querying  :  SELECT  *  FROM  DataTable  WHERE  id  >  12  AND  x  <  30
std::auto_ptr<  pool::IRelationalQuery  >  query1(  table.createQuery()  );
query1->setRowCacheSize(                          10                      );
pool::AttributeList                                    emptyBindVariableList;
query1->setCondition(  "id  >  12  AND  x  <  30",  emptyBindVariableList  );
pool::IRelationalCursor&        cursor1        =         query1->process();
if              (              cursor1.start()             )              {
              while(             cursor1.next()             )              {
       const     pool::AttributeList&     row     =     cursor1.currentRow();
     for  (  pool::AttributeList::const_iterator  iColumn  =  row.begin();
              iColumn      !=      row.end();      ++iColumn      )      {
      std::cout << iColumn->spec().name() << " : " << iColumn->getValueAsString() << "\t";
                                                                            }
                    std::cout                 <<                 std::endl;
                                                                            }
}
std::cout  <<  cursor1.numberOfRows()  <<  "  row(s)  selected."  <<  std::endl;

// Querying : SELECT comments, y AS The_Y FROM DataTable WHERE id > :idValue AND x < :xValu
std::auto_ptr<  pool::IRelationalQuery  >  query2(  table.createQuery()  );
query2->addToOutputList(                        "comments"                  );
query2->addToOutputList(                       "y",                  "The_Y");
query2->setMemoryCacheSize(                           1                     );
pool::AttributeListSpecification                           bindVariableListSpec;
bindVariableListSpec.push_back<int>(                "idValue"               );
bindVariableListSpec.push_back<float>(                "xValue"              );
pool::AttributeList      bindVariableList(      bindVariableListSpec      );
query2->setCondition( "id > :idValue AND x < :xValue", bindVariableList );
std::cout     <<     "Setting     idValue     to     21"     <<     std::endl;
bindVariableList["idValue"].setValue<int>(                    21              );
std::cout     <<     "Setting     xValue     to     27"     <<     std::endl;
bindVariableList["xValue"].setValue<float>(                   27              );
pool::IRelationalCursor&        cursor2        =         query2->process();
if              (              cursor2.start()             )              {
              while(             cursor2.next()             )              {
       const     pool::AttributeList&     row     =     cursor2.currentRow();
     for  (  pool::AttributeList::const_iterator  iColumn  =  row.begin();
              iColumn      !=      row.end();      ++iColumn      )      {
      std::cout << iColumn->spec().name() << " : " << iColumn->getValueAsString() << "\t";
                                                                            }
                    std::cout                 <<                 std::endl;
                                                                            }
}
std::cout  <<  cursor2.numberOfRows()  <<  "  row(s)  selected."  <<  std::endl;
.
.
.
```

14.6.3. More general queries involving several tables

In more realisting cases, a user will perform more general queries which involve more than one table. For this case the IRelationalQuery is extended to the **IRelationalQueryWithMultipleTables** interface. The IRelationalSchema interface acts as the factory for such objects. The extended interface allows the user to:

- *Add tables to the selection list.* The user may define an alias for the table names. This is a usefull operation when for example one attempts an self inner join of a table.
- *Define a sub-query*. The user assigns an alias name for the sub-query whose result set can be used as the input table in the parent query. Sub-queries are handled via the **IRelationalSubQuery** interface and they can have an arbitrary level of depth.

14.6.4. Example of a query involving several tables

```
.
.
.
#include            "RelationalAccess/IRelationalQueryWithMultipleTables.h"
#include                          "RelationalAccess/IRelationalSubQuery.h"
.
.
.
// Performing the query SELECT Offices.Orientation, Departments.Name FROM Offices, De

std::auto_ptr< pool::IRelationalQueryWithMultipleTables > query1( session->userSchema
query1->addToOutputList(                                "Offices.Orientation");
query1->addToOutputList(                                   "Departments.Name");
query1->addToTableList(                                             "Offices");
query1->addToTableList(                                         "Departments");
pool::AttributeList                                    emptyBindVariableList;
query1->setCondition( "Offices.Department = Departments.id", emptyBindVariableList );
query1->addToOrderList(              "Offices.Orientation"                     );
query1->setRowCacheSize(                          10                           );
query1->limitReturnedRows(                         3                           );
pool::IRelationalCursor&        cursor1        =        query1->process();
if              (              cursor1.start()              )                  {
              while(             cursor1.next()              )                  {
       const     pool::AttributeList&     row     =     cursor1.currentRow();
     for   (   pool::AttributeList::const_iterator   iColumn   =   row.begin();
               iColumn      !=      row.end();      ++iColumn      )      {
      std::cout << iColumn->spec().name() << " : " << iColumn->getValueAsString() <<
                                                                              }
                    std::cout                  <<                 std::endl;
                                                                              }
}
std::cout  <<  cursor1.numberOfRows()  <<  "  row(s)  selected."  <<  std::endl;


// Performing the query SELECT Personnel.Name, SelectedOffices.Orientation FROM ( SEL

std::auto_ptr< pool::IRelationalQueryWithMultipleTables > query2( session->userSchema
pool::IRelationalSubQuery& subquery = query2->defineSubQuery( "SelectedOffices" );
subquery.addToOutputList(              "Offices.Orientation"                    );
subquery.addToOutputList(        "Offices.id",         "office_id"             );
subquery.addToTableList(                        "Offices"                      );
subquery.addToTableList(                      "Departments"                    );
pool::AttributeListSpecification                                  bindSpec;
bindSpec.push_back<std::string>(                   "depname"                   );
pool::AttributeList          bindVariableList(          bindSpec               );
bindVariableList["depname"].setValue<std::string>(         "Dep2"              );
subquery.setCondition( "Offices.Department = Departments.id AND Departments.Name = :d
query2->addToOutputList(                  "Personnel.Name"                     );
query2->addToOutputList(           "SelectedOffices.Orientation"               );
query2->addToTableList(                 "SelectedOffices"                      );
query2->addToTableList(                    "Personnel"                         );
query2->setCondition( "Personnel.Office = SelectedOffices.office_id", emptyBindVariab
query2->addToOrderList(            "SelectedOffices.Orientation"               );
query2->setRowCacheSize(                          10                           );
pool::IRelationalCursor&        cursor2        =        query2->process();
if              (              cursor2.start()              )                  {
              while(             cursor2.next()              )                  {
       const     pool::AttributeList&     row     =     cursor2.currentRow();
     for   (   pool::AttributeList::const_iterator   iColumn   =   row.begin();
               iColumn      !=      row.end();      ++iColumn      )      {
      std::cout << iColumn->spec().name() << " : " << iColumn->getValueAsString() <<
                                                                              }
                    std::cout                  <<                 std::endl;
                                                                              }
}
std::cout  <<  cursor2.numberOfRows()  <<  "  row(s)  selected."  <<  std::endl;
.
.
.
```

15. RelationalAccess component: Reference Manual

15.1. Signatures of public interfaces

- *IRelationalService* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalService.html
- *IRelationalDomain* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalDomain.html
- *IRelationalTypeConverter* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalTypeConverter.html
- *IRelationalSession* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalSession.html
- *IRelationalTransaction* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalTransaction.html
- *IAuthenticationService* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IAuthenticationService.html
- *IRelationalSchema* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalSchema.html
- *IRelationalTable* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalTable.html
- *IRelationalTablePrivilegeManager* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalTablePrivilegeManager.html
- *IRelationalTableDescription* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalTableDescription.html
- *IRelationalPrimaryKey* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalPrimaryKey.html
- *IRelationalForeignKey* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalForeignKey.html
- *IRelationalIndex* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalIndex.html
- *IRelationalTableSchemaEditor* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalTableSchemaEditor.html
- *IRelationalTableIndexEditor* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalTableIndexEditor.html
- *IRelationalEditableTableDescription* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalEditableTableDescription.html
- *IRelationalTableDataEditor* : ht-

tp://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalTableDataEditor.html

- *IRelationalBulkRowInserter* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalBulkRowInserter.html
- *IRelationalQuery* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalQuery.html
- *IRelationalCursor* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalCursor.html
- *IRelationalQueryWithMultipleTables* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalQueryWithMultipleTables.html
- *IRelationalSubQuery* : http://lcgapp.cern.ch/doxygen/POOL/POOL_1_7_0/doxygen/classpool_1_1IRelationalSubQuery.html