OBJECT DATABASES AS DATA STORES FOR HIGH ENERGY PHYSICS

Dirk Düllmann
CERN IT/ASD & RD45, Geneva, Switzerland

Abstract
Starting from 2005, the LHC experiments will generate an unprecedented amount of data. Some 100 Peta-Bytes of event, calibration and analysis data will be stored and have to be analysed in a world-wide distributed environment. At CERN the RD45 project has been set-up in 1995 to investigate different approaches to solve the data storage problems at LHC. The focus of RD45 soon moved to the use of Object Database Management Systems (ODBMS) as a central component. This paper gives an overview of the main advantages of ODBMS systems for HEP data stores. Several prototype and production applications will be discussed and a summary of the current use of ODBMS based systems in HEP will be presented. The second part will concentrate on physics data analysis based on an ODBMS.

1 INTRODUCTION

1.1 Data Management at LHC

The new experiments at the Large Hadron Collider (LHC) at CERN will gather an unprecedented amount of data. Starting from 2005 each of the four LHC experiments ALICE, ATLAS, CMS and LHCb will measure of the order of 1 Peta Byte (10^{15} Bytes) per year. All together the experiments will store and repeatedly analyse some 100 PB of data during their lifetimes. Such an enormous task can only be accomplished by large international collaborations. Thousands of physicists from hundreds of institutes world-wide will participate. This also implies that nearly any available hardware platform will be used resulting in a truly heterogeneous and distributed system.

The computing technical proposals of LHC experiments do not only require access to the data from remote sites but in addition ask for distribution of the data store itself to several regional centers.

Experiment	Data Rate	Data Volume
ALICE	1.5 GB/sec	1 PB/month (1 month per year)
ATLAS	100 MB/sec	1 PB/year
CMS	100 MB/sec	1 PB/year
LHCb		400 TB/year

Table 1: Expected Data Rates and Volumes at LHC

1.1.1 HEP Data Models

HEP data models are typically very complex. The number of different data types (e.g. bank types or classes) needed to describe the event data of a large HEP experiment reaches easily several hundreds.

A single event often contains thousands of individual data structures (e.g. banks or objects) and a large number of relations between those data items (e.g. links or pointers).

The design and documentation of HEP data models is an essential and non-trivial task during the experiment software development process. Especially the data model of stored data structures is of significant influence on many other software developments. Since the definition of this data is shared between multiple subsystems (e.g. data acquisition, event reconstruction and physics analysis) with very different access patterns, it is often difficult to fulfil all flexibility and performance requirements in a single design.

All LHC experiments exploit Object Oriented (OO) technology to implement and maintain their very large software systems. Today most software development is done in C++ with a growing interest in Java. The data store therefore has to support the main concepts of these OO languages such as abstraction, inheritance, polymorphism and parameterised types.

1.1.2 The RD45 project

From the analysis of the LHC data management requirements, it seemed clear that existing solutions based on FORTRAN and sequential files such as ZEBRA would be inadequate for the LHC era. At CERN, the RD45 project was started in 1995 to investigate new solutions to the LHC data management problems. After an evaluation of different technology choices such as language extensions for persistency, light-weight object managers, object request brokers, RD45 focused rapidly on a solution consisting of a commercial Object Database Management System (ODBMS) coupled to a Mass Storage System (MSS).

2 OBJECT DATABASE SYSTEMS

2.1 ODBMS and Programming Languages

The natural view of data for application programmers is that of a network of objects in the application memory space. Figure 1 shows a simple example of such a configuration describing part of the event data of some HEP experiment.

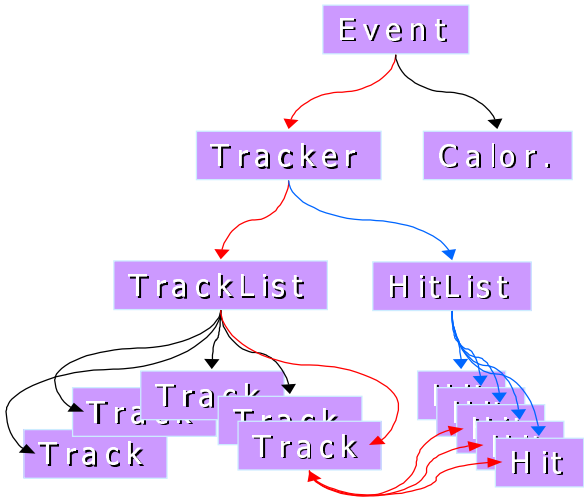

Figure 1: Simple Logical Model of Event Data Objects

A large fraction of any programs operation will consist of navigation within this net to find needed objects, change their state, create new objects or delete existing ones, e.g. navigation from the event object to its tracking detector, retrieving a particular track from its track list, and navigation to all associated hit objects to perform a track refit.

For transient data - objects that only exist within the context of a single program - this navigation is well supported by OO languages. The pointer and reference types provided by C++ and Java allow efficiently creating and maintaining complex in-memory object networks. When I/O operations have to be performed on such a network, e.g. because object data need to be stored or to be exchanged between two programs, the support from today's OO languages is rather limited.

2.1.1 Object Input / Output

Neither C++ nor Java provides an I/O sub-system capable of dealing directly with object trees or nets. The significant burden of providing this capability within a particular application is therefore left to the application programmer. In particular the programmer is left with the task to maintain two distinct copies of each data item that has to be stored:

- "in-memory" data – an object net with pointers or references describing relations between objects
- "on-disk" data – a sequence of bytes in one or more disk files

Since the representations of both copies are necessarily different (e.g. for heterogeneity reasons) the application code has to perform a quite complex transformation between these copies. In particular the application programmer has to *explicitly* code:

- When (and how often) to perform the transfer?
  The user has to explicitly trigger any data transfer (e.g. read data from disk before first use of some program variable, write to disk after its last update) and any re-transfer in case that the on disk data might have been changed by another program. Since in a complex HEP application it is difficult to predict which data items will be used, often simply all event data is transferred. This approach may results in degraded performance.

- How to perform the transformation?
  The network of objects has to be decomposed into single objects (or even single attributes) which may be stored directly. Any pointers or references have to be handled by special code that translates them into some storable format.

Often more than one third of the application code is necessary to perform this in principle well-defined mapping between disk and memory. In practice it turns out that maintaining this I/O related code is not only tedious but also relatively error prone. Many software problems in fact result from the lack of consistency between "in-memory" and "on-disk" data copies. Since the I/O subsystem is only loosely bound to the programming language, typically as a set of library routines, consistency can not be maintained automatically. In addition the large fraction of I/O related code obscures the real services provided by a given class and makes it sometimes impossible to understand the objective of a code fragment without knowing about the details of the I/O related code.

2.1.2 Object Persistency

Object databases feature a very different I/O model. The starting point is a so-called "tight language binding" in which the consistency between data on disk and in memory is maintained by the database system. The programmer of an ODBMS application in fact only deals with a single copy of the data, the object in the programming language itself. The database system is responsible for maintaining the semantics of *persistent objects:* objects that retain all their state information between two program contexts.

Any data transfers needed to synchronise program objects with the data on disk are performed automatically by the ODBMS on a per object basis. Only data of those objects that are actually used

by a particular program are transferred which might result in a drastically increased application performance compared to reading an entire event.

ODBMSs transfer complete objects from disk[1] into application memory. The state of all data members and the dynamic type of a persistent object are retained. Virtual function calls through a base class pointer behave as expected (support for polymorphism). ODBMSs fully support abstract data types and allow creating persistent types using templates.

2.2 Navigational Access

During the creation of a new persistent object the database assigns a unique Object Identifier (OID) to each object. When an object is accessed from an application program, its OID is used by the database to find the object data in the disk store. The different ODBMS products vary largely in their OID implementation ranging from direct extensions of virtual memory pointers (Objectstore) to structures that refer more directly to a physical location in the disk store (Objectivity/DB).

OID may themselves be embedded as data members in persistent objects, which allows implementing a relation between two persistent objects (association). Most ODBMS products allow creating in addition to uni-directional 1-to-1 relations 1-to-n associations (between one object and a varying number of other objects) and bi-directional associations.

2.2.1 Smart Pointers

An application programmer typically does not use OID values directly but rather through so-called smart pointer types which allow to implement "on-demand " I/O. Smart pointers are small objects which are provided by the database implementation that behave semantically as a normal object pointer. E.g. in C++ they allow to use the "->" operator to access an object attribute or to call a method. During this access the smart pointer will call back the database system to retrieve object data from disk if necessary. ODBMSs maintain an object cache in the application program (client side cache) to increase the performance of repeated accesses to the same objects.

```
Collection<Event> events;           // an event collection
Collection<Event>::iterator evt;    // a collection iterator

// loop over all events in the input collection
for(evt = events.begin();  evt != events.end(); evt++)
     {
     // access the first track in the tracklist
     d_Ref<Track> aTrack;
     aTrack  = evt->tracker->tracks[0];

     // print the charge of all its hits
     for (int i = 0; i < aTrack->hits.size(); i++)
          cout  << aTrack->hits[i]->charge
               << endl;
     }
```

Example 1: Navigation using a C++ program

[1] Some databases like Objectivity/DB treat large embedded attributes in an on-demand fashion. E.g. large embedded arrays will be read from disk only if the application accesses at least one array element.

As a consequence of the tight binding of ODBMS to the programming language the application programmer perceives the database as a natural extension of normal “in memory” objects. Using the database one can create networks of objects with indefinite lifetime and efficiently navigate among them.

2.3 Schema Handling

Before any instances of a persistent C++ class may be created, the class layout has to be registered with the database. The information about attribute position, name and type of attributes is used e.g., to provide the correct memory layout for an object on all different platforms.

For the C++ language this class registration is performed using a pre-processor program which scans class definitions of persistent classes in Objectivity’s Data Definition Language (DDL) and generates C++ header and implementation files for persistent classes. The generated header files define the class interface for clients of a persistent class. The generated implementation files contain C++ code which implements smart-pointer types and various collection iterators for each persistent class. All generated files are then compiled together with any other application code and linked against the Objectivity library to form a complete database application.

The set of all class definitions - also called database schema - is stored centrally in the federation file together with the catalogue of all database files.

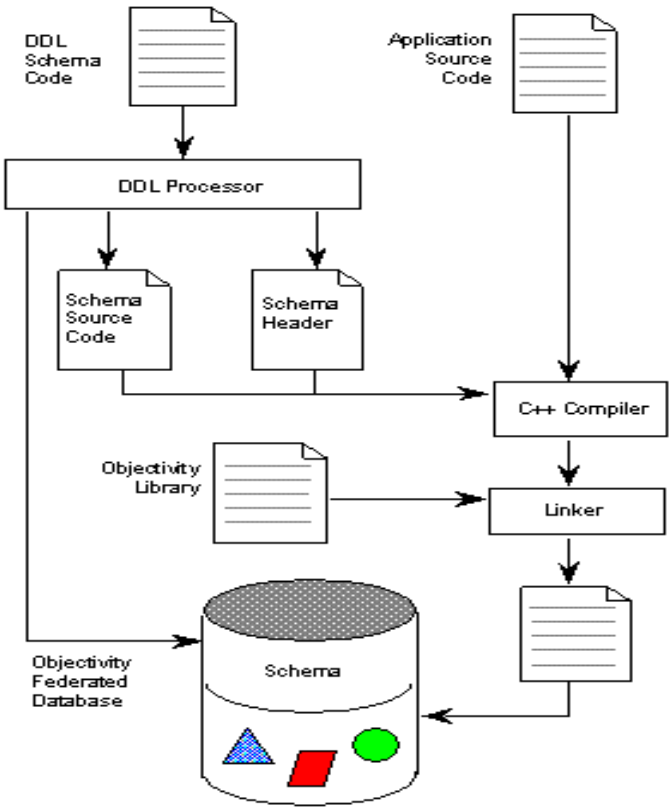

Figure 2: Schema Capture and Build Process

2.4 Consistent Access to Shared Data

ODBMS products provide support for multiple clients working on the same data store and concurrently updating it. Usually ODBMSs introduce a central “lockserver” that is responsible to co-ordinate the updates by keeping a lock table for the whole system. To guaranty data consistency in the

system, object databases use (as e.g. relational database) the notion of transactions. Any data change is part of a transaction with the ACID properties. Some vendors provide special transaction modes in which multiple reading processes can coexist with a single writer per locked entity which may enhance the concurrency behaviour. This feature is very useful for many HEP applications:

- **Simplified Support of Parallel Applications**
  Many sub-systems of LHC experiments like data acquisition, filter- and reconstruction farms and distributed simulation will only achieve their performance requirements by making use of parallel processing. These systems will profit form the build-in concurrency support of the data store.

- **Data Consistency and Reduced Storage Size**
  During HEP analysis the current practice involves many redundant copies of the original data. Starting from the fully reconstructed data one has to repeatedly create copies of the original data selecting a subset of the events, a subset of the data within an event or both. The reason for these redundant data copies is the inability of today's I/O systems to effectively access sparse data, to re-cluster data or to add additional data to an existing store. In addition to the often significantly increased total storage size these data copies may lead to subtle consistency problems. Since any new reconstruction of the original events invalidates these copies, they have to be manually updated before results of different analysis groups can be compared.

2.5 Physical Store Implementation

All ODBMS products use a multilevel hierarchy to implement the possibly distributed physical store. Objectivity/DB for example uses a hierarchy of five different levels to implement the physical storage. The topmost level - the Federated Database - keeps system wide information about the shape of persistent classes within the store. In addition the catalogue of physical location of all data files is kept centrally in this file. Each federation consists of up to 64k databases - files that contain the actual data of all stored objects. Each database is structured internally into up to 32k "containers" - contiguous areas of objects within a database file. Containers consist themselves of up to 64k pages containing the actual object data. The starting position of object data on a page is called slot and uniquely defines a particular object on the page.

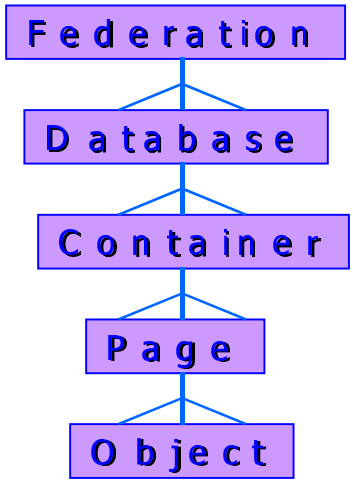

Figure 3: Storage Hierarchy in Objectivity/DB

The structure of the physical store hierarchy is directly reflected by the internal structure of the OID implementation. A 4-tupel of 16 bit numbers uniquely references any object within the store.

Figure 4: Object Identifier Implementation used by Objectivity/DB

2.5.1 Separation of Logical and Physical Storage Model

The concept of OIDs allows to directly access any object in the potentially large distributed store without requiring the application programmer to maintain store implementation details like e.g. file and host names. Since this information about the physical layout of the store is kept only once, centrally by the database, it is much easier to change the storage layout without compromising existing applications. One may change the location of a particular file to a new host by moving the data and changing the catalogue entry. Since the catalogue is shared by all database applications, they will use the data from the new location without any change.

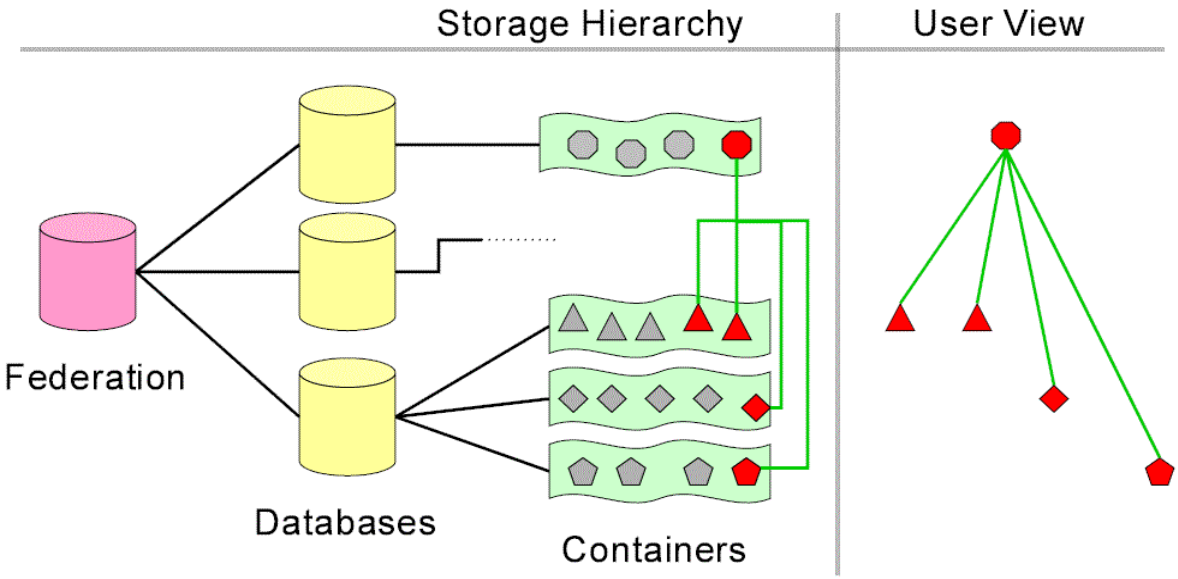

Figure 5: Physical Storage Hierarchy and Logical User View

2.5.2 Data Clustering and Re-Clustering

An important feature offered by several ODBMS products is the support of object clustering. When a persistent object is created, the user may supply information where the object should be placed within the physical storage hierarchy. In C++ a clustering hint may be passed as an argument to the new operator. The statement

```
d_Ref<Track> aTrack = new(event)  Track;
```

will for example instruct the database to create a new persistent track object physically close to the event object. This ability to cluster data on the physical storage medium is very important for optimising the performance of applications which access data selectively.
The goal of this clustering optimisation is to transfer only useful data from disk to the application memory (or one storage level below: from tape storage to a disk pool). Grouping data close together

that will later be read together can drastically reduce the number of I/O operations needed to acquire this data from disk or tape. It is important to note that this optimisation requires some knowledge about the relative contributions of different access patterns to the data.

An simple clustering strategy is the “type based clustering” where all objects of some particular class are placed together: e.g. Track and Hit objects within an event may be placed close to each other since both classes will often be used together during the event reconstruction.

For physics analysis this simple approach is probably not very efficient since the selection of data that will be read by a particular analysis application depends more on the physics process. In this case one may group the analysis data for a particular physics process together.

2.5.3 Architectural and Practical Storage Size Limitations:

The theoretical storage size constraints in the current Objectivity implementation result from the implementation object identifier in four 16-bit words. Each OID consists of four parts:

- **16 bit database number**
  resulting in up to 64K databases per federated database
- **15 bit container number (one bit used internally)**
  32K containers per database
- **16 bit page number**
  64K logical pages per container
- **16 bit slot number**
  64K possible object locations per page

Assuming that the maximum number of database pages is allocated one obtains a maximum container size of 4GB (for 64kB page size) or 0.5GB (for 8kB page size). The theoretical limit for the total size of a federated database would amount in this case to some 10 000PB.

In the current implementation the maximum size of a federation is still significantly constrained by the maximum file size in the system. This calculation of a theoretical limit assumes database sizes of 128TB. Since in the current implementation each database is represented by a single file, such large databases are not practical. Assuming a maximum file size (and therefore maximum database size) of 100GB one obtains a maximum federation size of 6.5PB. Since this might be a limitation for LHC experiments, RD45 has requested to modify the mapping between database and physical files to allow multiple files per database.

2.6 Limits and Scalability Tests

Various tests have been performed to check the scalability of Objectivity/DB. Federated Databases of 0.5 TB have been demonstrated and multiple federations of 20-80GB are used today in production, some of them also exploiting the parallel I/O capabilities. The NA45 experiment for example performed their reconstruction and formatting on 32 filter nodes writing in parallel into a single federated database. Even more concurrent database clients have been simulated in a recent test performed at Caltech. On a HP Exemplar supercomputer up to 240 concurrent readers have been used successfully against a single federated database.

3 OBJECTIVITY SPECIFIC FEATURES

In addition to the generic ODBMS functionality that is implemented by most vendors the products differ significantly in their data distribution and replication features. The following section describes these specific features and their potential use in HEP applications in more detail.

3.1 Federations of distributed databases

Applications are connected to one Objectivity federated database at a time. A federated database consists of many database files that may be located on different hosts connected by local or wide area network. Each client application communicates directly with the hosts that serve data used by the application. Any data transfer takes place between the client process and the Objectivity page server (ooams) which runs on each data-serving host[2]. Consistency for concurrent access is provided through one or more lockserver processes per federation. Before any data is read or modified, the client connects to this lockserver to obtain a suitable lock.

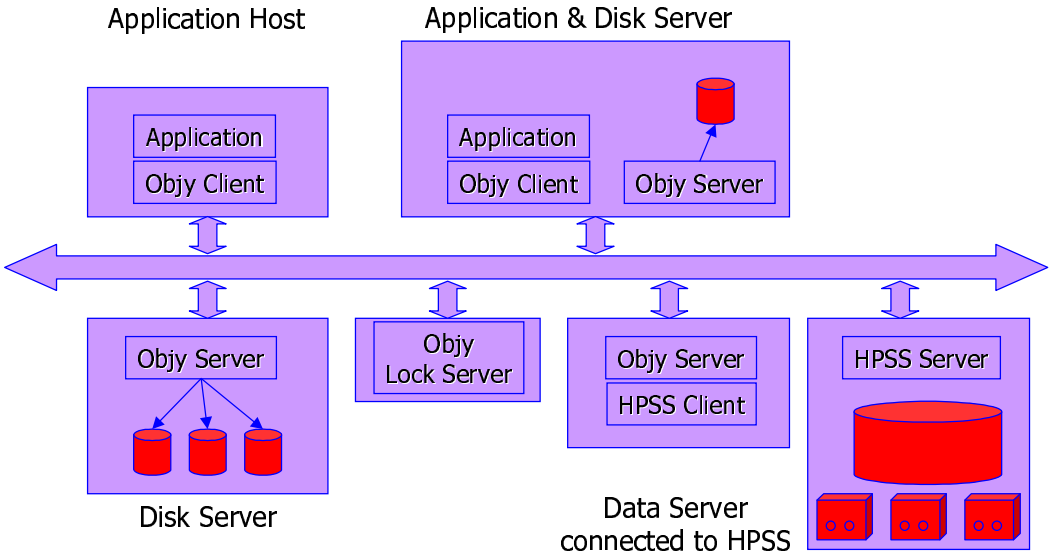

Figure 6: Distributed Applications Sharing a Federated Database

3.2 Data Replication

Objectivity/DB allows replicating all objects in a particular database to multiple physical locations. The aim of this data replication is twofold:

- Enhance performance:
  Client programs may access a local copy of the data instead of transferring data over a network.
- Enhance availability:
  Clients on sites which are temporarily disconnected from the full data store may continue to work on the subset of data for which local replicas are available.

The following diagram shows a simple configuration where one database is replicated from site 1 to two other remote sites over a wide area network.

[2] Processes that run on the same host on which the database is located directly access the database files without using the page server.

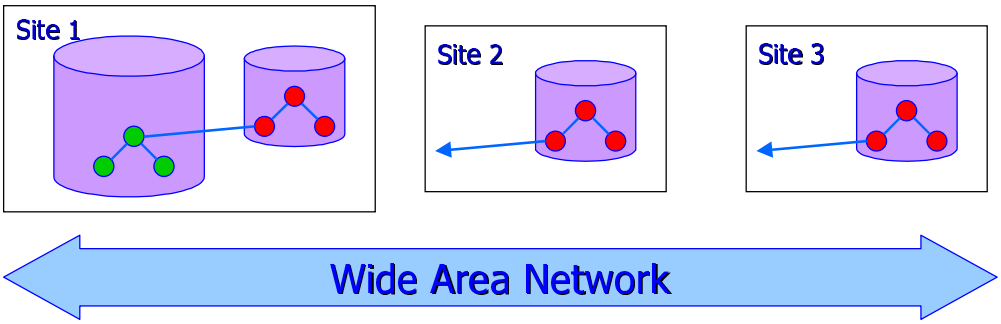

Figure 7: Database Replication

Any state changes of replicated objects on either site are transparently propagated to all other replicas by the database system. In the case that some of the replicas are not reachable, a quorum-based mechanism is used to determine which replica may be modified and a backlog of all changes is kept until other replicas become online again.

The data replication feature is expected to be very useful, for example to distribute central event selection data to multiple regional data centers.

3.3 Schema Evolution

Given the extremely long time scale for LHC experiments it is important to foresee the possibility to change the object model of experiment data during the experiment lifetime. Not only new persistent classes need to be incorporated into the federation schema but also existing class definitions will need to be changed.

The schema evolution feature of Objectivity/DB allows for example to add, move or remove attributes within classes or to change the inheritance hierarchy between persistent classes. If a schema change affects any existing persistent objects the database provides a flexible migration scheme. Depending for example on the amount of involved data one can choose between:

- Immediate Conversion
  all affected objects are converted to the new class layout using an upgrade application
- Lazy Conversion:
  affected objects are upgraded as they are accessed

3.4 Object Versioning

Several ODBMS systems provide support for maintaining multiple versions of the same logical object. In Objectivity these versions may for example constitute a simple linear time series of states of one object or form a more complex tree of states. The following diagram shows a general version genealogy involving multiple linear versions and branches of one object.

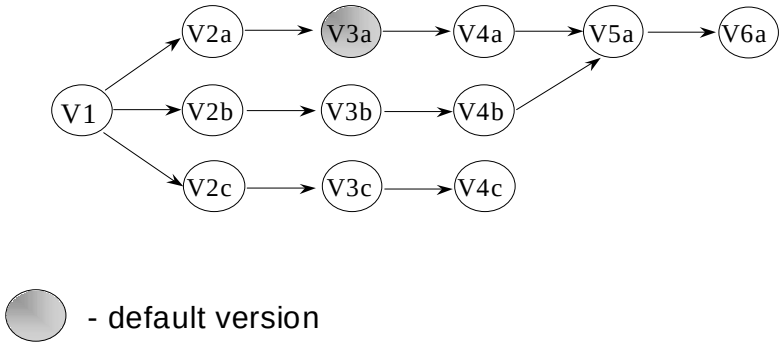

Figure 8: Object Versioning

The object versioning feature is used to implement the calibration database of the BaBar experiment.

3.5 Other ODBMS Products

3.5.1 Versant

Versant uses a different splitting between database client and server than Objectivity/DB. Whereas Objectivity/DB implements a "fat-client/thin-server" model, Versant provides the opposite. The object identifier (OID) in Objectivity/DB has a nearly direct physical mapping, whereas Versant uses a logical object identifier (LOID).

- In Versant, distributed databases are less tightly coupled than in Objectivity/DB. Each database has its own schema, as opposed to Objectivity/DB, where the schema is shared across the entire federation. In our environment, where many databases are likely to reside offline on tape schema inconsistencies between different databases are likely to happen.
- Although Versant implements a LOID, an application must know in which database *each* object was created and is responsible for opening the databases in question. This contrasts strongly with Objectivity/DB where it is sufficient to initialise access to the federation.
- The support for object clustering in Versant is less flexible than in Objectivity/DB.

4 HEP PRODUCTION SCENARIOS: ODBMS-BASED DATA ANALYSIS

A typical analysis scenario can be split in two parts. The first part concerns populating the database with event data and is usually done in a non-interactive C++ program (e.g., in batch mode). The second part implies using an interactive tool, such as the IRIS Explorer framework, to actually produce summary data, usually as histograms, out of the event data.

4.1.1 Building a Tag Database

In this first stage, the analysis data store, assumed to be provided by Objectivity/DB and HPSS, is populated, e.g. from a former reconstruction phase. The following figure shows schematically the difference between the PAW+Ntuple and tagDB models.

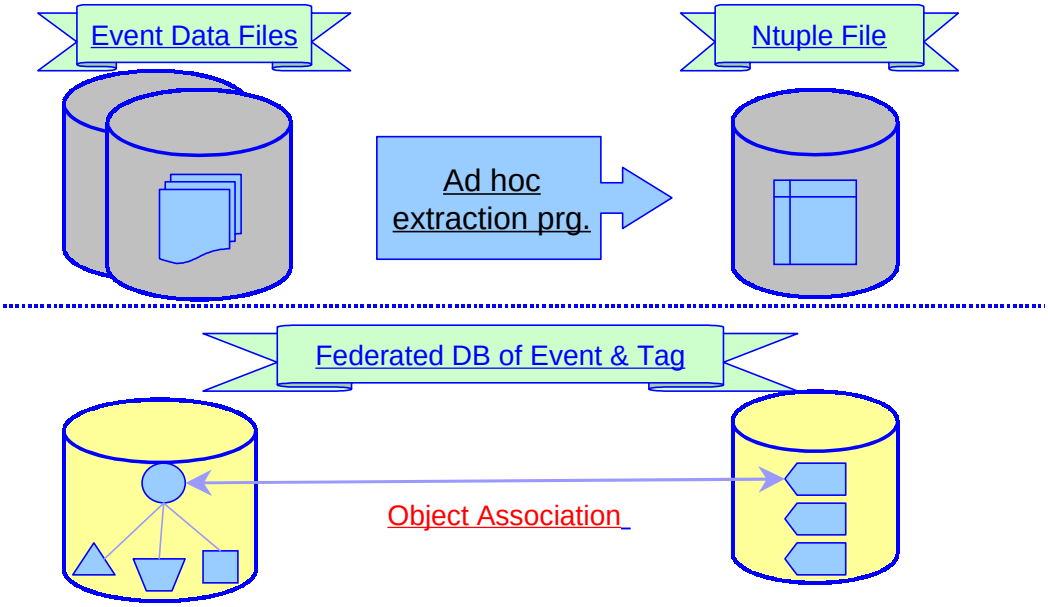

Figure 9 - NTuple vs. TagDB Models

Rather than imposing a pre-defined data model, as is the case with today's NTuple, the LHC++ analysis tools work directly with the data model of the experiment. To assist in the selection of the required events, a new concept is introduced - that of an event *tag*. An event tag is a small object containing selection relevant attributes for each event. The tag objects are stored physically separate from the event to permit efficient clustering, but have an association to the full event data. At the moment, two classes of tags have been implemented:

- *Concrete tags* have their own schema, and are recommended for experiment-wide or work-group activities, as they offer optimal performance.
- *Generic tags*, on the other hand, are more suited for individual physicists - they can be defined on the fly and do not require the definition of new schema, albeit with a small, but acceptable, performance penalty.

Both implementations share the same interface, which is entirely decoupled from the physical storage model. This permits the implementation of different clustering strategies, such as attribute-based clustering - as in column-wise NTuple - without affecting the user interface.

```
// create a new tag collection
GenericTag simTag(“simulation tag”);

// define all attributes of my tags
TagAttribute<long>  evtNo(simTag,"event number");
TagAttribute<float> et   (simTag,"Et particle1");
TagAttribute<float> theta(simTag,”theta particle1");
TagAttribute<short> pid  (simTag,”id particle1");
```

Example 2 - Creation and Definition of a New Event Tag

Tags may be filled in a simple event loop, as shown below. It is important to note that the tag attributes are handled just like normal C++ variables.

```
while ( evt =  geant->nextEvent() )
{
      simTag.newTag();      // create a new tag

       et    = evt->getPart(1).et;
      theta = evt->getPart(1).theta;
      pid   = evt->getPart(1).pdg_code;
}
```

Example 3 - Filling a Previously Defined Tag

As has been described above, a fundamental feature of this strategy is the ease in which the full event data can be accessed. This is a significant piece of new functionality that was not possible using PAW and NTuples.

```
while (atlasTags->next())
{
    if (et > 4.5 &&  sin(theta) > .5) // for selected events…
    { // … fill histograms from the tag…
      cout << "event: " << eventNo << endl;
      etHisto->fill(et);
      thetaHisto->fill(theta);

      // … but also using data from the event.
      Ntracks = atlasTags->event->tracking->trackList.size();
      nTracksHisto->fill(nTracks);
  }
}
```

Example 4 - Accessing the Event Data from the Tag

Having populated a collection of tags - typically, but not necessarily, performed in batch, these data can then be directly be visualised e.g. using IRIS Explorer.

4.1.2 Interactive Data Analysis

Having defined and populated collections of tags, these can then be analysed in the IRIS Explorer framework, using a combination of standard and HEP-specific modules. A user does not need to be exposed to the details - a collection of modules can be predefined and presented to the user as a complete application. Functionality similar to that provided in PAW is available, but with a number of significant differences. The most important of these is that analysis is no longer restricted to the subset of data that has been copied into a NTuple. A user has access to all of the data for a given event and is presented with a unified interface to the entire data store. This includes not only the event data, but also associated meta-data, such as calibration information, the definitions of various event selections, and so on. In this way, is it possible to build associations from histograms to the selection criteria that produced them and the data itself.

In the past, the user was restricted to that subset of the data that was copied into a given NTuple. Furthermore, the management of these NTuple was entirely the responsibility of the user (naming, storing etc.). In the ODBMS approach, the user has transparent access to any event or indeed any part of the event that is stored in a consistent manner experiment wide. These features offer considerable improvements in flexibility and reproducibility over previous approaches.

5 HEP PROJECTS BASED ON OBJECTIVITY/DB

5.1.1 BaBar

The BaBar experiment at SLAC is expected to start taking data in 1999. BaBar uses Objectivity to store event, simulation, calibration and analysis data of an expected amount of 200TB/year. The majority of this storage will be managed by HPSS.

5.1.2 ZEUS

ZEUS is a large detector electron-proton collider HERA at DESY. Since ZEUS started taking data already in 1992, the analysis environment at ZEUS is mainly based FORTRAN using the ADAMO system. Since 1996 Objectivity is used for event selection in the analysis phase. About 20GB of "tag data" are used with the plan to extend to a store of 200GB. The result of re-implementing the event selection using Objectivity/DB was a significant gain in performance and flexibility compared to the old system.

5.1.3 AMS

The Alpha Magnetic Spectrometer (AMS) took its first data on a NASA space shuttle flight in 1998 and will be used later on the International Space Station. AMS's main research goal is the search for antimatter and dark matter. During its first data-taking period Objectivity was mainly used to store production data, slow control parameters and NASA auxiliary data.

5.1.4 CERES

The CERES/NA45 experiment is a heavy ion experiment at the CERN SPS studying electron-positron pairs in relativistic nuclear collisions. In 1997, CERES successfully used Objectivity/DB to perform a parallel reconstruction and filtering of their raw data from a multiprocessor farm (32 nodes MEIKO/Quadrics CS2). After recently upgrading their detector with a time projection chamber, CERES expects in 1999 to write 30 TB of raw data during a data-taking period of 1 month.

5.1.5 CHORUS

The CHORUS experiment searching for neutrino oscillations uses Objectivity/DB for an online emulsion scan database. CHORUS plans to deploy this application also at sites outside CERN.

5.1.6 COMPASS

The COMPASS experiment expects to begin full data taking in 2000 with a preliminary run in 1999. Some 300TB of raw data will be acquired per year at rates up to 35MB/second. Analysis data is expected to be stored on disk, requiring some 3-20TB of disk space. Some 50 concurrent users and many passes through the data are expected.

6 CONCLUSION

HEP data stores based on Object Database Management Systems (ODBMS) provide a number of important advantages in comparison with traditional systems. The database approach provides the user with in a coherent logical view of complex HEP object models and allows a tight integration with multiple of today's OO languages such as C++ and JAVA.

The clear separation of logical and physical data model introduced by object databases allows for transparent support of physical clustering and re-clustering of data which is expected to be an important tool to optimise the overall system performance.

The ODBMS implementation of Objectivity/DB in particular show scaling up to multi-PB distributed data stores and provides a seamless integration with Mass Storage Systems (MSS) like HPSS. Already today a significant number of HEP experiments in or close to production adopted an ODBMS based approach.

BIBLIOGRAPHY

RD45 - A Persistent Object Manager for HEP, LCB Status Report, March 1998, CERN/LHCC 98-x

Using an Object Database and Mass Storage System for Physics Production, March 1998, CERN/LHCC 98-x

RD45 Project Execution Plan, 1997-1998, April 1997, CERN/LCB 97-10

RD45 - A Persistent Object Manager for HEP, LCB Status Report, March 1997, CERN/LHCC 97-6

Object Databases and their Impact on Storage-Related Aspects of HEP Computing, the RD45 collaboration

Object Database Features and HEP Data Management, the RD45 collaboration

Using and Object Database and Mass Storage System for Physics Analysis, the RD45 collaboration

RD45 - A Persistent Object Manager for HEP, LCRB Status Report, March 1996, CERN/LHCC 96-15

Object Databases and Mass Storage Systems: The Prognosis, the RD45 collaboration, CERN/LHCC 96-17

Object Data Management. R.G.G. Cattell, Addison Wesley, ISBN 0-201-54748-1

DBMS Needs Assessment for Objects, Barry and Associates (release 3)

The Object-Oriented Database System Manifesto M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. In Proceedings of the First International Conference on Deductive and Object-Oriented Databases, pages 223-40, Kyoto, Japan, December 1989. Also appears in [13].

Object Management Group. The Common Object Request Broker: Architecture and Specification, Revision 1.1, OMG TC Document 91.12.1, 1991.

Object Management Group. Persistent Object Service Specification, Revision 1.0, OMG Document numbers 94-1-1 and 94-10-7.

The Object Database Standard, ODMG-93, Edited by R.G.G.Cattell, ISBN 1-55860-302-6, Morgan Kaufmann.

ATLAS Computing Technical Proposal, CERN/LHCC 96-43

CMS Computing Technical Proposal, CERN/LHCC 96-45