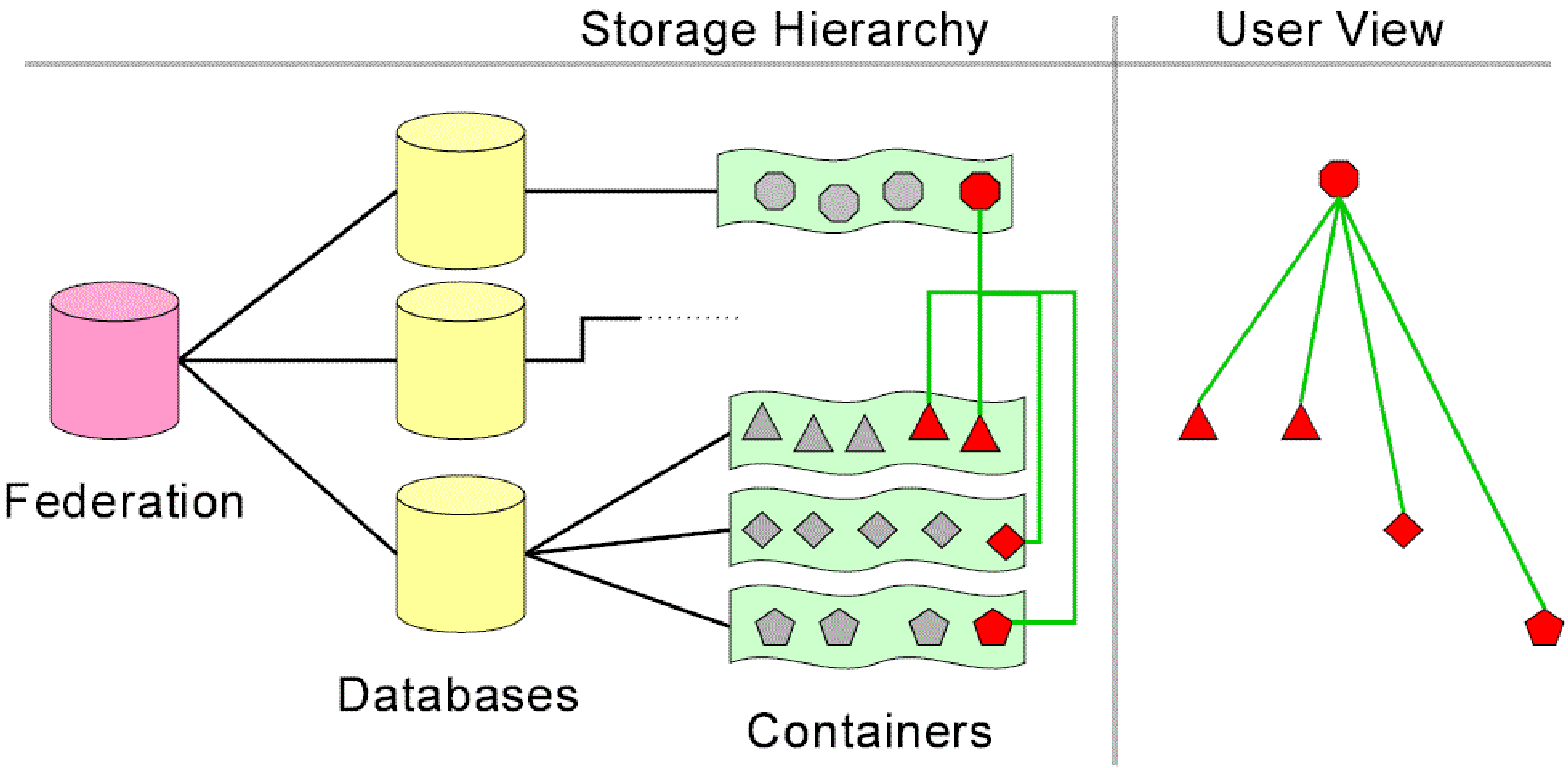
Object Databases as Data Stores for HEP

Part II

Dirk Düllmann
CERN IT/ASD, RD45

Physical Model and Logical Model

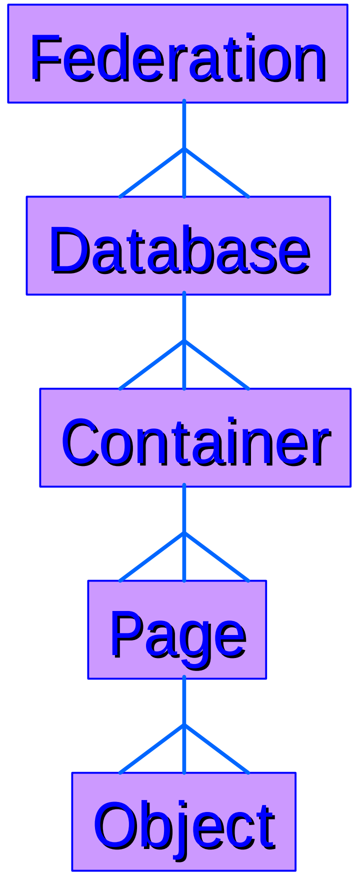


- Physical model may be changed to optimise performance
- Existing applications continue to work

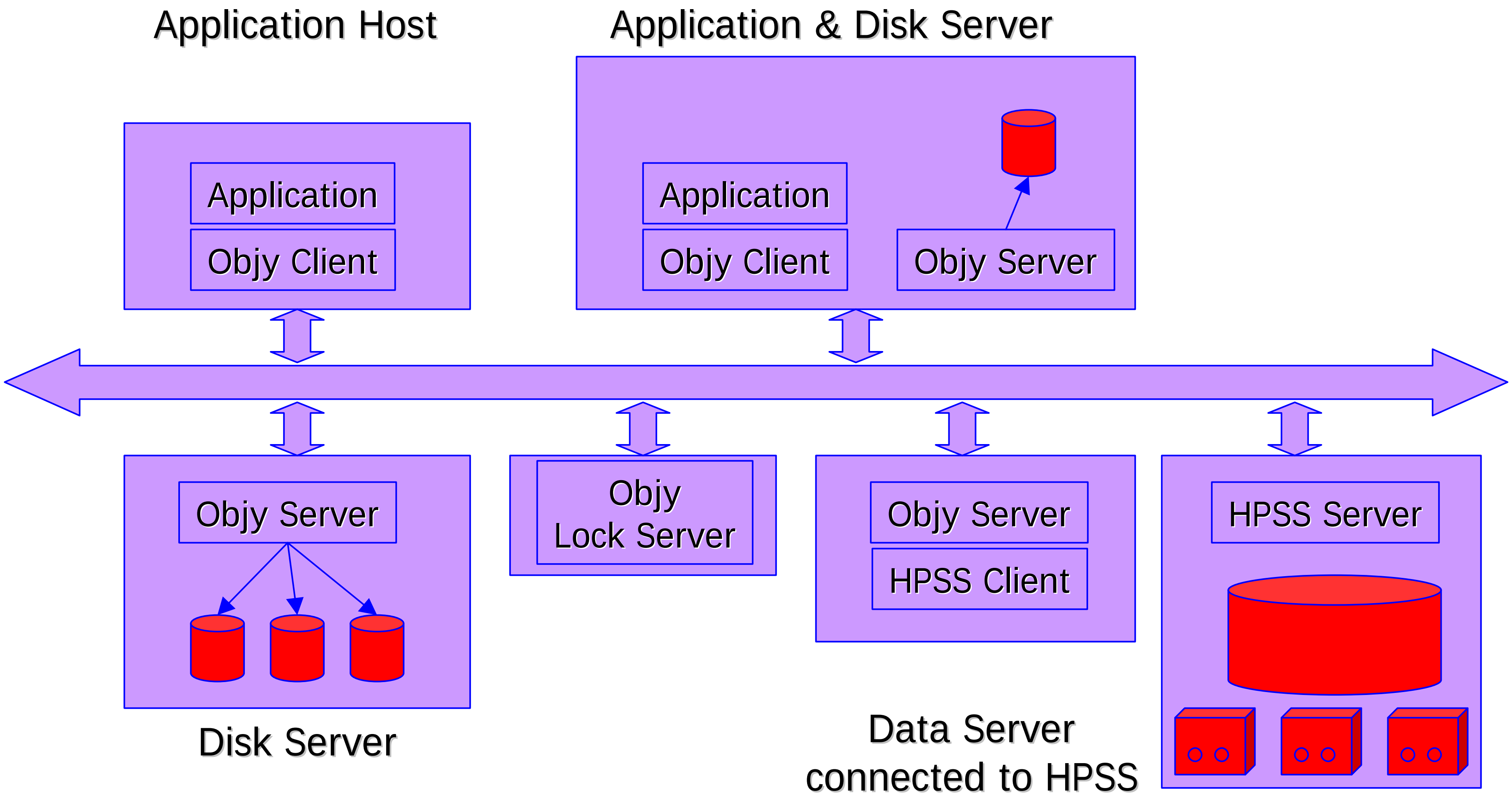
Objectivity/DB Architecture

- Architectural Limitations: OID size 8 bytes
- 64K databases
- 32K containers per database
- 64K logical pages per container
  - 4GB containers for 64kB page size
  - 0.5GB containers for 8kB page size
- 64K object slots per page
- Theoretical limit: 10 000PB
  - assuming database files of 128TB

- RD45 model assumes 6.5PB
  - assuming database files of 100GB
  - extension or re-mapping of OID have been requested



A Distributed Federation



Page Server & Container Locking

- Objectivity/DB
  - Page exchange between client and server
    - Page does contain not only requested data
    - In case of good clustering, it contains other objects that will be requested soon
  - Server only "knows" only about I/O pages
    - Thin server, fat client
    - Improved scalability
  - Locking on container level
    - All objects in one container are locked at once
    - Improved scalability and performance

Example I - populateDb

- /afs/cern.ch/sw/lhcxx/share/HepODBMS/pro/examples
- Objective:
  Populate a Database with Persistent Events
  - Define all involved classes
    - Simple object model consisting of :
      Event, Tracker, Track, Calo, Cluster
  - Create a Federation containing Databases and Containers
    - Tracking and calo data are kept in separate databases (files)
  - Create event objects
    - Events contain randomly generated tracks and clusters

Defining a persistent class

1. Define a C++ class in a .ddl file
   - very similar to a normal C++ header file
   - some restrictions apply (see next slides)
   - some additional features are available
2. Inherit from the persistent base class

```
class Event : public d_Object {
  public:
    int eventNr;
  ...
};
```

3. Introduce the new class to the database schema
   - Run to Objectivity Schema Processor ooddlx

DDL Restrictions

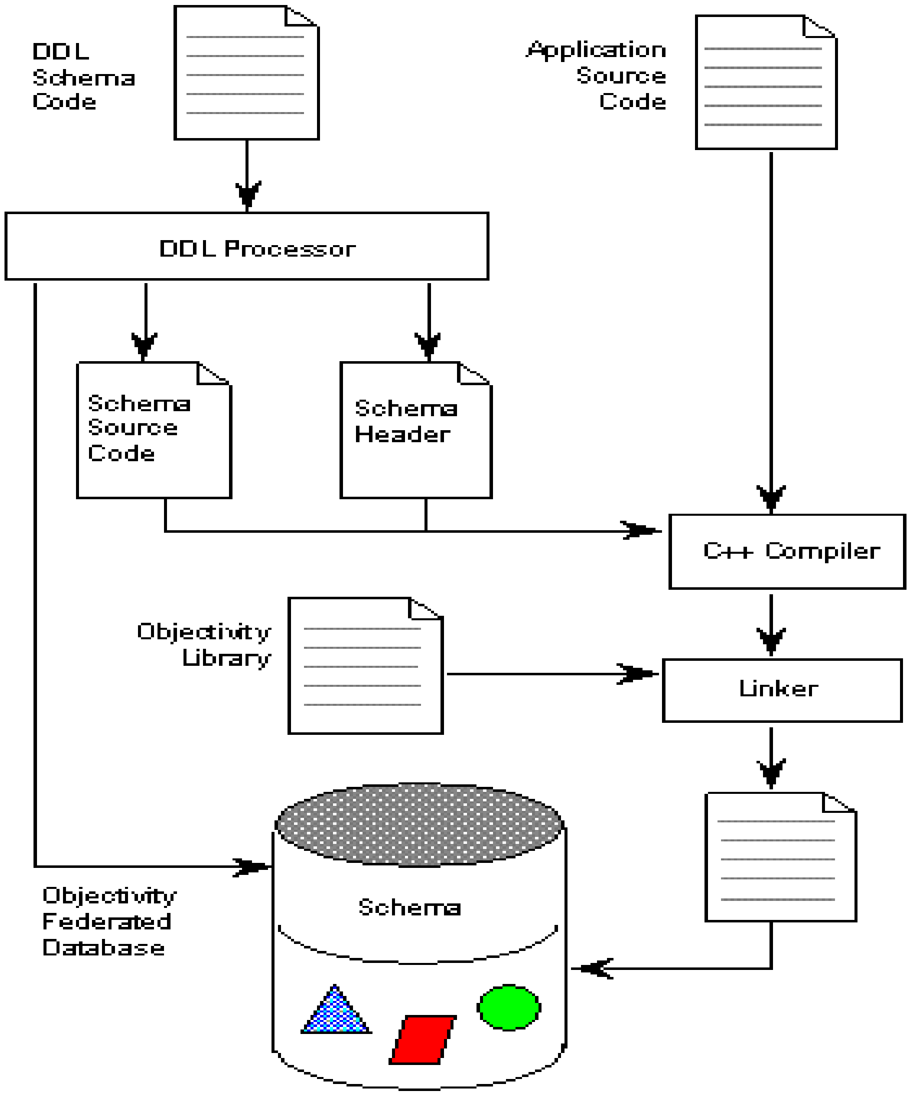
- Persistent classes may not:
  - Œ contain other persistent classes as data members
    - They may contain references to other persistent class though
    - Late (multiple-) inheritance from d_Object helps to keep transient and persistent classes in sync
  - • contain C++ pointers or references
    - Neither directly nor through embedded classes
    - replace C++ pointers by database smart pointers
- ❷ is the more intrusive change
    - **Type declarations** of pointers referencing persistent objects **have to be changed** for all clients of a persistent class.
    - Code that only **uses** these variables stays largely untouched.

DDL - Additional Features

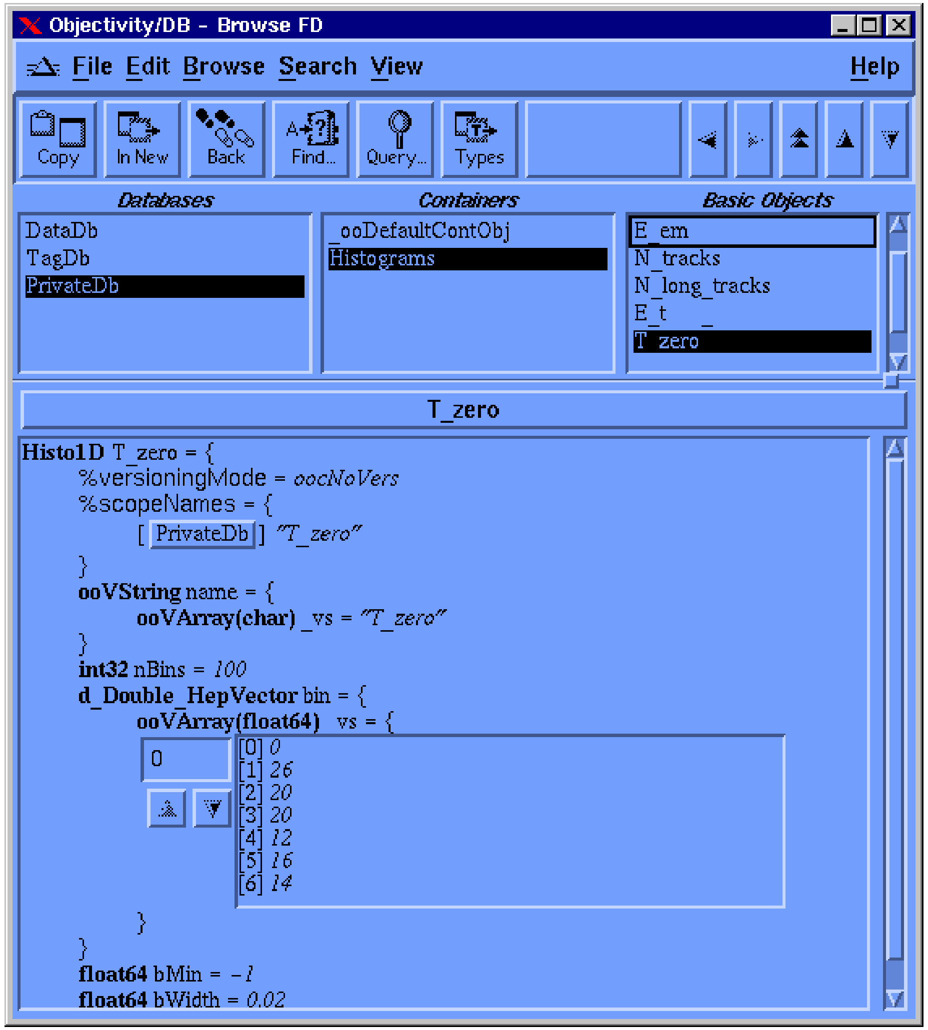
- Persistent classes may use in addition:
  - variable length arrays as data members
    - Example:
      A Tracker object contains a variable number of Track objects
      ```
      d_Varray<Track> tracks;
      ```
  - bi-directional associations
    - Example:
      Each Event has one Tracker, each Tracker belongs to one Event.
      ```
      d_Ref<Tracker> itsTracker <-> itsEvent;
      ```
  - 1-to-N or N-to-M associations
    - Example:
      One Run object keeps links to all its "N" events:
      ```
      d_Ref<Event> itsEvents[];
      ```

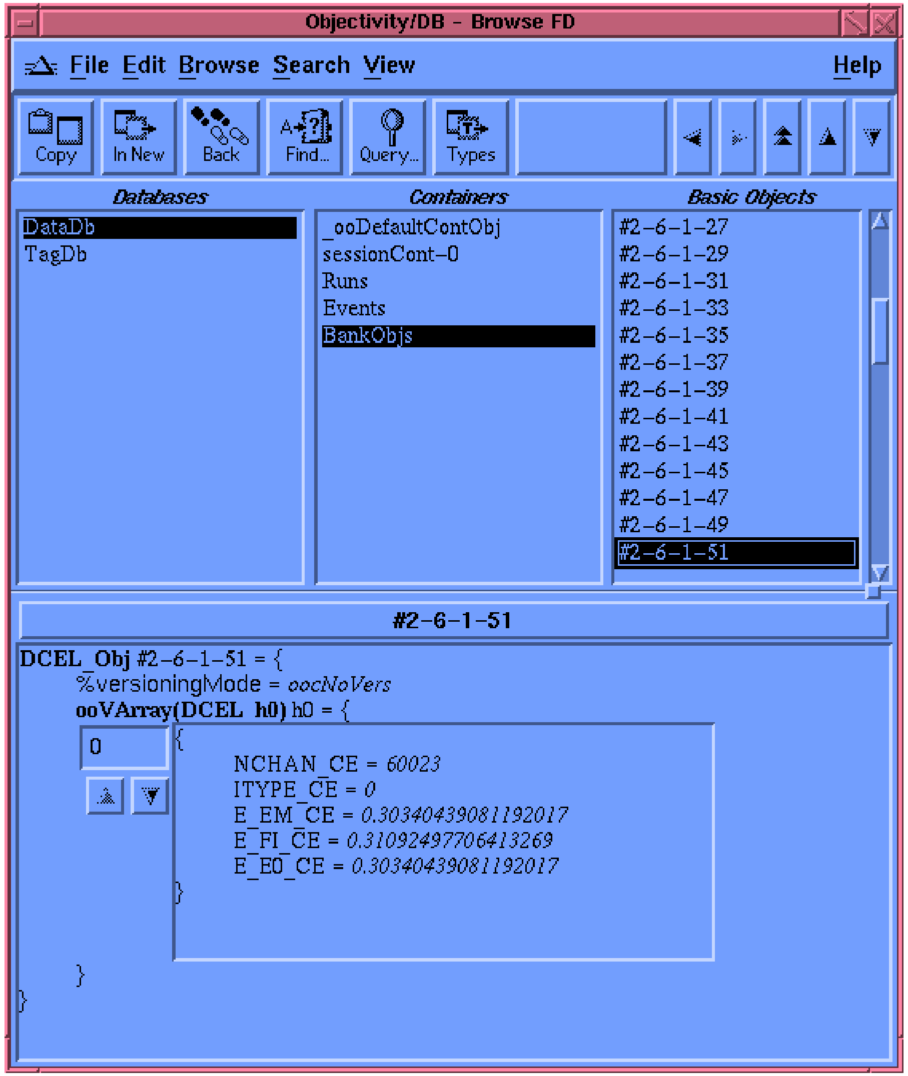
Schema Handling

- Definitions of persistent capable classes made in **.DDL** files
- ***ooddlx*** processor generates appropriate headers & source code
  - Schema is added to federated database
- Applications are built using generated files and the Objectivity library



Objectivity/DB Object Browser





HepODBMS Layer

- Goal:
  - Independence from vendor and/or release changes
    - Naming indirection of most prominent API classes
    - Provide missing features of the ODMG standard
  - HEP specific high level classes
    - Session control and diagnostic
    - Transaction control
    - Clustering Hint classes
    - Scalable collections (> 10^9 Objects)
    - Hierarchical Object Naming

Database Session Control

- HepDbApplication class - encapsulates db session control
  - Initialise the database session
  - Start/Commit/Abort Transactions
  - Set lock handling options, lock wait time, number of retries
  - High level interface that allows
    - open/create/find FDBs, DBs and containers
  - Provide job or transaction level diagnostics for
    - cache efficiency
    - disk I/Os
    - object accesses and updates
    - container and object extension
  - steered by API and/or environment variables
  - based on the ooSession class from Objectivity
    - small changes for Solaris, NT and transaction abort

Setting up a DB session using the HepDbApplication class

```
main()
  {
  HepDbApplication dbApp; // create an appl. object
  dbApp.init("MyFD");     // init FD connection
  dbApp.startUpdate();    // update mode transaction
  dbApp.db("UserDB");     // switch to db "UserDB"

  // create a new container
  ContRef histCont = dbApp.container("histos");
  // create a histogram in this container
  HepRef(Histo1D) h = new(histCont) Histo1D(10,0,5);

  dbApp.commit();      // Commit all changes
  }
```

Object Clustering

- The "new" operator provided by Objectivity allows to specify a clustering hint
  - may be a db, container or object reference
  - in which db, which container or close to which other object should the new object go
- HepODBMS contains classes to encapsulate the clustering strategy in "Clustering Hint" objects
  - clustering into single physical containers (< .5 GB for 8kB pages)
  - clustering into logical containers (infinite size, spread over several db files)
  - parallel writing without lock contention
  - parallel load balanced reading
  - definition of class based clustering through persistent objects

Clustering by Class

```
// class definition in Track.ddl
class Track : public d_Object {
    d_Double phi;
    d_Double theta;
    d_ULong  noOfHits;
// more stuff […]
public:
    static HepContainerHint clustering;
};
[…]
// define clustering at startup
Track::clustering = dbApp.container(“tracks”);
[…]
// use the clustering defined for tracks
HepRef(Track) aTrack =
      new (Track::clustering()) Track;
```

Clustering on a Larger Scale

- Objectivity limits containers to 64k logical pages
  - about 0.5 GB for 8kB page size

  Simple strategy:
  - check container size when a new object is created
  - create a new container if the current one approaches the limit
  - manage a ***persistent*** list of containers
- Objectivity locks on container level
  - Reduce lock contention in multi-processor environments

  Simple strategy:
  - assign one container per process
  - manage the list of containers as a logical super-container
- **`HepClusteringHint`** class implements both

Persistent Clustering & Parallel Writers

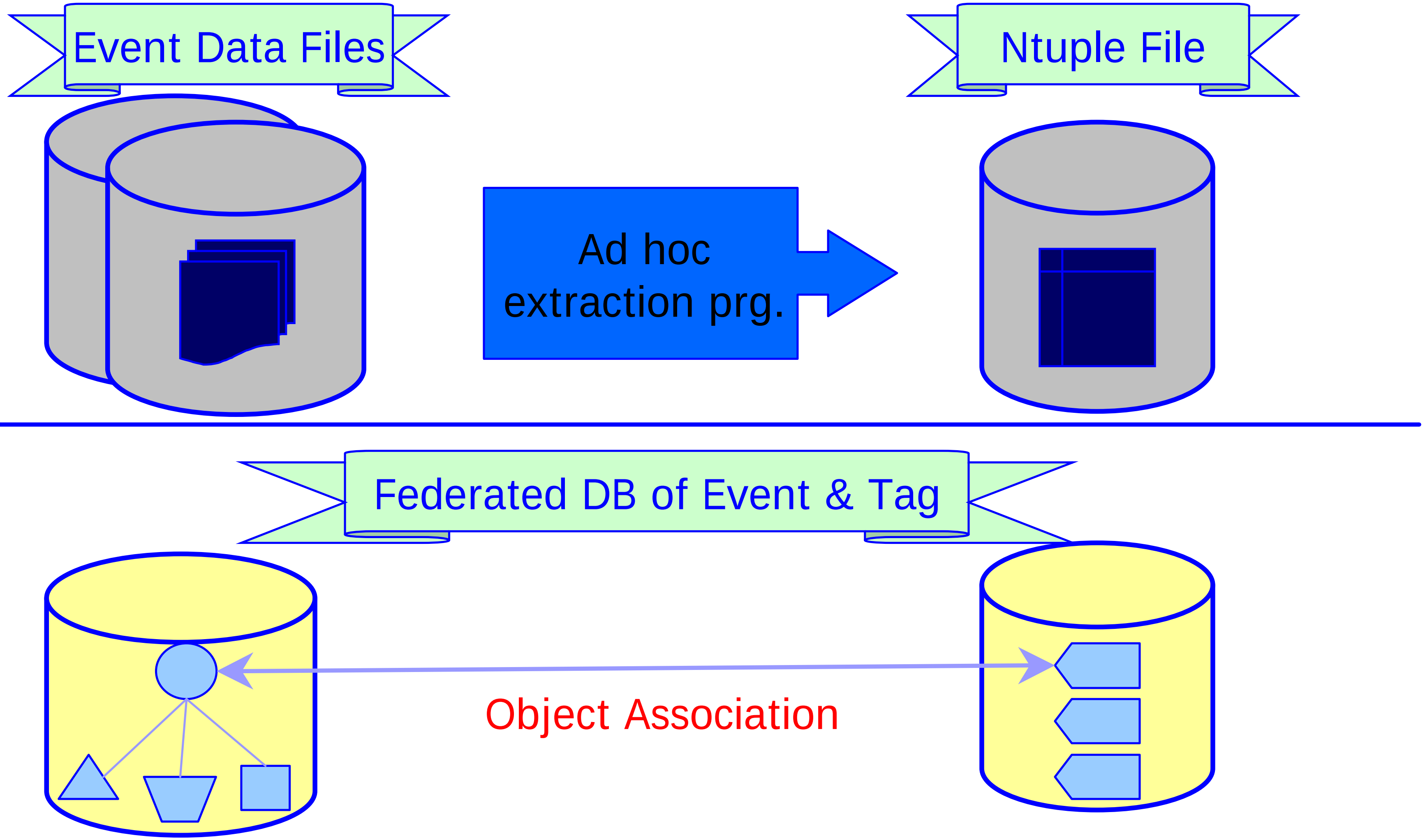
```
// class definition in Track.ddl
class Track : public d_Object {
    d_Double phi;
    d_Double theta;
    d_ULong  noOfHits;
// more stuff
public:
    static HepClusteringHint clustering;
};
// find the persistent clustering object for tracks
if ( !Track::clustering.find(“tracks”))
    Track::clustering.create(“tracks”));

HepClusteringHint::setParallelWriterMode(noOfProcs,myID);
// clustering use spread all over the source code
HepRef(Track) aTrack = new (Track::clustering()) Track;
```

Persistent Analysis Objects

- LHC ++ uses Objectivity/DB to
  - Provide persistency for Histogram, Tag and Event Data
  - Exchange objects between modules in a distributed environment
    - Object identifiers (OIDs) allow to **directly access** objects
- Setup before LHC++ 99a
  - Each user works in a private database (e.g., in AFS space)
    - Analysis programs run against local data
- Goal: Central Objectivity Service
  - Shared federated database per experiment
  - Common data is available experiment wide
    e.g. tag collections, simulated events or test beam data

Ntuple versus TagDB Model



Purpose of Using Tags

- Tags are mainly used to speedup selections
  - Tag data is much better clustered than the original event tree but still logically attached!
- Tag Collections define Event Collections
  - Tag Collections are only a special case of an Event Collection
- Tag attributes may be visualised interactively
  - without the need to write any code
- Association to the Event may be used to navigate to any other part of the Event
  - even from an interactive visualisation program

Example II: createTag

- Objective:
  Create a collection of all events which contain at least two oppositely charged tracks with p_t > 1 GeV

❶ Loop over all events

❷ Find tracks with p_t > 1

❸ Keep references to matching events in a persistent collection

❹ Define some useful variables in a tag for later interactive analysis

Collections of Tags

- Generic Tags
  - Generic content: No need to define a new persistent class
  - May use predefined types: float, double, short, long, char
  - Additional attributes may be added later
  - Interactive display using IRIS Explorer

```
// create a new tag collection
GenericTag highPt(“high pt events”);

// define all attributes of my tags
TagAttribute<long>   evtNo(highPt,"eventNo");
TagAttribute<double> ptPlus(highPt,”ptPlus");
TagAttribute<double> ptMinus(highPt,”ptMinus");
```

Filling a Tag Collection

- Tag Attributes are used **just like other C++ variables**

```
TagAttribute<long>   evtNo(highPt,"eventNo");
TagAttribute<double> ptPlus(highPt,"ptPlus");
TagAttribute<long>   nTracks(highPt,"nTracks");

if (highPtTracks > 2)
{
  // create a new tag and store the event reference
  highPt.newTag(evt);

  // define its tag attributes
  evtNo     = evt->eventNo;
  ptPlus    = evt->tracker.tracks[plusTrack].pt;
  nTracks   = evt->tracker.tracks.size();
}
```

Analysis using Tag & Event Data

- Select on tag attributes and **directly access event data**

```
for(more=highPt->start(); more!=0; more=highPt->next())
{
  // apply more cuts
   if (ptPlus > 3 &&  nTracks < 10)
   {   // … fill histograms from the tag…
        cout << “eventNo: “ << eventNo << endl;
        ptPlusHisto->fill(ptPlus);
        ptMinusHisto->fill(ptMinus);

        HepRef(Event) evt;
        highPt->getEvent(evt);
        // … but also using data from the event.
        nClusterHisto->fill(evt->calo.clusters.size());
  }
}
```

Hierarchical Naming

- Need a way to organise/lookup objects which are entry points into disconnected parts of our object model
  - e.g. Event Collections or Histograms
- Each user might need to reference thousands of those objects
  - Flat name space would become difficult to manage
  - Tree like approach (as used in file systems) is familiar to most users
- At the RD45 Workshop in February/April
  - Hierarchical naming service for (any) persistent object
  - Agreement on the main requirements

Requirements

- External Naming
  - any persistent class may be named
  - no change to object schema
- Independent of Physical Model
  - named object may be anywhere in the FD
  - similar approach to bookmarks in Netscape
- Multiple Names for the same object
- Scalable
  - One hash table per directory

Do not replace associations with names!

HepNamingTree

- Two class implementation using Objy
  - HepNamingNode (persistent)
  - HepNamingTree (transient)
- HepNamingTree provides all methods to navigate within the tree structure and to create new nodes
  - makeDirectory(path), changeDirectory(path), removeDirectory(path)
  - nameObject(objRef,path), findObject(path), removeName(path), removeObject(path)
  - startItr(), nextItr()

Current and Future Use

- Implementation available in HepODBMS
  - Used e.g. by HTL to provide named Histograms
- BaBar is using a similar approach for their event collections
- LHC++ will need to provide a more flexible way to deal with histograms in shared federations
  - currently based on physical model
  - database and container browser
  - support for logical naming starting with 99a release

Improvements wrt. Old Class

- Switched from scope names to ooMaps to implement directory hash lookups
  - better control over tuning parameters for hashing
- Using ooMap solves also inconsistency problems if named objects are deleted through OID
  - ooMap uses a bi-directional association to the named object (predefined in ooObj)
  - When an object is deleted Objy deletes any associated ooMap entries as well.

Limited Support for Meta Data

- Naming Node keeps some Meta Data
  - always
    - time of creation
    - external object type
  - optional
    - extendible list of property value pairs (strings)
    - e.g. Comment = "test";
- Basic support for finding objects by property
  - iteration over directory or subtree
  - application of search predicate object

Integration with HepDbApplication

- Default HepNamingTree naming is available from the HepDbApplication object
- User will be put into a private "Home Directory" at startup

```
typedef h_seq<Event> EventCol;
HepDbApplication app;
app.init("fdBootName"); // implicit cd /usr/$USER/
app.naming().changeDirectory("test-beam");
evtCol = app.naming().findObj("inputEvents");
EventCol::iterator it;
for (it = evtCol.begin(); it != evtCol.end(); it++)
```

C++ Example: Yet Another Shell

- Simple C++ example program showing how to use the naming interface
  - navigation in the tree
  - creation/deletion of named objects
  - printing/dumping of objects by name
  - Source comes as part of the HepODBMS example tree

- DEMO?

Java Interoperabilty

- Simple Interactive Tree Browser
  - Ported the naming tree classes to Java
    - read only access for now
  - less 200 lines using the Swing GUI classes
  - No native callbacks, just using the Objy/Java binding
- Few problems during the port
  - My first Objy/Java program :-)
  - Had to "use" an ooMap instead of inheriting from it
    - ooMap is "final" in Java
  - Difficulties to obtain an the OID of an object if the corresponding class does not exist in Java
    - Java binding sometimes is too typesafe to implement e.g. generic browsers
- DEMO?

HepODBMS Collections

- Why yet another set of collections?
  - Our requirements are different
    - very large collections
      - efficient set operations
      - efficient iteration order
    - problems with exposing the underlying implementation of many different collection types
    - we will need some integration of queries
  - Collections and Iterators are a MAJOR part of the visible interface of an ODBMS
    - Extension of the HepODBMS insulation layer
    - Minimise the code changes after changing the ODBMS

Collection Implementation

- Templated collection of any kind of persistent objects
  - typedef h_seq<Event> EventCollection;
- Single class interface
  - STL interface independent of implementation
    - Single User visible collection class : h_seq<T>
    - Single STL like iterator: h_seq<T>::iterator
  - Uses hybrid of templated classes and delegation
  - User extendible through strategy objects
- Currently Implemented Strategies
  - vector of references (based on STL)
  - paged vector of references
  - single container
  - group of containers

Reader Example

```
// find a collection using the naming service
EventCollection evtCol = app.naming().findObj( “/usr/dirkd/collections/myEvents ”);

// STL like iterator
EventCollection::const_iterator it;

it = evtCol.begin();
while( it !=  evtCol.end() )
    {
     cout << "Event: " << (*it)->getEventNo() << endl;
     ++it;
    }

// support for (some) STL algorithms
 int cnt=0;
 count(evtCol.begin(),evtCol.end(),1,cnt);
```

Writer Example

```
EventCollection evtCol( "collections/myEvents","container");

HepRef(Event) evt;

for (int i=0; i<500000; i++)
  {
  // create a new event using the clustering hint of the sequence
  evt = new(evtCol.clustering()) Event;

  // store the new object ref in the sequence (only needed for ref collections)
  evtCol.push_back(evt);

  // fill the event
  evt->setEventNo(i);
 }
```

The End