POOL 2.6.0 User guide

13 July 2007

Table of Contents

1. Introduction *1*
2. POOL by examples *3*
3. POOL Architecture *3*
4. POOLCore: User level semantics *6*
5. POOLCore: Reference Manual *6*
6. AttributeList: User level semantics *6*
7. AttributeList: Reference Manual *9*
8. FileCatalog: User level semantics *9*
9. FileCatalog: Reference Manual *21*
10. XMLCatalog: User level semantics *21*
11. XMLCatalog: Reference Manual *21*
12. MySQLCatalog: User level semantics *22*
13. MySQLCatalog: Reference Manual *22*
14. LFCCatalog: User level semantics *22*
15. LFCCatalog: Reference Manual *22*
16. DataSvc: User level semantics *22*
17. DataSvc: Reference Manual *32*
18. PersistencySvc: User level semantics *42*
19. PersistencySvc: Reference Manual *46*
20. StorageSvc: User level semantics *47*
21. StorageSvc: Reference Manual *49*
22. Collection: User level semantics *50*
23. Collection: Reference Manual *60*
24. ObjectRelationalAccess: User level semantics *70*
25. ObjectRelationalAccess: Reference Manual *71*
26. RelationalStorageService: User level semantics *74*
27. RelationalStorageService: Reference Manual *75*
28. RelationalCollection: User level semantics *75*
29. RelationalCollection: Reference Manual *75*

About this document

The POOL project [http://pool.cern.ch] has released version 2.6.0 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 institutes.

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.

1.5. Relational Abstraction Layer

As of release 1.7.0 the POOL framework provides in addition to the above a software abstraction layer for accessing relational databases. The purpose of this abstraction layer is to shield the user from the technology specific APIs removing at the same time the need to submit directly SQL commands. Techonology-specific implementations of the interfaces are packaged as plugin libraries which are loaded at run time. This architecture has allowed the POOL team to develop File Catalog, Collection and Storage Manager implementations based on this generic interfaces, allowing thus the exploitation of several database back-ends.

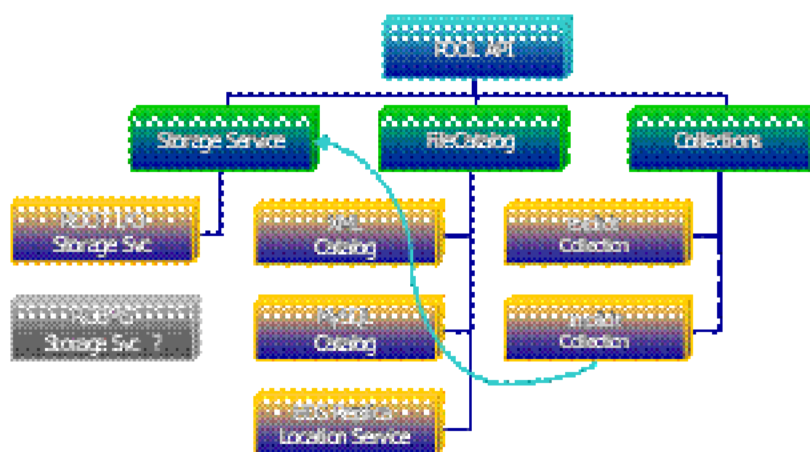
Note that as of POOL 2.2.7 and with the introduction of CORAL, the implementations based on RAL are now based on CORAL. As of POOL 2.3.0 the RAL packages have become obsolete.

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].

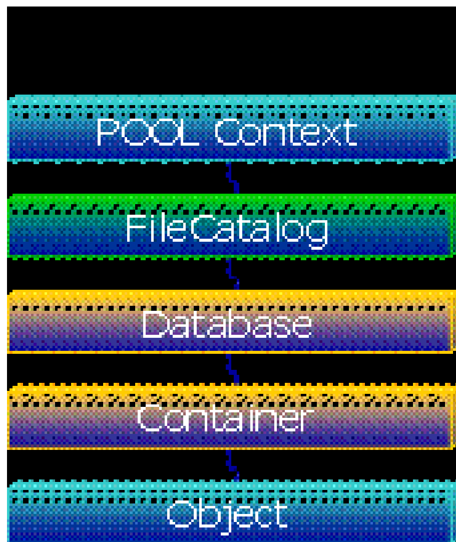


POOL breakdown in components

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

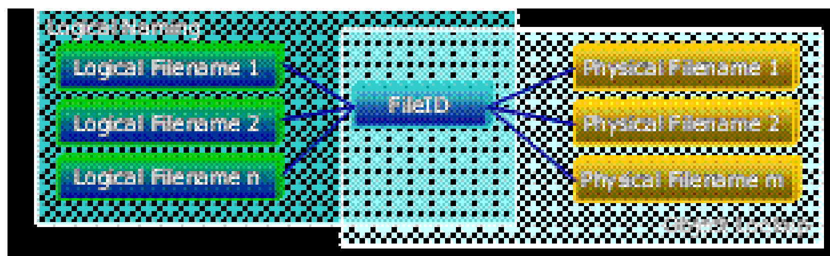
The main responsibility of the File Catalog is to keep track of all POOL databases (usually files that store objects) and to resolve file references into physical file names, which are then used by lower level components, such as the storage service, to access file contents. More recently, the POOL file 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. POOLCore: User level semantics

4.1. Example of usage

5. POOLCore: Reference Manual

5.1. Command line tools

`pool_gen_uuid`
Creates and prints a new GUID.

6. AttributeList: User level semantics

6.1. 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();
```

6.2. 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.

6.3. 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.

6.4. 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.

6.5. The difference between AttributeList and AttributeListSpecification

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.

6.6. 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.

6.7. 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).

6.8. 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.

6.9. 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 typeless way of setting attributes see the section "Converting Attributes to strings".

7. AttributeList: Reference Manual

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

8. FileCatalog: User level semantics

8.1. Public classes UML diagram

8.2. 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.

8.3. 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

  Relational FileCatalog:

  relationalcatalog_sqlite://mycatalog.db

  relationalcatalog_oracle://devdb9/username

8.4. 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%'".

8.5. 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 environ-

ment 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.

   -t file type

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] [-g guid] -r replica [-u uri -h]

   Register a replica file PFN (specified by **-r** option) to the master file PFN (specified by **-p** option) or directly to the Guid of the master file specified by the **-g** 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).

8.6. 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 read/writable and 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.

8.7. 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();
```

8.8. 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

8.9. 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.

8.10. 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.

8.10.1. 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

9. FileCatalog: Reference Manual

9.1. Signatures of public interfaces

[Method/code fragment]

9.2. Command line tools

[Method/code fragment]

10. XMLCatalog: User level semantics

10.1. Public classes UML diagram

10.2. [Component specific section]

10.3. Example of usage

[Method/code fragment]

11. XMLCatalog: Reference Manual

11.1. Signatures of public interfaces

[Method/code fragment]

11.2. Command line tools

[Method/code fragment]

12. MySQLCatalog: User level semantics

12.1. Public classes UML diagram

12.2. [Component specific section]

12.3. Example of usage

[Method/code fragment]

13. MySQLCatalog: Reference Manual

13.1. Signatures of public interfaces

[Method/code fragment]

13.2. Command line tools

[Method/code fragment]

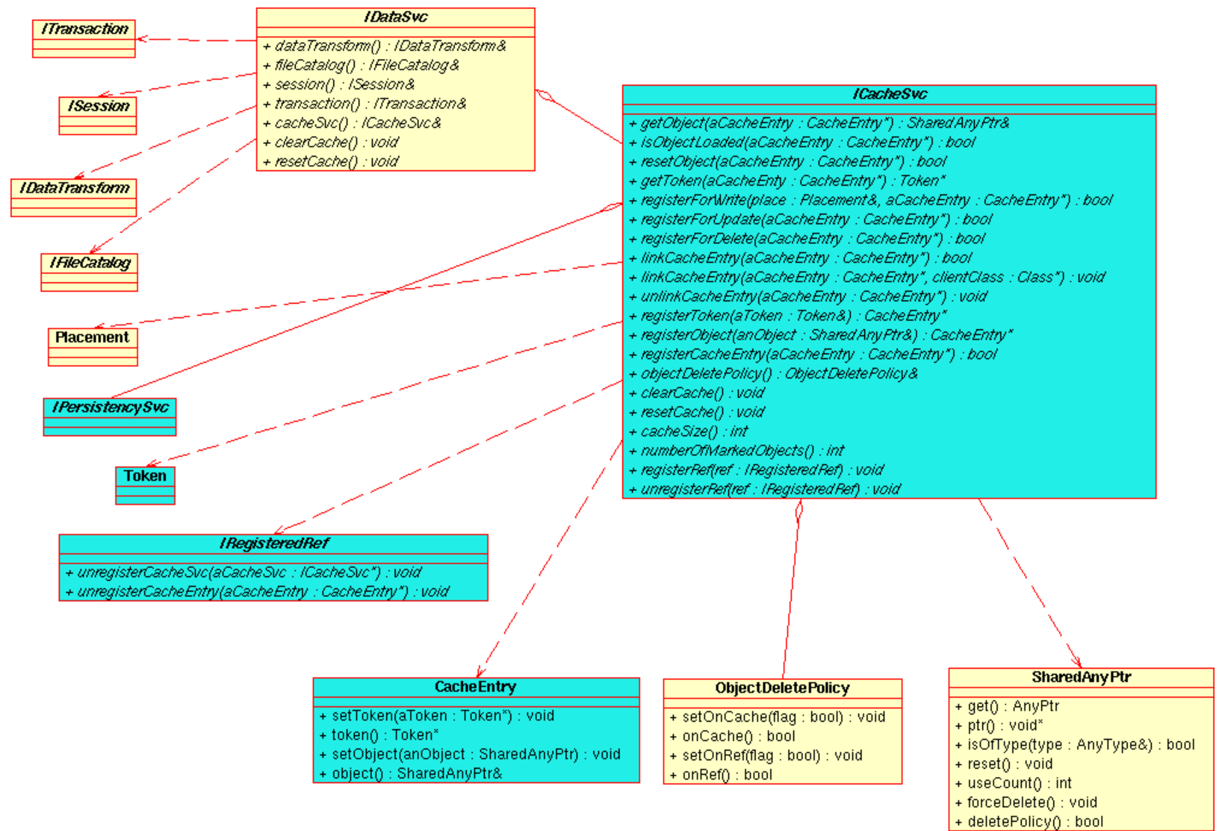
14. LFCCatalog: User level semantics

15. LFCCatalog: Reference Manual

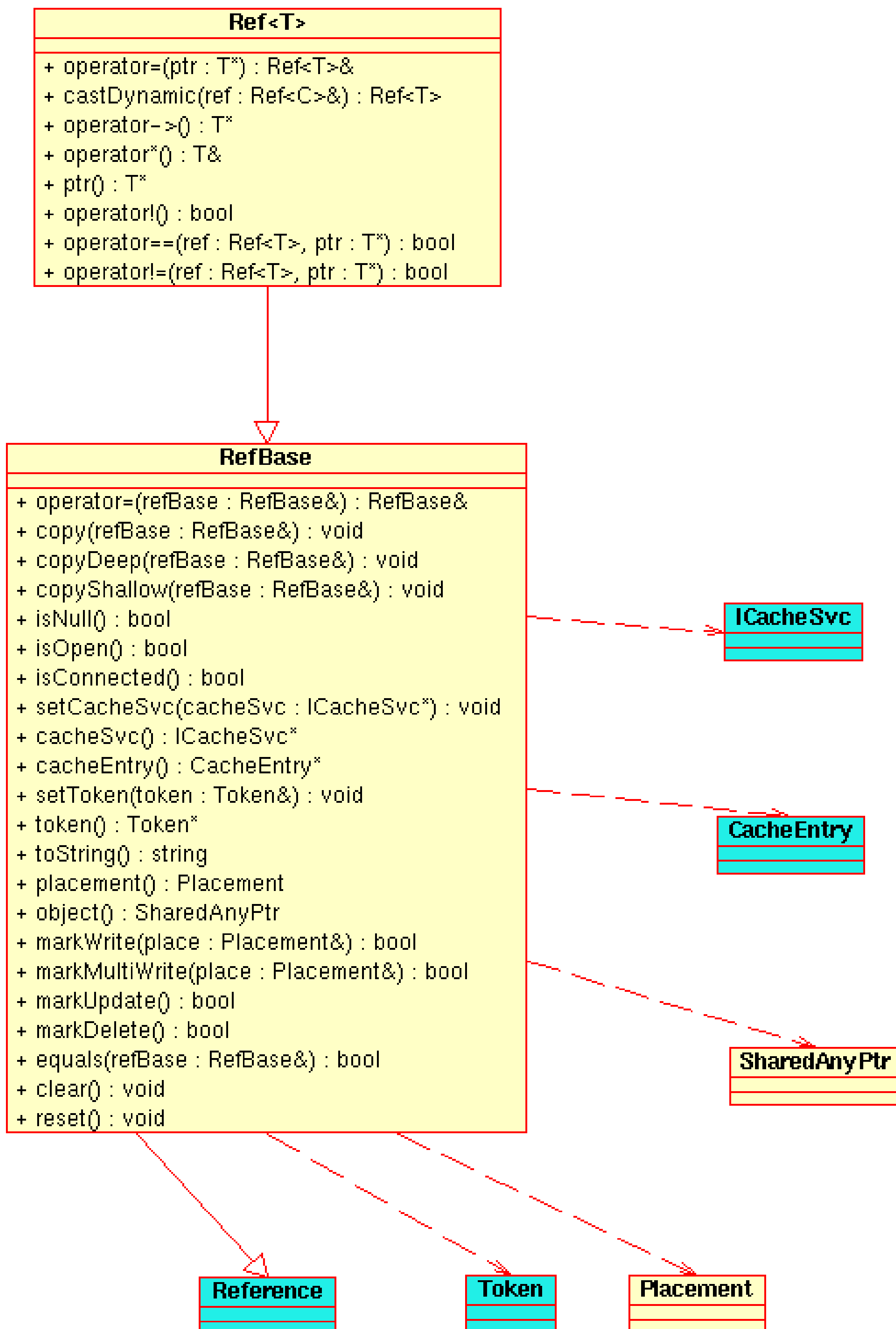
16. DataSvc: User level semantics

16.1. 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

16.2. 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.

16.2.1. 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 definining 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.

16.2.2. 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.

16.2.3. 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.

16.2.4. 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.

16.2.5. 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.

16.2.6. 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.

16.3. Example of usage

16.3.1. 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);
```

16.3.2. 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();
```

16.3.3. 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 propaga
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 d
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.

16.3.4. 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-de
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!
```

17. DataSvc: Reference Manual

17.1. class Ref<T>

17.1.1. 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).

17.1.2. 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)

17.1.3. 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.

17.1.4. 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.

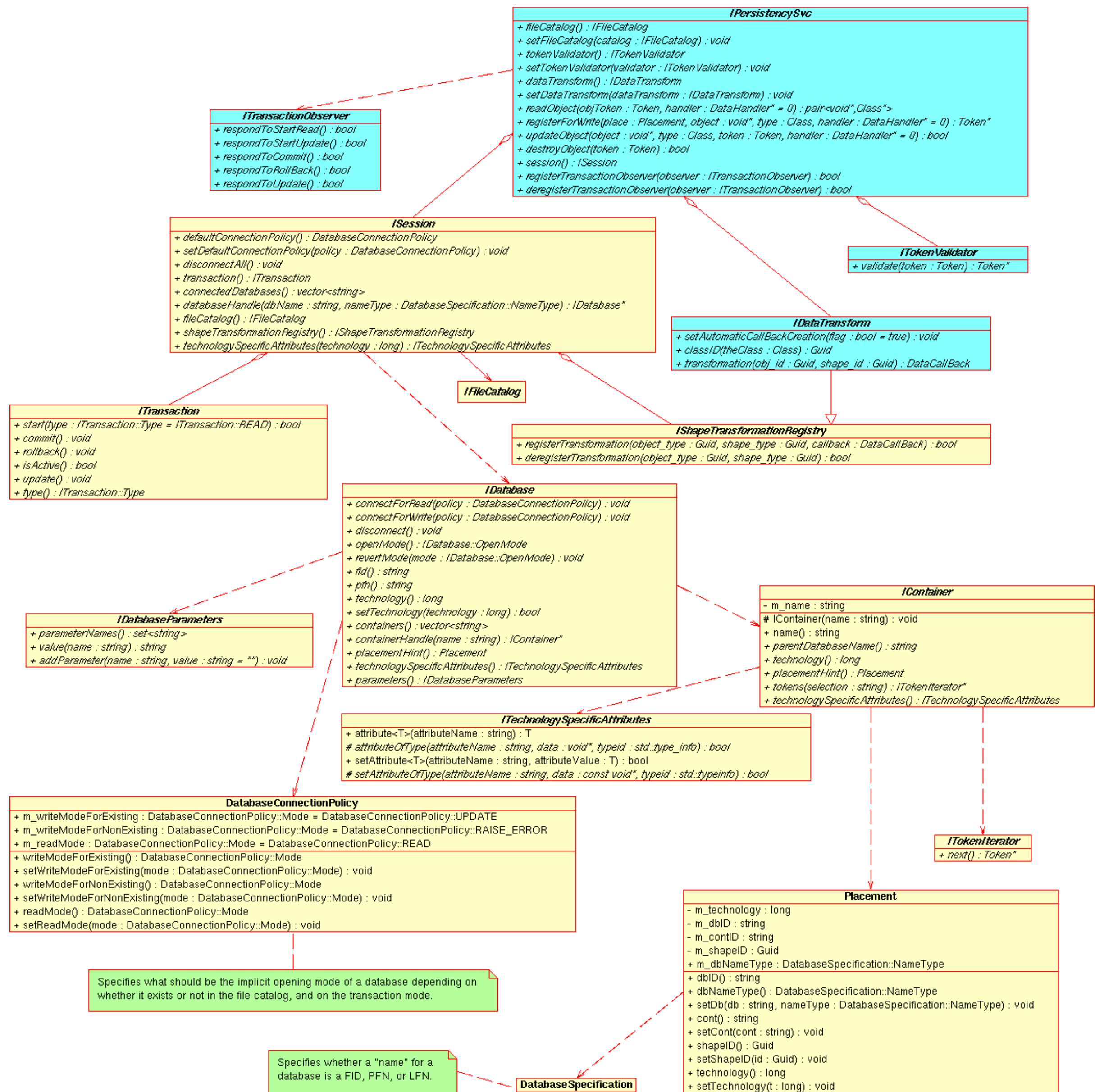
- Cache bound ref: false if no token is assigned, or the associated cache entry is already marked for write or update. True otherwise.

Throws: nothing.

18. PersistencySvc: User level semantics

18.1. 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

18.2. 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.

18.3. 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.

18.4. Implicit and explicit opening of databases

18.4.1. 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.

18.4.2. 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.

18.5. 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.

18.6. Customizable operations

18.6.1. 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. *IShapeTransformationRegistryIDataTransform*

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.

18.6.2. 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.

18.7. Example of usage

Creating an IPersistencySvc object (done by DataSvc)

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

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::DatabaseSpecificatio
                                                   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::DatabaseSpecificatio
 if  (db->openMode()  ==  pool::IDatabase::CLOSED  ) db->connectForRead();
       const        std::string&        fid        =        db->fid();
unsigned int sizeInkB = db->technologySpecificAttributes().attribute<int>( "FILE_SIZE"
                                                   transaction.commit();
```

Further examples of usage can be considered:

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

19. PersistencySvc: Reference Manual

19.1. Signatures of public interfaces

19.1.1. User-level interfaces

- *ISession* : http://pool.cern.ch/currentReleaseDoc/classReference/html/classpool_1_1ISession.html
- *ITransaction* : http://pool.cern.ch/currentReleaseDoc/classReference/html/classpool_1_1ITransaction.html
- *IShapeTransformationRegistry* : http://pool.cern.ch/currentReleaseDoc/classReference/html/classpool_1_1IShapeTransformationRegistry.html
- *IDatabase* : http://pool.cern.ch/currentReleaseDoc/classReference/html/classpool_1_1IDatabase.html [http://pool.cern.ch/currentReleaseDoc/classReference/html/classpool_1_1IIDatabase.html]
- *IDatabaseParameters* : http://pool.cern.ch/currentReleaseDoc/classReference/html/classpool_1_1IDatabaseParameters.html

- *DatabaseConnectionPolicy* : http://pool.cern.ch/currentReleaseDoc/classReference/html/classpool_1_1DatabaseConnectionPolicy.html

- *IContainer* : http://pool.cern.ch/currentReleaseDoc/classReference/html/classpool_1_1IContainer.html

- *ITechnologySpecificAttributes* : http://pool.cern.ch/currentReleaseDoc/classReference/html/classpool_1_1ITechnologySpecificAttributes.html

- *Placement* : http://pool.cern.ch/currentReleaseDoc/classReference/html/classpool_1_1Placement.html

- *ITokenIterator* : http://pool.cern.ch/currentReleaseDoc/classReference/html/classpool_1_1ITokenIterator.html

19.1.2. Developer-level interfaces

- *IPersistencySvc* : http://pool.cern.ch/currentReleaseDoc/classReference/html/classpool_1_1IPersistencySvc.html

- *ITransactionObserver* : http://pool.cern.ch/currentReleaseDoc/classReference/html/classpool_1_1ITransactionObserver.html

- *IDataTransform* : http://pool.cern.ch/currentReleaseDoc/classReference/html/classpool_1_1IDataTransform.html

- *ITokenValidator* : http://pool.cern.ch/currentReleaseDoc/classReference/html/classpool_1_1ITokenValidator.html

19.2. 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**.

`pool_extractFileIdentifier [-t technologyType] file1 file2 ...`
Opens the database files specified (PFN values), retrieves their File ID, and prints the pair FID (PFN) to the standard output. If the technology type is not specified, it is assumed that the database files have the ROOT format.

20. StorageSvc: User level semantics

20.1. Public classes UML diagram

20.2. How to use the component

The storage service (class: StorageSvc, interface IStorageSvc) allows object data to be saved to a persistent medium, so that these objects can be retrieved later by other processes and clients. If the shape of the transient object is the same as at the time of writing, the resulting object state must be identical.

The storage service saves and extracts object data to the persistent data store using a callback object provided by the conversion service. Such a callback, which must be supplied to the storage service, is called for both conversion directions (transient to persistent and persistent to transient) in the following sequence:

- Before the actual data conversion starts. At this initialization phase e.g. the callback can create the transient representation of the object.
- The callback object is notified by the storage service for converting every primitive data item of the object's persistent shape. Primitive is any data item, which is atomic to the database used. Hence, the type of these items depends on the database the data are supposed to be stored in. Examples are:

  - Numbers (bit, char, integer, float?)
  - String (varchar, longvarchar)
  - BLOB (**B**inary **L**arge **O**Bject)
  - o Some technologies, such as ROOT or Objectivity/DB, allow the handling of entire objects as single entities of the database. In this event the entire object may be treated as a single column.

  To receive the data, the address to a data buffer capable to hold the requested item is returned to the storage service, which subsequently fills the requested data. When reading, unwanted data items may be skipped.
- After all persistent data are converted a final callback is done. During this finalization phase eventually allocated temporary storage may be released or final calculations of redundant data items, which for optimization reasons are not present in the persistent shape may be recalculated.

Using this mechanism, the callback objects are able to build the transient representation of an object as the user requested it and fill it with the data stored in the persistent representation. Callback objects must be capable to translate a persistent shape of an object to its transient shape. When reading, the transient shape of the object may be altered and only a subset of the data in the persistent world can be used to populate the transient shape.

Callback objects must be declared outside the storage service and are passed to the service to perform a read or write action.

The creation of a transient object proceeds as follows:

- Given an identifier, the service locates the object in the persistent world. This object identifier , called token, only needs to be interpreted by the storage service, it is not needed anywhere outside.
- The storage service invokes the corresponding callback object supplied and populates the transient shape of the object with the persistent data items.

To populate the persistent data store is slightly different:

- The persistency service is given a transient object.
- The object is first allocated on the persistent medium. From this point on the object identifier (token) is fixed and can be used to address the object.
- It invokes the callback object to perform the data conversion. As a result of the conversion of a single data item a reference to the data to be stored is returned.

- It stores the persistent data.

20.3. Example of usage

Explicit use of this component is discouraged. Any usage by clients is implemented using a conversion service object, which collaborates with a data cache service to handling the lifetime of converted objects.

20.4. Solutions to common problems

The storage service component handles the technology specific aspects of writing data to a persistent medium or of reading data from a persistent medium. It serves the *conversion services* do actually do the full translation into transient objects. The conversion service instance(s) hereby only know about the interface of the storage manager.

20.5. Related components/classes

- *Conversion service* (class *ConversionSvc*): The conversion service determines which callback structure should be used in order to create the requested transient shape from the persistent shape of the data stored. When writing this callback must also be capable to populate the persistent shape from the data present in transient object.
- Data callback objects (class DataCallBack): Data callbacks are used for both directions of data flow:
  - When populating the transient object
  - When populating the persistent object

  The callbacks must conform to a common interface, which is specified in the class *DataCallBack*.

20.6. Public interfaces

The storage service exports it's functionality through the IStorageSvc interface. This interface allows clients to:

- Connect to an existing database or alternatively create a new database.
- Retrieve objects according to a given transient shape, provided a translation callback object between the transient and the persistent shape is supplied.
- Save objects in their persistent shape, provided a translation callback object between the transient and the persistent shape is supplied. The save process has three steps:
  1. A transaction must be started. This call to the interface returns an opaque TransAction handle, which must be used to later end the transaction.
  2. Marked any number of objects for persistency.
  3. The transaction must be closed.

21. StorageSvc: Reference Manual

21.1. Signatures of public interfaces

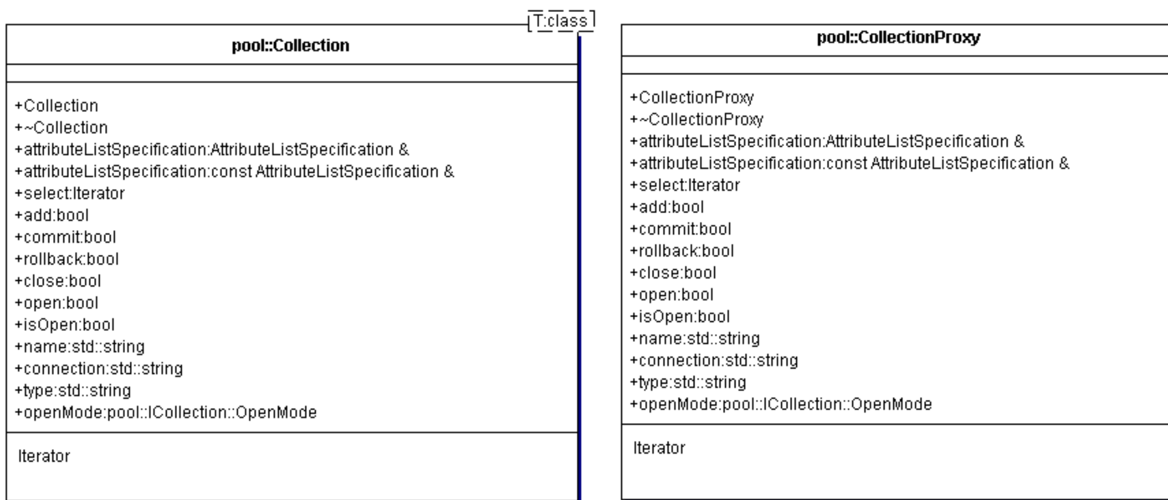
[Method/code fragment]

21.2. Command line tools

[Method/code fragment]

22. Collection: User level semantics

22.1. Public classes UML diagram



Class diagram showing public methods of Collection package

22.2. 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 23.

22.2.1. Collection creation

To a create a type-safe collection object called myCollection of known persistent data type, MyDataType, one can use the templated Collection class, Collection<T>, as follows:

```
pool::Collection<MyDataType> myCollection(
    myDataSvc,
    myCollectionType,
    myConnection,
    myCollectionName,
    pool::ICollection::CREATE);
```

where myDataSvc is a pointer to the POOL data service interface (see section Section 22.3.1 for an example), myCollectionType is an std::string specifying the type of POOL collection being created (e.g. "MySQLCollection" or "RootCollection"), myConnection is an std::string specifying the connection to the database where the collection is stored (set to "" for ROOT collections), and myCollectionName is an std::string specifying the persistent name of the collection (may be a full path name for ROOT collections). The last argument specifies that the collection is to be created if and only if a col-

lection 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.

Alternatively, if one does not wish to specify the type of persistent data being stored one can create a pointer to a Collection interface instead:

```
pool::ICollection* myCollection = pool::ICollectionFactory::get()->create(
                                         myDataSvc,
                                                                   myCollectionType,
                                       myConnection,
                                                                   myCollectionName,
                                                       pool::ICollection::CREATE);
```

where the input arguments have the same meaning as those mentioned above for the case of Collection<T> object creation.

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. For instance, with templated classes this can be done as follows:

```
pool::MultiCollection<MyDataType>          myMultiCollection(
                                                       myDataSvc,
                                               myMultiCollectionType,
                  myMultiCollectionConnection,
                                               myMultiCollectionName,
                                          pool::ICollection::CREATE);
```

where myDataSvc is a pointer to an instance of the POOL data service, myMultiCollectionType is an std::string specifying the type of multi-collection being created (e.g. "MSQLMultiCollection" or "RootMultiCollection"), myMulitCollectionConnection is an std::string specifying the connection to the database that contains the multi-collection (use "" for a ROOT collection) and myMultiCollectionName is an std::string specifying the persistent name of the multi-collection.

22.2.2. 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.attributeListSpeci
```

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[myAttributeName].setValue(myAttributeValue);
```

where myAttributeName is an std::string specifying the name of the given attribute in the list specification and myAttributeValue is the value of this attribute for the given event.

22.2.3. Adding data objects and associated metadata to a collection

To add data to a collection the user must first create a POOL reference (pool::Ref) to the given data object as follows:

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

where myDataSvc is a pointer to an instance of the POOL data service and MyDataType is the type of data object being stored. Then this object must be marked for writing by the persistency service:

```
myDataObjRef.markWrite(myPlacement);
```

where myPlacement 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). See section Section 22.3.1 for an example of a POOL placement object.

Then the data object reference and its associated attribute list are added to the collection. When using a collection object of type Collection<T> this is done as follows:

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

Alternatively, when using a pointer to the ICollection interface the Ref object needs to be converted to an object of type pool::Token first:

```
myCollection->add(myDataObjectRef.token(), myAttribList);
```

After a reference and its associated metadata have been added to the collection these changes must be committed for them to take effect:

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

After all references and their associated metadata have been added to the collection and committed for the given session the collection must be closed to properly complete the process:

```
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.

22.2.4. Performing queries on a collection

To perform a query on an existing explicit collection whose persistent data type is known, the collection can be opened as follows:

```
pool::Collection<MyDataType> myCollection(
    myDataSvc,
    myCollectionType,
    myConnection,
    myCollectionName,
    pool::ICollection::READ);
```

where the arguments have the same meanings as in section Section 22.2.1. On the other hand, if the collection is implicit (i.e. consists of the actual persistent data file itself) then the collection should be opened using the following arguments instead:

```
pool::Collection<MyDataType>                myCollection(
                                                     myDataSvc,
                                                ImplicitCollection,
                                                     myFileSpec,
                                                  myContainerName,
                                            pool::ICollection::READ);
```

where myFileSpec takes 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 <file name> is the name of the file containing the persistent data. The argument myContainer is the name of 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<MyDataType>::Iterator myCollIter =  myCollection.select(
                                                        myPrimaryQuery,
                                                       mySecondaryQuery,
                                                      myIteratorOptions)
```

where myPrimaryQuery is an std::string specifying any predicates to be 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 an std::string that 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 std::string 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, one could retrieve the GUID of the file containing the given persistified data object as follows:

```
std::string     myFileId     =     myCollIter.ref().token()->dbID();
```

where a pool::Token object is retrieved from the Ref object as an intermediate step.

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<MyDataType>::Iterator myCollIter = myCollection.select(
                                                                     "",
```

```
                                                    "",
                                                    myIteratorOptions);
```

Alternatively, one may avoid specification of the persistent data type by opening the collection via an ICollection interface and performing the query via an ICollectionIterator interface:

```
pool::ICollection* myCollection = pool::ICollectionFactory::get()->create(
                                                          myDataSvc,
                                                          myCollectionType,
                                    myConnection,
                                                          myCollectionName,
                                                   pool::ICollection::READ);

   pool::ICollectionIterator*    myCollIter    =    myCollection->select(
                                                          myPrimaryQuery,
                                                          mySecondaryQuery,
                                                          myIteratorOptions);
```

In this case, the Token object can be accessed directly without getting a Ref object first. For instance, to get the GUID of the file containing the persistent data one would now do the following:

```
std::string      myFileId      =      myCollIter->token()->dbID();
```

22.2.5. Exceptions Generated

Like the other POOL components, the collection 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 pool::Exception. Each thrown exception will contain a brief description of the exception, the method which threw it, and the corresponding module name. Additionally, some collection code may still throw exceptions of type std::exception.

22.3. 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 ICollection 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 ICollection using an open mode that reads an existing collection. Then declare an object of type CollectionIterator<T>::Iterator or ICollectionIterator 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 ICollectionIterator 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://pool.cvs.cern.ch/cgi-bin/pool.cgi/pool/Tests/]

22.3.1. Writing an Explicit Collection

First, a pointer to the POOL file catalog interface (pool:IFileCatalog) myFileCatalog is created and used to connect to and start a file catalog transaction.

```
pool::IFileCatalog* myFileCatalog = new pool::IFileCatalog;
pool::URIParser p("xmlfile_file:myFileCatalogName");
p.parse();
myFileCatalog->setWriteCatalog(p.contactstring());
myFileCatalog->connect();
myFileCatalog->start();
```

where in this case the file catalog is to be written to a local XML file called myFileCatalogName.

Next, a pointer to the POOL data service interface (POOL::IDataSvc) called myDataSvc is created and used to start a POOL data service transaction for use in data persistency.

```
pool::IDataSvc* myDataSvc = pool::DataSvcFactory::instance(myFileCatalog);
pool::DatabaseConnectionPolicy myPolicy;
myPolicy.setWriteModeForNonExisting(pool::DatabaseConnectionPolicy::CREATE);
myPolicy.setWriteModeForExisting(pool::DatabaseConnectionPolicy::OVERWRITE);
myDataSvc->session().setDefaultConnectionPolicy(myPolicy);
mydDataSvc->transaction().start(pool::ITransaction::UPDATE);
```

where myPolicy is an object of type pool::DatabaseConnectionPolicy used to specify protections on persistent data files. In this case, an existing file of the same name as the one being created will be overwritten.

Then 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> myCollection(
myDataSvc,
myCollectionType,
myConnection,
myCollectionName,
pool::ICollection::CREATE);
```

where myCollectionType is an std::string specifying the type of collection being created (e.g. "MySQLCollection" or "RootCollection"), myConnection is an std::string specifying the connection to the database containing the collection and myCollectionName is an std::string specifying the persistent name of the collection (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 can be specified in an Athena job options file.

Then the metadata schema of the collection is defined as follows:

```
pool::AttributeListSpecification& myAttribSpec =
myCollection.attributeListSpecification();
myAttribSpec.push_back("myRun", "unsigned int");
myAttribSpec.push_back("myEvent", "unsigned int");
myAttribSpec.push_back("myInt", "int");
myAttribSpec.push_back("myUInt", "unsigned int");
myAttribSpec.push_back("myShortInt", "short");
myAttribSpec.push_back("myUShortInt", "unsigned short");
myAttribSpec.push_back("myLongInt", "long");
myAttribSpec.push_back("myULongInt", "unsigned long");
myAttribSpec.push_back("myFloat", "float");
myAttribSpec.push_back("myDouble", "double");
myAttribSpec.push_back("myBool", "bool");
```

```
myAttribSpec.push_back("myString",            "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://pool.cvs.cern.ch/cgi-bin/pool.cgi/pool/Tests/Collection_Write/]

Next an object called myPlacement of type pool::Placement is created which specifies all of the information necessary for the POOL persistency service to persistify the data whose references and associated metadata are to be added to the collection:

```
pool::Placement                      myPlacement;
myPlacement.setDatabase(myDataFileName, pool::DatabaseSpecification::PFN);
myPlacement.setContainerName(myContainerName);
myPlacement.setTechnology(pool::ROOTTREE_StorageType.type());
```

In this case, the persistent data is stored in a container called myContainerName which resides in a file which has a physical file name (PFN) of myDataFileName and which uses ROOT trees as its persistent storage technology.

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>
mySimpleTestClass(myDataSvc, new SimpleTestClass());

mySimpleTestClass.markWrite(myPlacement);

mySimpleTestClass->setNonZero();

pool::AttributeList       myAttributeList(myAttribSpec);
myAttributeList["myRun"].setValue(run);
myAttributeList["myEvent"].setValue(evt);
myAttributeList["myUInt"].setValue(8);
myAttributeList["myShortInt"].setValue(-4);
myAttributeList["myUShortInt"].setValue(6);
myAttributeList["myLongInt"].setValue(-9);
myAttributeList["myULongInt"].setValue(5);
myAttributeList["myFloat"].setValue(3.5);
myAttributeList["myDouble"].setValue(9.7);
myAttributeList["myBool"].setValue(true);
myAttributeList["myString"].setValue("test      string");

myCollection.add(mySimpleTestClass,     myAttributeList);
}
}
}
```

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:

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

To complete the persistency process the POOL data service and file catalog transactions must also be committed:

```
myDataSvc->transaction().commit();
myFileCatalog->commit();
```

22.3.2. Reading an Explicit Collection

First, the POOL file catalog and data service are accessed for read:

```
pool::IFileCatalog* myFileCatalog = new pool::IFileCatalog;
pool::URIParser p("xmlfile_file:myFileCatalogName");
p.parse();
myFileCatalog->addReadCatalog(p.contactstring());
myFileCatalog->connect();
myFileCatalog->start();

pool::IDataSvc* myDataSvc = pool::DataSvcFactory::instance(myFileCatalog);
pool::DatabaseConnectionPolicy myPolicy;
myPolicy.setReadMode(pool::DatabaseConnectionPolicy::READ);
myDataSvc->session().setDefaultConnectionPolicy(myPolicy);
myDataSvc->transaction().start(pool::ITransaction::READ);
```

where, in this case, an existing local XML file catalog called myFileCatalog is opened for reading.

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

```
pool::Collection<SimpleTestClass> myCollection(
myDataSvc,
myCollectionType,
myConnection,
myCollectionName,
pool::ICollection::READ);
```

where myDataSvc, myCollectionType, myConnection and myCollectionName have the same meanings as in section Section 22.3.1. Then a server side query is defined and used to obtain an iterator to the collection as follows:

```
pool::Collection<SimpleTestClass>::Iterator myCollIter = myCollection.select(
myPrimaryQuery,
mySecondaryQuery,
myIteratorOptions)
```

where myPrimaryQuery consists of the desired predicates (e.g. 'event_number > 2 AND event_number < 5' for a MySQL query), mySecondaryQuery can contain a secondary set of predicates (which, as mentioned in section Section 22.2.4, is presently only implemented for multi-collections and has a default value of ""), and myIteratorOptions 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 myPrimaryQuery 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 (myCollIter.next())
{
myCollIter->streamOut(std::cout);

pool::AttributeList myAttributeList =
myCollIter.attributeList();

myAttributeList.print(std::cout);
```

```
std::string myFileId = myCollIter.ref().token()->dbID();
std::string myBestPFN;
std::string myFileType;
myFilecatalog->lookupBestPFN(
    myFileId,
    pool::IFileCatalog::READ,
    pool::IFileCatalog::SEQUENTIAL,
    myBestPFN,
    myFileType);
std::cout << "Physical file name is " << myBestPFN << std::endl;
std::cout << "File type is " << myFileType << 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 physical file name of the persistent data file.

After all desired queries have been performed the collection is closed and the data service and file catalog transactions are committed:

```
myCollection->close();
myDataSvc->transaction().commit();
myFileCatalog->commit();
```

22.3.3. Updating an Explicit Collection

First a POOL file catalog transaction is started for both read and write access to a local XML POOL file catalog called myFileCatalogName and a POOL data service transaction is started for update of existing persistent data files:

```
pool::IFileCatalog* myFileCatalog = new pool::IFileCatalog;
pool::URIParser p("xmlfile_file:myFileCatalogName");
p.parse();
myFileCatalog->addReadCatalog(p.contactstring());
myFileCatalog->addWriteCatalog(p.contactstring());
myFileCatalog->connect();
myFileCatalog->start();

pool::IDataSvc* myDataSvc = pool::DataSvcFactory::instance(myFileCatalog);
pool::DatabaseConnectionPolicy myPolicy;
policy.setWriteModeForExisting(pool::DatabaseConnectionPolicy::UPDATE);
policy.setReadMode(pool::DatabaseConnectionPolicy::UPDATE);
myDataSvc->session().setDefaultConnectionPolicy(myPolicy);
myDataSvc->transaction().start(pool::ITransaction::UPDATE);
```

Then 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> myCollection(
    myDataSvc,
    myCollectionType,
    myConnection,
    myCollectionName,
    pool::ICollection::UPDATE);
```

where the meanings of myDataSvc, myCollectionType, myConnection and myCollectionName are the same as described in the write and read examples above.

Then the metadata list specification is obtained from the collection:

```
pool::AttributeListSpecification& myAttribSpec =
    myCollection.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>
            mySimpleTestClass(myDataSvc, new SimpleTestClass());

        mySimpleTestClass.markWrite(myPlacement);

        mySimpleTestClass->setNonZero();

        pool::AttributeList myAttributeList(myAttribSpec);
        myAttributeList["myRun"].setValue(run);
        myAttributeList["myEvent"].setValue(evt);
        myAttributeList["myUInt"].setValue(3);
        myAttributeList["myShortInt"].setValue(-6);
        myAttributeList["myUShortInt"].setValue(7);
        myAttributeList["myLongInt"].setValue(-2);
        myAttributeList["myULongInt"].setValue(10);
        myAttributeList["myFloat"].setValue(4.6);
        myAttributeList["myDouble"].setValue(8.2);
        myAttributeList["myBool"].setValue(false);
        myAttributeList["myString"].setValue("another test string");

        myCollection.add(mySimpleTestClass, myAttributeList);
    }
}
```

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

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

Finally, the data service and file catalog transactions must be committed to complete the process:

```
myDataSvc->transaction().commit();
myFileCatalog->commit();
```

22.3.4. 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.

23. Collection: Reference Manual

23.1. Signatures of public interfaces

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

23.1.1. 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 technology specific co object (e.g. MySQLCollection or RootCollection) **Syntax:** `Collection(IDataSvc* dataSvc, std::string collectionType, std::string collecti std::string collection ICollection::OpenMode openM`
~Collection()	Default destructor. **Syntax:** `Not impleme`
attributeListSpecification()	Returns the metadata schema of the collection. **Syntax:** `const AttributeListSpecifica attributeListSpecification() c`
select()	Returns an iterator over a subset of the collection data that satisfies a s predicates on the data's associated metadata. Iterator options deter whether caching is used to store the collection data before the c (no caching is the default). **Syntax:** `Iterator select(std::string primaryQuery, std::string secondaryQ std::string IteratorOpti`
add()	Adds a data object and its associated metadata to the collection.

	Syntax: `bool add(const Ref<T>& const AttributeList&`

commit()	Persistifies the last changes made to the collection. **Syntax:** `bool`
rollback()	Aborts the last changes made to the collection before committing. **Syntax:** `bool`
close()	Closes the collection explicitly while aborting uncommitted changes. **Syntax:** `bool`
open()	Reopens the collection explicitly after it was closed. **Syntax:** `bool`
isOpen()	Checks if the collection is open. **Syntax:** `bool isOpen()`
name()	Returns the persistent name of the collection. **Syntax:** `std::string name()`

connection()	Returns the connection string to the database containing the collection. **Syntax:** `std::string connection()`

type()	Returns the technology specific type of collection being u (MySQLCollection, RootCollection, etc.). **Syntax:** `std::string type() c`
openMode()	Returns the mode used to open the collection (CREATE, READ, etc.) **Syntax:** `pool::ICollection::OpenMode openMode() c`

23.1.2. 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 values for the current data object. **Syntax:** `const AttributeList& attributeList()`

ref()	Returns a reference of type pool::Ref for the current data object.
	Syntax:
	`Ref<T>&`
	`const Ref<T>& ref()`

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

23.1.3. ICollection

A class used to store, retrieve and update collections of data objects and their associated metadata. If one does not want to specify the data object type upon collection creation this interface should be used instead of the Collection<T> class. The ICollection interface is created via the ICollectionFactory interface. See section Section 22.2 for details. The public methods of the ICollection interface are defined as follows:

ICollection()	Protected constructor. **Syntax:** `explicit ICollection(ISession*`
~ICollection()	Default destructor. **Syntax:** `~ICo`

attributeListSpecification()	Returns the metadata schema of the collection.
	Syntax:
	`AttributeListSpecification& attributeListSpeci`
	`const AttributeListSpe`
	`attributeListSpecification()`

select()	Returns an iterator over a subset of the collection data that

	of predicates on the data's associated metadata. Iterator opt determine whether caching is used to store the collection data b the query (no caching is the default).
	
	Syntax:
	
	`ICollectionIterator* select(std::string primaryQuer`
	`std::string secondary(`
	`std::string IteratorOpti`

add()	Adds a data object and its associated metadata to the collection.
	
	Syntax:
	
	`bool add(const Token& t`
	`const AttributeList& attrI`

commit()	Commits the last changes made to the collection. **Syntax:** `bool comm`
rollback()	Aborts the last changes made to collection before committing. **Syntax:** `bool rollba`
close()	Closes the collection explicitly while aborting uncommitted changes. **Syntax:** `bool clo`
open()	Reopens the collection explicitly after it was closed. **Syntax:** `bool op`
isOpen()	Checks if the collection is open. **Syntax:** `bool isOpen() c`
name()	Returns the persistent name of the collection. **Syntax:**

	`const std::string& name()`

connection()	Returns the connection string to the database containing the collection. **Syntax:** `const std::string& connection()`
type()	Returns the technology specific type of collection be (MySQLCollection, RootCollection, etc.). **Syntax:** `const std::string& type()`
openMode()	Returns the mode used to open the collection (CREATE, READ, etc.) **Syntax:** `const pool::ICollection::OpenMode openMode`
session()	Returns a pointer to the present POOL data service session. **Syntax:** `const ISession* session()`

23.1.4. ICollectionIterator

An iterator class to perform queries on objects inheriting from ICollection. If one does not want to specify the data object type upon collection creation this interface should be used along with the ICollection interface instead of the Collection<T> class and its associated templated iterator class.

Depending on how this 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:

~ICollectionIterator()	Default destructor. **Syntax:** `Not`
next()	Retrieves the next data object in the collection. **Syntax:** `bool`

attributeList()	Returns the list of associated metadata values for the current data object. **Syntax:** `const AttributeList& attributeList() c`
token()	Returns a reference of type pool::Token for the current data object. **Syntax:** `Token* token() c`

23.2. Command line tools

The package *pool/Utilities/Collection* [http://pool.cvs.cern.ch/cgi-bin/pool.cgi/pool/Utilities/Collection/] contains several C++ command line tools for accessing and managing POOL collections. For example, it includes tools to create a collection from one or more existing collections, append one or more existing collections to another existing collection or print out a list of the physical names of all files used by a set of collections.

All utilities allow the use of a query on the input collections. The utilities can be applied to any technology-specific collection recognized by POOL. Presently these types include ROOT, MySQL and Oracle (via the RelationalCollection package). For instance, a combination of ROOT and MySQL collections can be appended to an existing ROOT or MySQL collection. All utilities allow the specification of the full path name of the POOL file and collection catalogs being used if they differ from the default values. In addition, the copy and append utilities allow suppression of collection metadata copying and appending, respectively.

To facilitate back navigation (e.g. from an event tag collection to its parent ESD file), the copy utility allows for the creation of a new collection from a column of Token type metadata of an existing collection.

A description of each of the utilities and their usage is presented in the following table. The command line arguments shown in the usage specifications have the following meanings:

-src [+]<<input collection name> <input collection type>>> The input collection specification(s). Such a specification consists of a space separated collection name and associated collection type pair. If multiple input collections are used then a space separated list of such name-type pairs must be provided. The collection types presently recognized by these utilities include "RootCollection", "MySQLCollection", "MySQLltCollection" (uses an extra link table for more efficient storage), and "RelationalCollection" (these are MySQL or Oracle collections produced by the new POOL RelationalCollection package. The technology is determined by the syntax of the database connection string command line options. See below). For use with POOL collection catalogs, the collection types "PHYSICAL_NAME", "LOGICAL_NAME" and "GUID" are also recognized by this utility. NOTE: For a collection of type "RootCollection" the collection name must end with the suffix ".root" and this suffix MUST NOT be included in the collection name specification on the command line.

-dst [+]<<output collection name> <ouput collection type>> The output collection specification(s). Like the input collection specification, this specification consists of a space separated collection name and associated collection type pair. If multiple output collections are used then a space separated list of such name-type pairs must be provided. This qualifier recognizes the same collection types as the "-src" qualifier and the same rules apply to collection name specification for collections of type "RootCollection" (see above).

[-srcconnect [input database connection string]] A semi-optional command line argument which MUST be specified if any collections of type MySQL or Oracle are specified as input (via the -src qualifier). The default value for this connection string is "". The syntax for the connection string is:

[protocol]://[username]:[password]@[host]:[port]/[path]
where [protocol] can be "mysql" or "oracle" in this case. NOTE: If input collections of type "RelationalCollection" are used then the environment variables POOL_AUTH_USER and POOL_AUTH_PASSWORD must be set.

[-dstconnect [output database connection string]] A semi-optional command line argument which MUST be specified if any collections of type MySQL or Oracle are specified as ouput (via the -dst qualifier) and no input database connection string has been specified. Otherwise, the output database connection string for MySQL and Oracle collections is set to the input database connection string, which itself has to have been set or an error message results. The syntax of this connection string is the same as that for "-srcconnect". NOTE: If the output collections are of type "RelationalCollection" then the environment variables POOL_AUTH_USER and POOL_AUTH_PASSWORD must be set.

[-query [predicate string]] An optional set of query predicates to be applied to all input collections. DEFAULT = ""

[-queryopt [+][option]] A space separated list of options specifying how the metadata is to be read in. "SELECT *" implies that all metadata attributes are to be read in. Individual attributes may be read in by following "SELECT" with a comma separated list of the desired attributes. "SELECT" with nothing following it specifies that no metadata is to be read in. DEFAULT = "SELECT"

[-fcread [+][read file catalog connection]] An optional space separated list of the POOL read file catalog connection strings to use. DEFAULT = "xmlcatalog_file:PoolFileCatalog.xml"

[-fcwrite [write file catalog connection string]] An optional specification of the write file catalog connection string. DEFAULT = "xmlcatalog_file:PoolFileCatalog.xml"

[-ccread [+][read collection catalog connection]] An optional space separated list of the POOL read collection catalog connection strings to use. DEFAULT = "xmlcatalog_file:CollectionCatalog.xml"

[-ccwrite [write collection catalog connection string]] An optional specification of the write collection catalog connection string. DEFAULT = "xmlcatalog_file:CollectionCatalog.xml"

[-noattrib] An optional qualifier with no arguments which specifies that metadata is not to be copied or appended to the output collection.

[-tokattrib [token attribute name]] An optional argument which specifies that the Token objects (i.e. persistent data references) of the ouput collection are to consist of the Token object metadata of name "token attribute name" of the input collections.

[-nevtdmp] An optional specification of the number of events starting from the first for which to print out the Token or event level metadata information. If the string "all" is specified for this option then the Token or metadata values are printed out for each event. DEFAULT = 0

[-nevtstat] An optional specification of the number of events starting from the first for which to calculate maximum, minimum and average values of the metadata (for the data types for which these calculations are applicable). If the string "all" is specified for this option then these calculations are performed for every event. DEFAULT = 0

[-nevtread] An optional specification of the number of events starting from the first for which to read in the metadata. If the string "all" is specified for this option then the metadata is read in for every event. If this option accidentally specifies that less events are to have their metadata read in than are to have their metadata printed out or analyzed (via the qualifiers "-nevtread" and "-nevtstat", respectively) then this option is set to the larger of these two numbers. DEFAULT = 0

[-nevtscan] An optional specification of the number of events starting from the first for which to scan and perform a simple count. If the string "all" is specified for this option then every event is scanned and counted. If this option accidentally specifies that less events are to be scanned than are to have their metadata read in, printed out or analyzed (via the qualifiers "-nevtread", "-nevtdmp" and "-nevtstat", respectively) then this option is set to the largest of these three numbers. DEFAULT = "all"

[-nevtpercommit] An optional specification of the maximum number of events to process before committing the latest changes to the given collection. The default is to commit after every 10000

events. Note: Committing too often can lead to lower performance while committing too seldom can lead to excessive memory consumption between commits.

[-nevtperprint] An optional specification of the number of events to process before printing a message to screen stating how many events have been processed. The default is to not print anything to the screen.

[-timing] An optional qualifier with no arguments which specifies that timing information of key processes is to printed out for each input collection at the end of event processing. Useful for performance testing.

[-quiet] An optional qualifier with no arguments which specifies that no information except the (optional) event dump and event scan status information is to be printed to the screen. Useful for performance testing.

Note: Since only one input and one output database connection string can be specified one is restricted to only merge MySQL or Oracle type collections from a single database and only output the results to a single database unless a collection catalog is being used (via the collection types PHYSICAL_NAME, LOGICAL_NAME or GUID). Examples of collection catalog usage with the collection utilities can be found in: *pool/Utilities/Collection/tests* [http://pool.cvs.cern.ch/cgi-bin/pool.cgi/pool/Utilities/Collection/tests]

CollCopy	Creates one or more identical new collections by copying a single exi collection or by merging two or more existing collections in the orde which they are listed on the command
	
	Usage:
	
	`CollCopy`
	`-src [+]<<input collection name> <input collection type`
	`-dst [+]<<output collection name> <ouput collection type>`
	`[-noattrib]`
	`[-tokattrib [token attribute name]]`
	`[-srcconnect [input database connection string]`
	`[-dstconnect [output database connection string]`
	`[-query [predicate string]]`
	`[-queryopt [+][metadata read opt`
	`[-ccread [+][read collection catalog connection string`
	`[-ccwrite [write collection catalog connection string`
	`[-nevtpercommit [max. Number of events to process betwe`
	`collection commits]]`
	`[-nevtperprint [number of events to process between prin`
	`to the scr`

CollAppend	Appends one or more existing collections to one or more existing collec in the order of which they are
	
	Usage:
	
	`CollAppend`
	`-src [+]<<input collection name> <input collection type`
	`-dst [+]<<output collection name> <ouput collection type>`
	`[-noattrib]`
	`[-srcconnect [input database connection string]`
	`[-dstconnect [output database connection string]`
	`[-query [predicate string]]`
	`[-queryopt [+][metadata read opt`
	`[-ccread [+][read collection catalog connection string`
	`[-ccwrite [write collection catalog connection string`
	`[-nevtpercommit [max. Number of events to process betwe`
	`collection commits]]`
	`[-nevtperprint [number of events to process between prin`
	`to the scr`

CollListPFN	Prints out the list of physical names of all files used by existing **Usage:** CollListPFN -src [+]<<input collection name> <input collectio [-srcconnect [input database connection st [-query [predicate string]] [-queryopt [+][metadata read [-fcread [+][read file catalog connection s [-ccread [+][read collection catalog connectio
CollListFileGUID	Prints out a list of the globally unique identifiers of all by a list of existing **Usage:** CollListFileGUID -src [+]<<input collection name> <input collectio [-srcconnect [input database connection st [-query [predicate string]] [-queryopt [+][metadata read [-fcread [+][read file catalog connection s [-ccread [+][read collection catalog connectio
CollListToken	Prints out the stringified POOL Token objects (i.e. stringified references) for the number of events **Usage:** CollListToken -src [+]<<input collection name> <input collectio [-srcconnect [input database connection st [-query [predicate string]] [-queryopt [+][metadata read [-ccread [+][read collection catalog connection [-nevtdmp [number of events for which to print

CollListAttrib	Prints out the metadata values for all events in a list o collections. If one chooses to not print out any events (via th option) then only the metadata specification is p **Usage:** CollListAttrib -src [+]<<input collection name> <input collectio [-srcconnect [input database connection st [-query [predicate string]] [-queryopt [+][metadata read [-ccread [+][read collection catalog connection [-nevtdmp [number of events for which to print out m [-nevtstat [number of events for which to analyze m [-nevtread [number of events for which to read in m [-nevtscan [number of events to scan and [-nevtperprint [Number of events to scan between to the screen]] [-timing]

Examples of usage of the tools described in this section can be found in the regression tests area of the POOL Collection utilities package: *pool/Utilities/Collection/tests* [http://pool.cvs.cern.ch/cgi-bin/pool.cgi/pool/Utilities/Collection/tests]

In addition to the C++ command line utilities described above, there is an external package by Julius Hrivnac called ColMan (Collection Management for AIDA ITuples) which consists of utilities that use a FreeHEP SQLTuple interface to access and manage POOL collections. SQLTuple is an extension of the AIDA ITuple interface which enables ITuples to be stored in an SQL type database via a JDBC API. ColMan is designed to be compatible with the POOL AttributeList package. In addition to the Java command line interfaces, the ColMan utilities can also access a collection via a web service.

ColMan utilities can be used to make a copy of an existing collection (or a subset of the collection via a query), merge two collections, get a list of the OID's of a collection and other useful operations.

A detailed description of these tools and their usage can be found at:

http://hrivnac.home.cern.ch/hrivnac/Activities/Packages/ColMan/
[http://hrivnac.home.cern.ch/hrivnac/Activities/Packages/ColMan/]

24. ObjectRelationalAccess: User level semantics

24.1. Object/Relational mapping element hierarchies

An *object/relational mapping* is a hierarchy of *object/relational mapping elements* accomodating one or more classes. The immediate elements of a mapping correspond to the C++ classes of the persistent-capable objects.

A mapping element can be of one of the following types:

Object

Primitive

Array

InlineCArray

CArray

Pointer

Reference

PoolReferencepool::Reference

A mapping element has the following attributes:

scope name

scope name

variable name

variable type

table name

column names

sub-elements

24.2. Versioning, storing and materializing mappings

A user may use the **ObjectRelationalMappingPersistency** class to store into a relational database the complete information for a given mapping. Mappings are stored with versions and a class may appear in several versions with different mappings.

The ObjectRelationalMappingPersistency class allows the user not only to perform mapping I/O, but also to **materialize** a mapping, i.e. prepare the relational schema such that all the necessary tables and constraints involved in the mapping are created.

24.3. Default and customizable mapping rules

Given a class (through the corresponding SEAL dictionary interface) the **ObjectRelationalDefault-MappingBuilder** class can be used to generate a mapping of a C++ class using default mapping rules. The latter include capitalizing the names of the types and the variable names, so that the corresponding table and column names are capitalized as well.

It is strongly advised to use this class in collaboration with the **ObjectRelationalTableNameRationalizer** class so that the generated table names are of a reasonable size, and they do not clash with existing names in the current schema.

A user can overwrite the default rules, partially or completely, through the API of the ObjectRelationalMappingElement interface. Alternatively, one can run the appropriate XML-driven command line tool (see Section 25.2 for more details), which can also be used for the accomodation of existing relational schema and data.

25. ObjectRelationalAccess: Reference Manual

25.1. Signatures of public interfaces

- *ObjectRelationalMapping.h* : http://pool.cern.ch/releases/POOL_2_0_0/doc/classReference/html/classpool_1_1ObjectRelationalMapping.html
- *ObjectRelationalMappingElement* : http://pool.cern.ch/releases/POOL_2_0_0/doc/classReference/html/classpool_1_1ObjectRelationalMappingElement.html
- *ObjectRelationalMappingPersistency* : http://pool.cern.ch/releases/POOL_2_0_0/doc/classReference/html/classpool_1_1ObjectRelationalMappingPersistency.html
- *ObjectRelationalDefaultMappingBuilder* : http://pool.cern.ch/releases/POOL_2_0_0/doc/classReference/html/classpool_1_1ObjectRelationalDefaultMappingBuilder.html
- *ObjectRelationalTableNameRationalizer* : http://pool.cern.ch/releases/POOL_2_0_0/doc/classReference/html/classpool_1_1ObjectRelationalTableNameRationalizer.html

25.2. Command line tools

A set of commands has been provided for the management of the mapping information stored in special tables into the database. The mapping information can be submitted and retrieved as a special driver file in XML format. Here follows a list of the basic entities of the XML semantics:

- *Mapping*- The main entity, encapsulating the entire information. Mandatory attribute is *version* . A XML mapping descriptor should contain a single *Mapping* entity, which contains a list of *Class* subelements.

- *Class*- The entity describing the class item to be mapped into RDBMS tables and columns.Mandatory attribute is *name*, optional attributes *table* and *id_columns* can be specified to force the data from this class to be stored in the specified table, using as primary key the list of columns specified (spacebar separated). When one or two optional parameter are not specified, the corresponding default mapping value is used.

  For each class attribute, an specific entity can be introduced to describe the mapping into the RD-BMS. Sub-elements allowed for *Class* are: *Primitive, Container, Object, Blob, CArray, Inline-CArray*

- *Primitive*- The entity describing a class attribute of primitive type. Mandatory attribute is *name* (variable name of the attribute in the class definition), optional attribute is *column*, which can be specified to force the data corresponding to this attribute to be stored in the specified column. If not specified, the default mapping value is used.

  Primitive does not allow sub elements.

- *Object*- The entity describing a complex Class-defined attribute. Mandatory attribute is *name* (variable name of the attribute in the class definition). Optional attributes are *table, id_columns*, which can be specified respectively to force a specific tablename for the object data and a specific list of columns for the primary key (spacebar separated list).

  In order to allow to specify the mapping for the attributes of the object, *Object* allows subelements: *Primitive, Container, Object, Blob, CArray, InlineCArray.*

- *Blob*- The entity describing a complex Class-defined attribute, to be streamed using an user-defined Streamer function, and stored in the database a BLOB type. Mandatory attribute is *name* (variable name of the attribute in the class definition). Optional attributes are *table, id_columns*, which can be specified respectively to force a specific tablename for the object data and a specific column for the blob data.

  No subelements are admitted for *Blob* elements.

- *Container*- The entity describing any attribute instance of an STL container of primitive or complex objects.Mandatory attribute is *name* (variable name of the attribute in the class definition). Optional attributes are *table, position_column, id_columns*, which can be specified respectively to force a specific tablename for the array data, a specific name for the column describing the position in the array, and a specific list of columns for the primary key (spacebar separated list). When one or two optional parameter are not specified, the corresponding default mapping value is used.

  In order to allow to specify the mapping for the objects composing the array, *Container* allows subelements: *Primitive, Container, Object, CArray, InlineCArray.* As a subelements of the Container entities, more elements are in principle admitted. However, the only case where more subelements are required is for the description of custom mapping of associative containers, where key_type and mapped_type have to be specified.

  Note that the following convention has to be used (mandatory) for the field *name* of the *Container* subelements:

  - Non associative containers (std::vector, std::set, std::list, std::queue, std::deque): *value_type*

  - Associative containers (std::map, std::multimap, hash_map): *key_type* for key type and *mapped_type* for the mapped type.

- *InlineCArray*- The entity describing a C-Array attribute, where individual elements (maximum is 100) are stored in the main parent table, assigning a column to each of them.Mandatory attribute is *name* (variable name of the attribute in the class definition, indicating the index of the element). Optional attributes are *table, id_columns*, which can be specified respectively to force a specific

tablename for the object data and a specific list of columns for the primary key (spacebar separated list).

In order to allow to specify the mapping for the attributes of the object, *InlineCArray* allows subelements: *Primitive, Container, CArray, InlineCArray, Object, Blob.*

- *CArray*- The entity describing a C-Array attribute, where individual elements (maximum is 100) are stored in a dedicated table, in the same schema used for the Container. Mandatory attribute is *name* (variable name of the attribute in the class definition). Optional attributes are *table, position_column, id_columns*, which can be specified respectively to force a specific tablename for the array data, a specific name for the column describing the position in the array, and a specific list of columns for the primary key (spacebar separated list). When one or two optional parameter are not specified, the corresponding default mapping value is used.

  In order to allow to specify the mapping for the objects composing the c-array, *CArray* allows subelements: *Primitive, Container, Object, CArray, InlineCArray.*

Commands available:

- **pool_build_object_relational_mapping**

  Builds a mapping for a set of classes, with directives specified in the XML driver file.

  Mandatory parameters to be specified are:

  - **-f (--file)** : the mapping xml file name
  - **-d (--dictionary)** : the dictionary libraries
  - **-c (--connectionString)** : the connection string

  Mandatory parameters for connections requiring authentication:

  - **-u (--user)** : the user name
  - **-p (--password)** : the user password

  Optional parameters:

  - **-o (--outputfile)** : the full mapping xml file name (output)
  - **-b (--buildonly)** : don't store the mapping built
  - **-debug (--debug)** : enable the verbose mode
  - **-h (--help)** : display the help

- **pool_list_object_relational_mapping**

  Dumps on the screen the complete list of the stored mapping versions

  Mandatory parameters to be specified as alternative:

  - **-v (--version)** : the version label of the mapping to find
  - **-l (--list)** : the versions list of the mappings

  Mandatory parameters always to be specified are:

- **-c (--connectionString)** : the connection string

  Mandatory parameters for connections requiring authentication:

  - **-u (--user)** : the user name
  - **-p (--password)** : the user password

  Optional parameters:

  - **-o (--outputfile)** : the full mapping xml file name (output)
  - **-debug (--debug)** : enable the verbose mode
  - **-h (--help)** : display the help

- **pool_retrieve_object_relational_mapping**

  Retrieves the full mapping information for a set of classes given a mapping version into the XML file.

  Mandatory parameters to be specified are:

  - **-v (--version)** : the version label of the mapping to retrieve
  - **-c (--connectionString)** : the connection string

  Mandatory parameters for connections requiring authentication:

  - **-u (--user)** : the user name
  - **-p (--password)** : the user password

  Optional parameters:

  - **-o (--outputfile)** : the full mapping xml file name (output)
  - **-debug (--debug)** : enable the verbose mode
  - **-h (--help)** : display the help

26. RelationalStorageService: User level semantics

26.1. Object Storage through the StorageSvc interfaces

A user never interacts directly with the RelationalStorageService component. One uses exclusively the StorageSvc interfaces (Refer to the relevant documentation). The loading of the RelationalStorageService plugin is loaded automatically during runtime whenever the corresponding StorageSvc technology is specified.

The domain technology name is "POOL_RDBMS" and the corresponding static pool::DbType object is **pool::POOL_RDBMS_StorageType**.

There are two minor technologies associated to this major one.

pool::POOL_RDBMS_HOMOGENEOUS_StorageType

pool::POOL_RDBMS_POLYMORPHIC_StorageType

The customizing of the object/relational mapping, i.e. defining how the object data will be organized in terms of tables rows and constraints, overidding the default automatic rules, can be achieved using the relevant command-line tools of the ObjectRelationalAccess package. (Refer to the relevant documentation).

The component requires the existence of an IRelationalService and an IAuthenticationService in the context hierarchy. If nothing is found the default RelationalService as well as the EnvironmentAuthenticationService components are loaded.

27. RelationalStorageService: Reference Manual

27.1. Parameters of the POOL_RDBMS database

- **FORMAT_VSN** : A version of the data format/layout to be used in the future for backward compatibility. Its current value is "1.0".

28. RelationalCollection: User level semantics

28.1. Semantical behaviour

A user never interacts directly with the RelationalCollection component. One uses exclusively the Collection interfaces (Refer to the relevant documentation). The loading of the RelationalCollection plugin is loaded automatically during runtime whenever the corresponding Collection technology is specified.

The RelationalCollection package has been implemented in such a way so that multiple collections can be residing within a given RDBMS schema.

Note that for the RelationalCollection to be useable, an AuthenticationService module should be installed in the application context, except for the case where a back-end is selected that does not require authentication (like an SQLite-based back-end). If no such service is found, the "CORAL/Services/EnvironmentAuthenticationService" component is loaded by default.

For the current implementation whenever a query is issued, row prefetching is enabled. The number of pre-fetched rows is set to 10000.

29. RelationalCollection: Reference Manual

29.1. RelationalCollection parameters

- **Number of pre-fetched rows** : This is currently set to 10000.