# Collections of Collections in POOL

- Overview of the Collection component
  - Collection Design / Functionality
  - Meta Data Interface
  - Performance
- Collections of Collections, the MultiCollection component
- Summary and Outlook

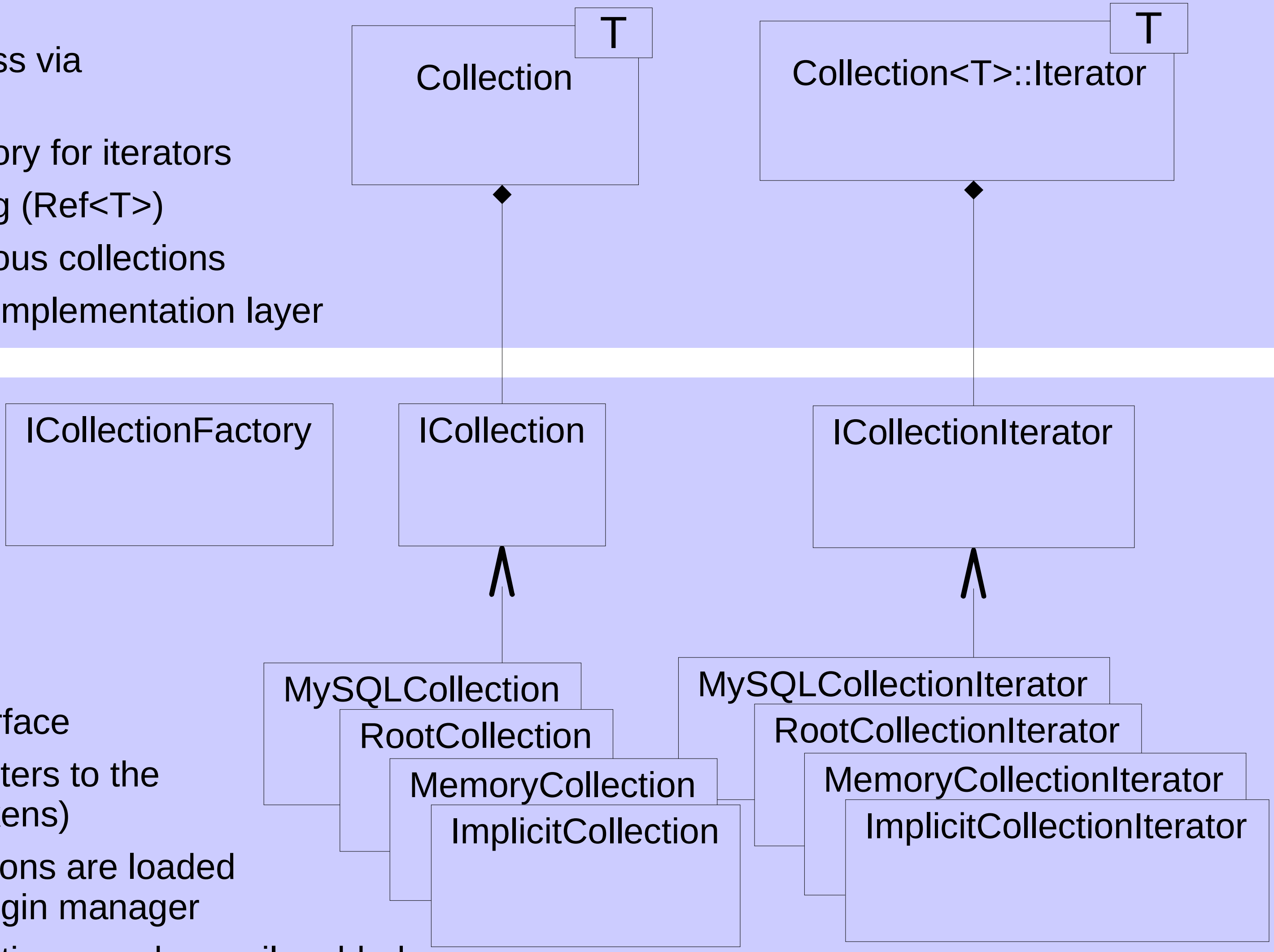
# Collection Overview

- Collections keep track of large sets of objects and their description (meta data)
- **Explicit Collections:**
  - Explicit list of object references
  - Allows logical grouping
  - Queryable if associated with meta data
  - Use case: Event collections, fast pre-selection based on event meta data (CMS: O($10^9$) events/year)
  - Current implementations:<br>MySQL, Root and STL(transient)
- **Implicit Collections**
  - Defined by the physical containment of the objects
  - Provide user friendly access to all objects in a given database container
- Both collection types use the same interface

# Collection Design

**User interface**

- Type safe object access via Ref<T> smart pointers
- Collection acts as factory for iterators
- Dynamic type checking (Ref<T>)
- Support for heterogenous collections
- Proxy for the untyped implementation layer

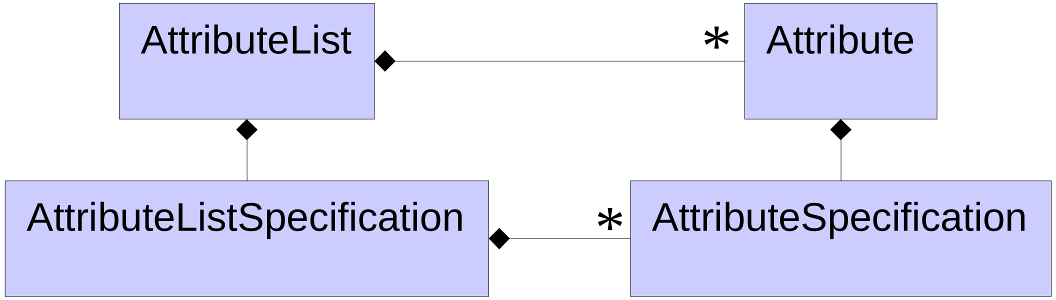


**Implementation:**

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

# Meta Data Interface

- AttributeList component:
  - Specifies the transient C++ part of the meta data interface
  - Used by FileCatalog and Collection component
  - Implemented by Kuba



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

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

# Meta Data How-to

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

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

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

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

# Collection How-to

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

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


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


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

Server side selection

Client side selection

# Implementation Details

- RootCollection:
  - Meta data and the Tokens are stored in a simple **TTree**
    (You might play with it in an interactive Root session)
  - Selection is performed using **TTreeFormula**
  - Allows reading of **meta data sub-sets**
    (performance of client selections based on AttributeList)
  - Provides remote access via **rootd**
  - Allows **server side selection** (experimental)
    Simple C++ script that returns a TEventList to the client side iterator
- MySQLCollection:
  - Two iterator types: "FetchAll" and "FetchOne"
- Tokens are currently stored as strings:

```
[DB=DAF83A9C-9DC5-D711-9201-000347F31C25][CNT=helm_Root_1000000_100_Container]
[CLID=4E1F4DBB-1973-1974-1999-204F37331A01][TECH=00000202][OID=00000003-000CFA0F]
```

  - Not very resource thrifty (Root compresses them by a factor of 25)
  - Might be optimized by introduction of a lookup table

# Performance

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


while(iter.next())
   if( mySelection( iter.attributeList() ) )
      analyze( *iter);
```

*Performance draw back in case of homogenous collections:*
*Dynamic type check is performed in next(),*
*before client side selection*

- In order to allow a faster client side selection an alternative user interface has been recently introduced: **CollectionProxy**
- Allows for late type checking
- Will be released this weak: Many thanks to Ioannis!

```
CollectionProxy::Iterator iter = Coll.select("NHiggs>0");

while(iter.next())
   if( mySelection( iter.attributeList() ) )
      if( iter.isCurrentObjectOfType<Event>() )
         analyze( iter.ref<Event>() )
```

# Performance

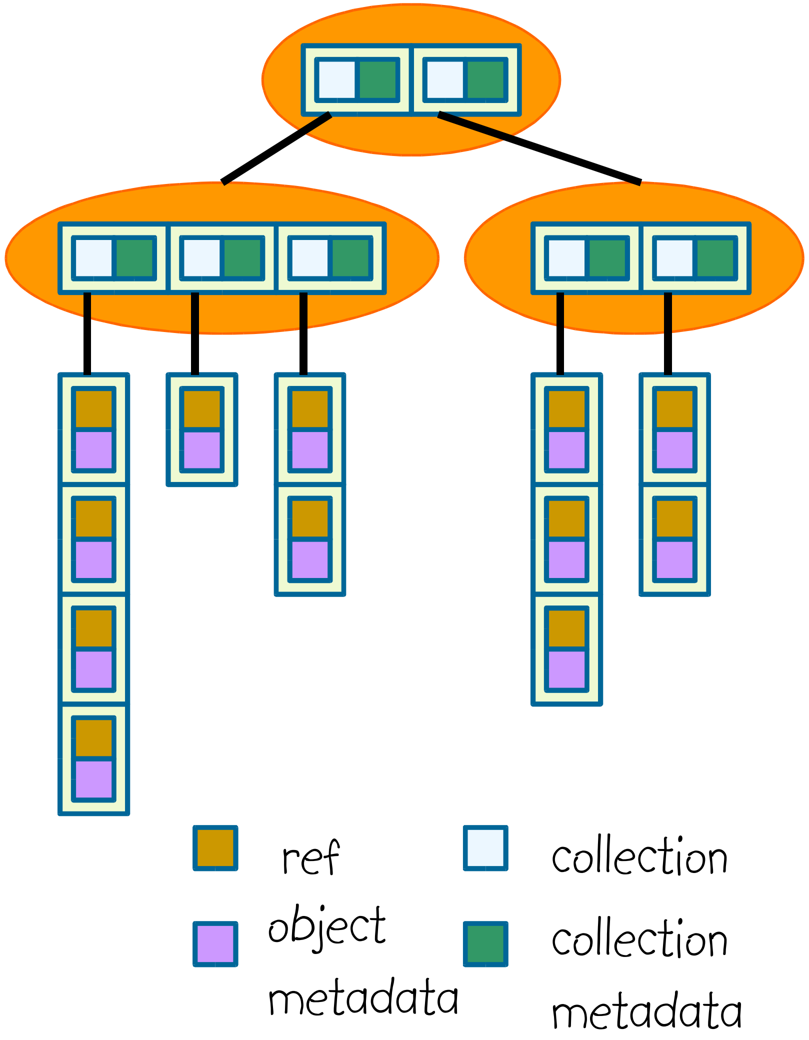
- Performance test: (preliminary)
  - $10^6$ tiny test objects in a local pool database file (size: 19 KB)
  - 100 meta data attributes per object
    (50 ints,50 floats randomly distributed)
  - Object access via the CollectionProxy interface (late type checking)
  - RootCollection file size: 215MB
  - Performed on a PC: P4 1.8 GHz 256 MB (ROOTMARKS = 419.7)
  - Thanks to Matthias for abusing his PC as rootd data server
  - MySQL server: lxshare070d.cern.ch

| Real Time [sec] spent to read $10^6$ entries | MySQL FetchAll | MySQL FetchOne | Root local file | Root rootd | Root rootd serverside selection | Implicit | Vanilla Root local file |
|---|---|---|---|---|---|---|---|
| Type check |  | 241 | 20 | 30 |  | 22 |  |
| Object access |  | 338 | 158 | 168 |  | 148 |  |
| Read 100 Attributes |  | 1924 | 41 | 65 |  |  | 39 |
| Read 2 Attributes |  |  | <1 | 14 |  |  | <1 |
| Select $\sim$1‰ and access objects (941) | 50 | 48 | 2 | 18 | 8 |  |  |

# Multi-Collections

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

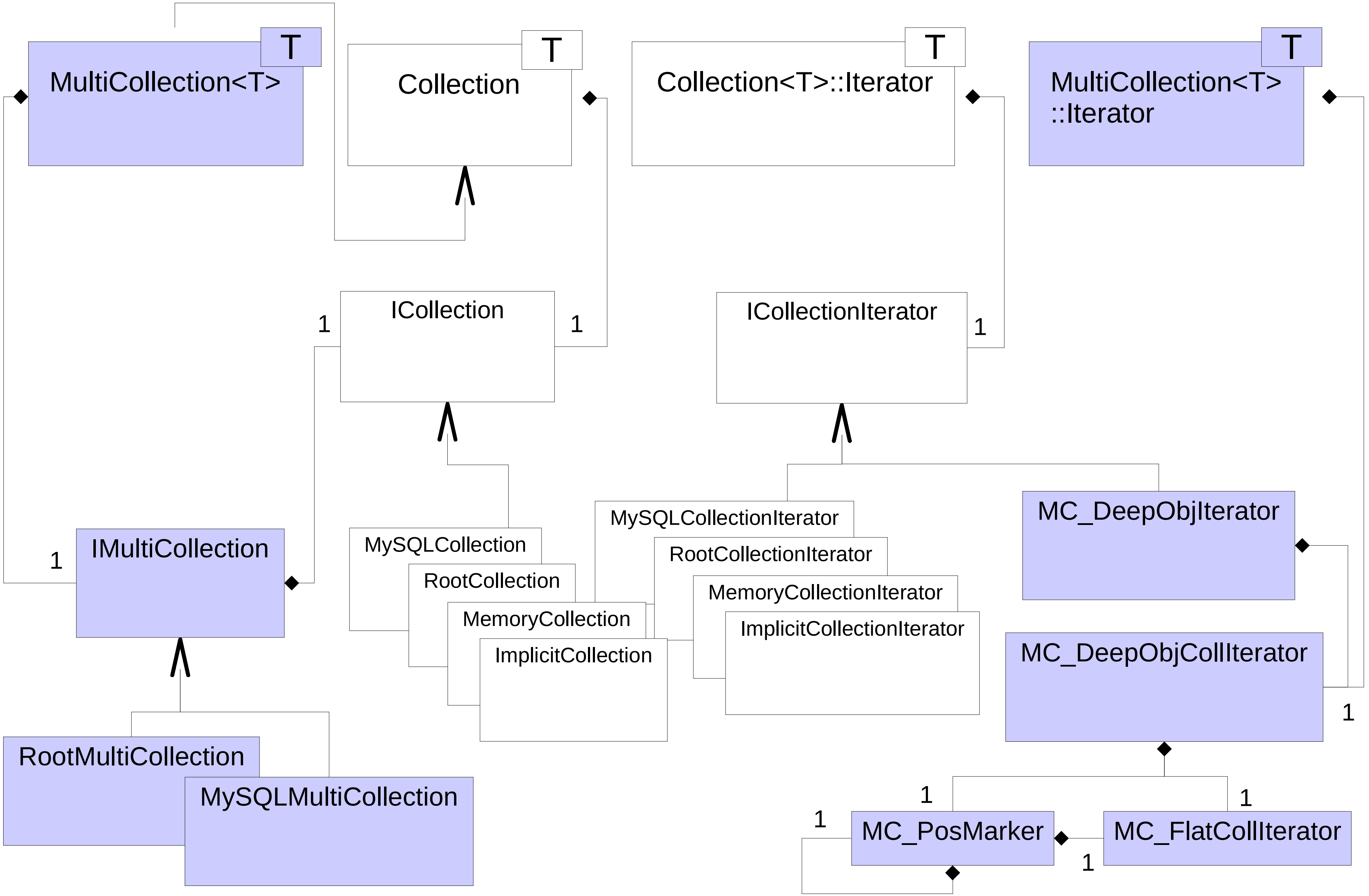
    ⇒ A hierarchical tree of collection can be build
- Allows complex user queries on collection and object meta data
  - all events with >4 selected electrons from runs with working ECAL and selected calibration, alignment setup
- Allows distributed storage of large object collections



# Multi-Collection Design

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

# Multi-Collection Design



# MultiCollection How-to

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

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

typedef Collection<Event>::Iterator EventIter;

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


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

# MultiCollection How-to

**The MultiCollection<T> interface provides  MultiCollection specific features**

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

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

typedef MultiCollection<Event>::Iterator RunIter;

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

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

# Summary / Outlook

- Explicit and implicit collections are ready to use
- To do
  - Further performance optimizations
    - Concurrent access ?
  - MultiCollections still in prototyping phase
    - Will be released this week
    - A simple test/example application is available
  - Different query syntax in Root and MySQL
- Outlook
  - Root server side selections?
  - Synchronization of collections containing the same objects but different meta data attributes (like TTree friends)
  - Indexing (even in Root, bit sliced indices for TTrees ?)
  - Collections directly based on DataSvc ?
- Many thanks to Ioannis, Kristo and Kuba for the help on performance optimizations