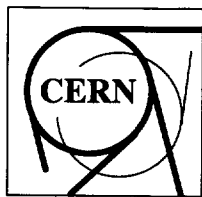


CERN - LHCC - 97 - 7

SCP



EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

CERN/LHCC 97-7
LCB/RD45
February 3, 1997

CERN LIBRARIES, GENEVA



SC00000795

***OBJECT DATABASES AND THEIR IMPACT
ON STORAGE-RELATED ASPECTS OF
HEP COMPUTING***

The RD45 collaboration
CERN, Geneva, Switzerland

We present an analysis of the impact of using an Object Database (ODBMS) for the storage of HEP event data on various aspects of HEP computing, including the Object Model, physical data organisation, coding conventions and the use of third party class libraries. This document has been produced in response to the first milestone set by the LCRB for the second year of the RD45 collaboration, namely:

“Identify and analyse the impact of using an ODBMS for event data on the Object Model, the physical organisation of the data, coding guidelines and the use of third party class libraries.”

TABLE OF CONTENTS

1. Executive Summary	1
2. Introduction	1
3. Impact of the Use of an ODBMS on HEP Applications	1
3.1 Impact on the Object Model.....	2
3.1.1 Object Model for Persistent Data	2
3.1.2 Impact on Physical Data Organisation	3
3.1.3 Object Model Design Using CASE Tools.....	6
3.2 Impact of an ODBMS on C++ Application Code	7
3.2.1 Creation of Persistent Objects	8
3.2.2 Replace References to Persistent Objects by Database References	11
3.2.3 Implementation of a Clustering Strategy.....	11
3.2.4 Implementation of a Locking Strategy.....	11
3.3 Experience from CERES/NA45	11
3.3.1 Object Model Changes	11
3.3.2 Use of Collection Classes.....	12
3.3.3 Clustering Strategy	12
3.3.4 Locking Strategy.....	12
3.4 Experience from CLHEP.....	12
3.4.1 Use of Objectivity/DB with HistOOgrams.....	12
3.5 Experience from GEANT-4	13
4. Impact of Using an ODBMS on 3rd Party Class Libraries.....	14
4.1 Issues Related to Compilers and Compiler/Operating System Levels.....	15
4.2 Use of an ODBMS with the Standard C++ Library	16
4.3 Use of an ODBMS with Rogue Wave Tools.h++	16
4.3.1 Porting Tools.h++ to ObjectStore	18
4.4 Use of an ODBMS with OpenInventor	18
4.5 Use of Objectivity/DB with IRIS Explorer	19
5. Design Hints for the Development of Persistent-Capable Applications	20
5.1 ODMG Object Model.....	20
5.1.1 Persistent Objects Should Inherit from the d_Object base class	20
5.1.1	20
5.1.2 Use of d_Ref<T> Smart Pointer.....	20
5.1.3 ODMG Fixed Length Types.....	21
5.1.4 Avoid Transient Memory References as Part of Persistent Objects.....	22
5.2 Objectivity/DB Implementation	22
5.2.1 Persistent Capable Objects Cannot Contain Other Persistent Capable Objects ..	22
5.2.2 Objectivity/DB References and Transient Objects.....	23
5.2.3 Avoid Creating Persistent and Transient Objects of the Same Class.....	24
5.2.4 Passing of Persistent References and C++ Pointers	24

5.2.5 Initialisation of Persistent References from C++ Pointers.....	24
5.2.6 Default Constructors of Embedded Classes.....	25
5.3 The HepODBMS Portability Layer.....	25
5.3.1 Support for a Consistent Type Naming Scheme.....	25
5.3.2 Support Classes to Implement Clustering and Locking Strategies.....	25
5.3.3 Use of the HepHintDeclare, HepHintInit and HepHintSet Macros.....	25
5.3.4 Use HepNew and HepDelete Instead of the C++ New and Delete Operators....	26
5.3.4.....	26
5.3.5 Encapsulation of Database Session Control.....	27
5.3.5.....	27
5.3.6 Support for Container Libraries.....	27
6. Conclusions	29
7. Glossary	30
8. References	31

1. Executive Summary

In response to the milestones set by the last LCRB review of the RD45 project, we have identified and analysed the impact of using an ODBMS for HEP event data on the object model of the experiment, on physical data organisation, coding guidelines, and the use of 3rd party products, including class libraries and CASE tools.

It is our conclusion that the standards for ODBMSs defined by the Object Database Management Group (ODMG) [20], extend the object model of the language in question (C++) in a very natural way. The impact on both existing and new applications can be minimised by a small layer of software and a small number of coding guidelines.

More details regarding the topics that have been investigated are given below.

2. Introduction

This report has been produced in response to the first milestone set at the March 1996 review of the RD45 project by the LCRB, namely:

Identify and analyse the impact of using an ODBMS for event data on the Object Model, the physical organisation of the data, coding guidelines and the use of third party class libraries.

This document should be read in conjunction with the March 1997 RD45 status report to the LCB [2], together with the supporting documents produced for the work relating to milestones 2 [5] and 3 [6]. This document assumes a working knowledge of object-oriented methods and object data management, as described in [10].

3. Impact of the Use of an ODBMS on HEP Applications

Traditionally, the selection of a storage sub-system has been one of the most important choices during the design of software systems for HEP experiments, as this choice strongly influences the data model of the experiment. All data types and relations between different parts of the data have to be expressed within the model implemented by the chosen system. In addition, traditional solutions enforced a special coding style on a large fraction of the software of the experiment. Contributing collaborators had to be trained to adhere to a common set of coding rules in order to make the software system maintainable as a whole.

Compared with such approaches, object databases offer the advantage that, rather than defining an arbitrary new data model, they extend the object model of standard OO languages such as C++, Smalltalk or Java with respect to object persistency and do not require large object model or code changes to transient applications that have to be ported.

In the next two sections we will report on the impact of the use of an object database on the object model in general and on C++ applications in particular. All information presented here is based on the experience gained in several prototype projects, in which non-trivial C++ applications from most major HEP problem domains like simulation, reconstruction, filtering and analysis, have been ported to use an ODBMS.

3.1 Impact on the Object Model

In order to estimate the impact of the use of an ODBMS on the Object Model, we have considered the following points:

- the ODMG standard requires that persistent-capable classes are defined using a special definition language, the *Object Definition Language*, or ODL. Non-persistent applications are typically created by defining C++ classes using header files, often generated automatically using a CASE tool. We evaluate how the use of ODL impacts the development of both persistent and transient applications,
- an application that is designed without persistence in mind might result in an object model that leads to unacceptable performance when combined with an ODBMS. To investigate the impact of an ODBMS on the Object Model, we consider aspects such as object granularity and the use of C++ pointers versus ODMG associations, based on the experience with both NA45 and GEANT-4,
- closely linked to the above is the use of CASE tools. We evaluate the possibilities for generating ODL files directly from two CASE tools - Rational ROSE and Classify/DB.

3.1.1 Object Model for Persistent Data

The starting point for the conversion of a transient application to use the database is to identify all persistent classes and make them inherit from the persistent capable base class *d_Object*.

3.1.1.1 ODMG Container Classes

The ODMG ODL provides some significant extensions to the standard C++ object model. For example, it includes a variety of container classes, including sets, variable length arrays and so forth. These classes can contain persistent objects and may also be embedded in other objects.

3.1.1.2 ODMG Object Associations

Relations between persistent objects - from simple uni-directional 1:1 object references (like pointers in C++) to 1:n and bi-directional associations - are provided by the database. In the case of bi-directional associations, referential integrity is guaranteed by the database,

so that common coding errors in the C++ object model like invalid references to objects which have been moved or deleted are avoided.

3.1.1.3 Typical Problems During the Model Conversion

3.1.1.3.1 Model Defines a Logical Entity by the Lifetime of its Components

During the design of transient applications, a consistent model for the complete class hierarchy is developed. However, when the application is ported to a database this model is typically split into transient and persistent parts. Since only the persistent part is available to another process, the model has to be revised to make sure that all use cases can be implemented from the persistent model.

As an example, we cite the case of an event. In a purely transient application, an event is typically defined as all reconstructed objects which exist at a given time. Before the next event is processed, be it simulated, reconstructed or analysed, all objects corresponding to the previous event are deleted and new instances are created. In the case of persistent objects, this approach must be changed. Deleting persistent objects from within an application also results in their deletion from the database, and so should be avoided. In addition, mechanisms for retrieving persistent objects need to be provided, which is typically performed by creating additional persistent classes, such as event, run, burst etc.. These new classes contain associations to all their persistent components and thus permit, for example, iteration over all of the events belonging to a given run.

3.1.1.3.2 Classes Containing Components of Different Lifetime

Another problem that must be faced when making transient applications persistent, is that of object lifetime. One such example is the definition of a detector object, which *contains* its calibration and event data. A transient application might, for example, reset the event data for each new event but only reset the calibration part once per run. Simply making the detector object persistent would result in a very inefficient implementation. Since the lifetime of the event data determines the lifetime of the detector, one would have to create a new instance of the detector for each event. This would mean that, not only would the event data be written to the database per event, but also the calibration constants, even though, in this scenario, they are constant over a much longer period, i.e. the entire run.

This problem can easily be solved by splitting the detector object into multiple parts, each a with well defined lifetime. Multiple detector event data objects could, for example, reference the same detector calibration constant object, thereby avoiding the problem described above.

3.1.2 Impact on Physical Data Organisation

To understand how the use of an ODBMS affects the physical organisation of data, we have considered the following topics:

- Performance,
- Replication and other distributed techniques,
- Impact of the above on the Object Model.

3.1.2.1 Performance Issues

The fundamental key to improving performance is to minimise I/O. The only way that I/O can be minimised is to ensure that each read or write that is performed transfers the maximum amount of useful data.

In other words:

- the data overhead introduced by ODBMS should be minimised,
- the objects must be physically *clustered* so that the amount of unwanted data that is transferred is minimised.

Obviously, there is no way of clustering data that satisfies all conceivable access patterns - it is always possible to invent a pathological case that defeats any attempt at optimisation. However, by understanding the dominating access patterns, it is possible to greatly improve both read and write performance. In HEP, data is typically written once, and then read (very) many times. Hence, if a trade-off is required between read and write performance, read performance should probably be given priority.

I/O is always performed in units that are much larger than the size of an atomic data item - it is not possible to read or write a single byte or word from a storage device. A transfer will typically involve a page of 512 bytes or more, depending on the characteristics of the filesystem and/or device(s) involved. In addition, the network overhead is often such that a large block size is most efficient.

In the current version of Objectivity/DB, replication is performed at the level of a physical database, or file, although different policies are possible for each physical database. This obviously has a strong impact on the physical organisation of data - the database structure must clearly match the replication policy. Furthermore, data that do not need to be replicated should clearly not be stored in databases that are replicated.

The question of performance is investigated further in [6], whilst replication is covered in [5]. We summarise the main conclusions of these reports below, and discuss the implications for physical data organisation.

3.1.2.2 Persistent Object Granularity

Making an object persistent by storing it in an ODBMS typically involves a small space overhead. This varies from product to product, and is a minimum of 14 bytes with Objectivity/DB. For objects with associations to other objects, the overhead increases accordingly. During the design of a persistent object model, one therefore needs to consider different containment scenarios. For example, should the instances of a particular class exist as separate objects and used by reference from other classes or should they be directly embedded in these other classes?

Although such questions are also relevant in the context of transient-only applications, there are additional issues involved in the case of persistent applications.

The alternatives - persistence by containment or not - have both advantages and drawbacks:

- Instances of classes which are persistent by containment do not have an OID and can therefore not be used as a target in associations, hash tables or indexes, nor can they be named.
- On the other hand, individual persistent objects - each with their own OID - have a storage overhead of 14 bytes. This overhead can lead to a significant storage inefficiency especially for large numbers of relatively small objects, e.g. those less than about 10 words in size.

In other words one may decide to use individual, rather than contained, objects if one or more of the following are needed:

- indexing,
- naming,
- associations,
- maps,
- versioning,
- general flexibility.

Finding the most efficient implementation with respect to storage efficiency and retrieval performance needs careful consideration and the knowledge of typical access patterns of the software which uses these objects.

Most databases implement variable length arrays (Varrays) as a storage entity separate from the containing object. If any of the array elements is accessed¹, the whole VArray storage is mapped automatically into the client process as one continuous block. Although this is useful if all or most elements will be accessed, it is not always the optimal solution. An example would be a variable size array of 10000 calorimeter clusters which are ordered by energy. In many use cases, the analysis will only access those clusters which carry large energies and thus mapping the entire VArray to memory will result in a large I/O overhead.

```
class Calorimeter;
class CaloCluster : public d_Object {
public:
    d_Float depositedEnergy;           // a single float of data
    d_Ref<Calorimeter> itsCalorimeter; // associated with the calorimeter
};

class Calorimeter : public d_Object {
d_VArray<d_Ref<CaloCluster> > itsClusters;
};
```

Separate object implementation

¹ Complete mapping in one step is the simplest way to retain the semantics of calculations with pointers to vector elements and permits the assignment of e.g. multiple vector elements through bit-wise copy operations (e.g. *memcpy()*).

```
class CaloCluster { // transient
public:
  d_Float depositedEnergy; // just one float of data
};

class Calorimeter : public d_Object { // persistent
  d_VArray<CaloCluster> itsClusters;
};
```

Contained object implementation

3.1.3 Object Model Design Using CASE Tools

3.1.3.1 Tools Based on Objectivity/DB - Classify

Classify/DB is a set of products developed by Micram Object Technology, the distributor of Objectivity products in Germany. They are designed to work exclusively with Objectivity/DB - not only does the product generate Objectivity DDL (as well as ODMG² ODL for the C++-binding), but an Objectivity/DB ODBMS is used to store and manage the object model. The Classify CASE tool, Classify/Views, provides an OMT editor and diagram generator.

Classify/DB offers a CASE-environment that facilitates the creation of object models and the development of object-oriented applications for these models. A central (Objectivity) database, the repository, is used to store all information describing the static and dynamic aspects of the various object models. Thus, Classify/DB integrates the object modelling phase with the implementation of the model.

Moreover, Classify/DB is capable of automating and managing the entire software development life cycle and eliminates the laborious and error-prone task of writing code by hand.

The Classify products can handle a variety of input/output formats, including:

- C++ header files,
- ODMG ODL files,
- Objectivity/DB DDL files,
- Metaphase MODEl,
- Step EXPRESS.
- Java.

The Classify tools were used to help design the ALICE raw data model, and also for an ATLAS DAQ project. These exercises demonstrated that ODL can indeed be generated from a CASE tool. However, due to problems with the early release of the tool, its use was abandoned. A newer release of the product showed significant improvements, but it is our

² In fact, the Classify products were the first of any to support an ODMG binding.

feeling that the CASE tool of choice should be able to generate ODL directly, rather than a specific tool being imposed due to the requirement of generating ODL.

3.1.3.2 General Design Tools ROSE and StP

In order to produce ODL from a commercial design tool, such as ROSE or StP, one typically needs to customise the C++ code generator, adding a post-processing script to handle associations. Early investigations using ROSE, indicate that the main problems are:

- obtaining the role names of bi-directional associations,
- change the association implementation from a C++ pointer to ODMG-compliant base types (ODL),
- dealing with additional properties on associations, such as *delete* and *lock propagation*.

We are aware of a number of customisations of ROSE to produce both ODL and Objectivity/DB DDL. However, we were unable to obtain these customisations under favourable terms.

3.1.3.3 Conclusions Regarding CASE Tools

Although today's CASE tools do not normally generate ODMG ODL directly, they are typically sufficiently tailorable so that rudimentary ODL generation can be added with relatively little effort. In the short term, such export filters should be made available for the commonly used tools, namely StP (ATLAS) and ROSE (CMS, GEANT-4). In the longer term, as the market penetration of ODBMSs increases, we believe that this capability will be added to these products, and the need for a specialised tool, such as Classify/DB, will disappear.

In any case, it is our conclusion that a CASE tool designed specifically to work with a given ODBMS product is of limited appeal. A highly preferable solution is that described above, namely that the CASE tool of choice be capable of generating ODMG ODL files, and of performing reverse engineering from such files.

3.2 Impact of an ODBMS on C++ Application Code

In ODMG compliant databases object persistency is introduced through a very natural extension of the standard object allocation and object pointer semantics.

- The database overrides the standard *new* operator in C++ and thereby intercepts the normal object creation. Instead of allocating the storage for a new object on the heap, the object is registered with the database and allocated in a client-side database cache. The cache is written to disk at the end of the current database transaction.
- References to existing persistent objects are maintained via smart pointer types (e.g. `d_Ref<T>` or `ooHandle(T)`) which replace C++ object references through raw pointers. The smart pointers transparently intercept any program access to a persistent object and allow the database to bring in the object from the disk storage as needed by the application.

Since this approach does not involve any explicit I/O statements and no explicit copying between I/O buffers and program variables, porting of existing code to a database typically involves many fewer code changes than a port to a traditional I/O system, such as ZEBRA.

To understand the impact of a database on existing C++ code, several non-trivial HEP applications and class libraries have been ported to use object persistency provided by an ODBMS. The ported code covers most HEP problem domains, including event simulation, reconstruction, filtering and analysis. It should be noted that none of these applications was originally designed with the use of an ODBMS in mind.

We present below some considerations on the common tasks that have to be performed when adding persistence, including:

- implementation of a creation and deletion strategy for persistent objects,
- definition of a coherent logical object model of the persistent part of the data,
- definition of a physical object model: clustering and locking.

We also describe the results of the different porting efforts.

3.2.1 Creation of Persistent Objects

As described above, clustering of objects is of crucial importance for both write and retrieval performance. In an ODMG-compliant system, the *new* operator is overloaded to allow a clustering hint to be specified at object-creation time, via an additional argument. This argument specifies another object in the database close to which the new object should be placed. The exact meaning of "close to" is implementation dependent. Typical implementations place the new object on the same page as the hint object, or in an adjacent page if there is insufficient space on the page in question. With Objectivity/DB, this clustering hint also defines the container and database within the federation where the new object will reside.

The clustering hint argument gives the application a great deal of flexibility in defining even complex clustering strategies to optimise the database performance, based on typical access patterns.

One drawback of this implementation, especially with respect to existing C++ code, is that all occurrences of the *new* operator have to be changed consistently to use an appropriate hint argument.

```
// old code: create a new track on the program
Track *tr = new Track(phi,theta,p_t);

// new code: create a persistent track in
// the container that the variable "clustering" points to
d_Ref_Any clustering = calculatePlacement();
d_Ref<Track> tr = new(clustering) Track(phi,theta,p_t);
```

Comparison of transient and persistent object creation

In many cases, the clustering strategy used for all objects of a particular class is the same, e.g. all persistent Tracks go into one well defined container and database:

```
// new code: create a new persistent track in the container "trackCont"  
tr = new(trackCont) Track(phi,theta,p_t);
```

Creation of a persistent object

In this case of class-based clustering, we suggest the definition of a static member (class variable) named "clustering" in each persistent class. This member, which should be derived from the base class HepClusteringHint, is used to encapsulate the calculations needed to determine the object placement. The prototype HepODBMS package, which will eventually be distributed as a component of LHC++, provides several concrete hint subclasses, e.g.

- **HepContainerHint**

This class implements the simple strategy to put all objects into a single Objectivity/DB container.

- **HepContainerGroupHint**

This class generalises the above strategy for large amounts of data: If the total amount of data is larger than the maximum number of pages per container, new containers will be created automatically as needed.

- **HepParallelWriterHint**

This class implements a lock-free clustering strategy for multiple writer configurations. Objectivity containers are not only a physically clustered group of objects but are also the locking granularity of the system. Access to individual containers is therefore organised in such a way that each process is guaranteed to write to a separate group of containers.

These classes could perhaps be implemented as a single class with different initialisation, which would permit, for example, automatic switching to multiple container and/or parallel writing mode as required.

```
#include "HepODBMS.h"

class Track : public HepPersObject {
protected:
    d_Double phi;
    d_Double theta;
    d_Double p_t;
    d_Double chiSqr;
public:
    Track();

    // clustering hint class variable
    static HepClusteringHint clustering;
};

class MyTrack : public Track{
public:
    refit();    // my new refitting method

    // clustering hint class variable
    static HepClusteringHint clustering;
};
```

Object definition

```
// during the application initialisation
MyApp::ClusteringInit()
{
    // call the application base class to establish the clustering
    // strategy for HepXXXX base classes
    HepDbApplication::ClusteringInit();

    // select a logical db for reconstructed data
    db("ReconstructedData");

    // set track clustering to the container "Tracks"
    Track::clustering = container("Tracks");

    db("PrivateDB");
    // put my tracks into the container "MyTracks"
    MyTrack::clustering = container("MyTracks");
}

MyApp::processTracks()
{
    tr = new(Track::clustering) Track(...);
}

MyApp::refitTracks()
{
    // some processing ...

    // creat new hit list object
    hits = new(FadchHits::clustering) FadchHits;

    // ... more processing
}
```

Example use of ClusteringHint objects

3.2.2 Replace References to Persistent Objects by Database References

- references to persistent objects like pointers and C++ references (operator &) with smart pointer *d_Ref<T>* supplied by the database,
- existing collections must be replaced by the equivalent ODBMS collection.

The first item is typically the most intrusive change that has to be made. However, although the type declarations of the pointers that reference persistent objects need to be changed, the code that uses these variables stays untouched. In addition, there are other reasons why the use of smart pointers should be considered, including:

- to implement automatic garbage collection,
- to allow checks to be made on pointer validity, particularly during the development phase.

3.2.3 Implementation of a Clustering Strategy

In an ODBMS, the physical organisation of the data is independent of the logical object model. However, the physical organisation has a direct impact on performance, as I/O can be minimised by storing objects that will be accessed together on the same, or adjacent, database pages. Clustering can be performed at many levels, including per class or even per object. Further investigations of clustering strategies are discussed in the report on milestone 3 [6].

3.2.4 Implementation of a Locking Strategy

Although HEP data is read-mostly, there are areas where locking is required, including the reconstruction farm. To minimise conflicts, an optimal locking strategy needs to be defined. Different ODBMS products implement locking in a variety of ways, including object level, page level and container level locking. Objectivity/DB implements container level locking, which allows lock traffic to be minimised. In the case of parallel database update, e.g. from multiple nodes in a reconstruction farm, a lock-free strategy can be used, as described below for NA45. Objectivity/DB also implements an access mode which permits multiple, concurrent readers and a single writer (MROW). This implementation also guarantees that users see consistent data during a transaction.

3.3 Experience from CERES/NA45

The NA45 (CERES) experiment redesigned and reimplemented their reconstruction program in C++ in early 1996. The initial design and implementation was made without any consideration of persistence and a number of changes had to be made to accomplish the latter. The experience gained from this exercise has helped establish the list of guidelines described in section 3.1 on page 2.

3.3.1 Object Model Changes

The CERES data model was made persistent in two steps. Firstly, only the raw data classes (labels and raw data event) and the setup information (start of run, end of run, start of burst,

end of burst, etc.) classes were made persistent. These changes were made without encountering any major problems. The main change was the introduction of new persistent classes CEvent, CRun and Cburst, which were used in the persistent part of the model to implement a burst object with a 1:n relation to all associated events.

3.3.2 Use of Collection Classes

The transient version of the NA45 reconstruction program made heavy use of a number of collection classes from Rogue Wave's Tools.h++ class library. Although a version of this library exists for Objectivity/DB, it is unfortunately unavailable on HP/UX systems, due to compiler limitations, HP/UX being the primary development platform used by NA45.

This problem was circumvented by migrating to functionally equivalent classes based on the Objectivity/DB ooVArray persistent collection class.

3.3.3 Clustering Strategy

The NA45 data was written into separate streams, using a special clustering hint class. The streams were as follows:

- Raw database: for raw data "labels", raw data event objects,
- Event database: for run, burst and event objects,
- Reconstructed database: for RICH ring-lists, vertex and track lists,
- Start-of-run database: for setup data from start of run/burst.

3.3.4 Locking Strategy

To avoid locking conflicts, a lock-free strategy was used. Each node used in the production process used a separate container in the same federation, thus avoiding locks. These containers may be accessed as a single logical container, using special application classes.

The locking strategy was implemented via the clustering hint class, as it is the clustering directive which determines where individual objects are placed.

3.4 Experience from CLHEP

3.4.1 Use of Objectivity/DB with HistOOgrams

In the context of the LHC++ working group, a set of persistent-capable histogram classes are being developed. To make these classes persistent, the main changes were:

- the use of ODMG-defined types (*d_Double*) etc. The ODMG-defined types have a fixed implementation regardless of platform to overcome compiler and machine architecture differences,
- the use of smart pointers to implement associations.

Thus, by following the guidelines described earlier in this report, and by using the header files and classes (e.g. the HepRef smart pointer) that will become part of LHC++, it was relatively straightforward to implement both transient and persistent-capable histogram classes.

3.5 Experience from GEANT-4

Persistence for calorimeter and tracker "hits" objects has been introduced in GEANT-4 using Objectivity/DB. Two different implementations have been tested:

- storing each hit object as a separate persistent object in the database,
- storing the hits in a persistent VArray.

In both cases, as can be seen from the tables below, the overhead introduced by making the objects persistent is very small. In the case of the calorimeter hits, two collections are created, of 19 and 17 objects respectively. The objects are accessed 100 times, as the energy deposition is accumulated. In the case of the tracker hits, collections of 1900 and 1700 objects are created. However, each object is accessed only once (at construction time).

	Calorimeter Hits		Tracker Hits	
	Transient	Persistent	Transient	Persistent
User time	7.96	9.63	8.80	13.09
Real time	12.2	14.22	9.63	26.33

Individual Persistent Objects

User time	8.66	8.37	9.66	8.89
Real time	10.96	15.87	11.28	14.41

Persistence by Containment in a VArray

These numbers include only minor optimisations - for both persistent and transient Varrays, the arrays are extended in blocks of 1024, rather than one entry at a time. In both cases, this results in a performance improvement of approximately a factor of four. One might also expect improvements in the case of individual objects, if an enhancement was made to the database to create multiple objects in one go.

The small overhead introduced by the database is striking, and can be compared with that incurred by storing ZEBRA objects in an RZ file. In the case of a very simple test, e.g. using a linear chain of 1900 banks, each containing 10 data words, the I/O overhead represents a small factor, rather than a fractional increase, as is the case using an ODBMS.

4. Impact of Using an ODBMS on 3rd Party Class Libraries

Certain foundation classes are unfortunately highly intrusive. As a result, it is non-trivial to change from one implementation to another and consequently the choice of appropriate foundation class libraries is extremely important.

The types of class library involved include those that provide:

- a “universal base class” of all objects,
- object references and pointers,
- object containers and collections,
- container and collection iterators.

For example, in the case of container libraries, the interface is typically very different, as may be the object model relation between containers, collections and iterators.

In an environment where multiple large class libraries have to collaborate in order to provide a consistent set of foundation classes, any component that forces the user to use yet another non-standard container system or base class would be major problem. Only a properly layered system based on standardised foundation classes will allow a consistent integration and keep the system open for new packages

In order to understand the implications of using an ODBMS together with class libraries, we have tested the use of Objectivity/DB with most of the components of LHC++, namely:

- The Standard C++ library
- RogueWave Tools.h++
- CLHEP
- OpenInventor
- IRIS Explorer
- GEANT-4

In all cases, the issues involved can primarily be divided into two classes:

1. Those related to requirements for specific compilers, compiler versions, and/or operating system levels,
2. Those that are concerned with persistence - i.e. producing persistent-capable versions of classes defined in the class libraries, and not using an existing (transient) class library and an ODBMS in the same application.

4.1 Issues Related to Compilers and Compiler/Operating System Levels

As is common in the field of software, and has long been true for the current, FORTRAN-based, CERN Program Library, software packages or libraries often require, or are only validated on, certain compiler/operating-system combinations. Object databases need an intimate knowledge of how different compilers lay out objects in memory - and, in the case of certain ODBMS products, also require detailed operating system knowledge. Irrespective of these concerns, one of the principle roles of a database is to guarantee integrity, and this implies that exhaustive testing of the database code has been performed on a system ideally identical to that employed for the deployment of the application in question.

Objectivity/DB is currently supported on all of the "reference platforms" in use at CERN (namely AIX, Digital-Unix, HP/UX, SGI Irix, Sun Solaris and NT). Although more than one compiler version is supported on one platform, and in some cases multiple operating system/compiler levels are either supported or can be shown to work, it is clearly advisable to adhere as closely as possible to the officially supported platforms.

Given that the proposed LHC++ environment (the approximate equivalent of the current CERNLIB) is based upon several commercial products (all *de-facto* or *de-jure* standards), the possibility of conflicting requirements arises.

As examples of possible conflicts, we cite the following cases:

- Objectivity/DB version 4.0 supports a persistent-version of Rogue Wave's Tools.h++ class library, (based on Tools.h++ 6.1) whereas the current version of the standard (transient) version of this product from Rogue Wave is 7.0,
- OpenInventor (from Silicon Graphics) does not support g++, nor are there any plans for such support to be provided in the future,
- Objectivity/DB does not yet support a persistent version of Tools.h++ on HP/UX or Alpha NT, the former as Tools.h++ requires full template support, which is not yet available in the current C++ compiler from HP, the latter simply due to manpower constraints.

Eventually, one may hope that such problems will disappear - as the appropriate technologies become mature. In the interim, however, we have taken the pragmatic approach of strongly encouraging the vendors involved to establish links, and produce compatible products. Although strictly an LHC++ issue, we report here the following successes:

- Objectivity/DB have committed to improve their relationship with Rogue Wave, and synchronous releases of their respective versions of Tools.h++,
- NAG (the supplier to CERN of IRIS Explorer, Open Inventor, Open GL), has established links with both Objectivity and Rogue Wave. For example, NAG is now a distributor of Rogue Wave products, and the two companies are combining forces on future C++ based mathematical libraries,

- Finally, through regular workshops at CERN, and via close contacts with the companies in question, we intend to push for synchronised (or at least compatible) releases.

In summary, we propose to minimise this unavoidable problem by encouraging strong relationships between the major suppliers of the LHC++ environment - a strategy that offers considerable advantages not only to the HEP community but also to the suppliers involved - and hence offers a reasonable chance of success.

4.2 Use of an ODBMS with the Standard C++ Library

The current version of the ODMG standard, V1.2, provides a number of templated collection classes, namely:

- `d_Set`
- `d_Bag`
- `d_List`
- `d_Varray`

For these collection classes, a templated iterator, `d_Iterator`, is defined. It is the stated intent of the ODMG to comply fully with the C++ Standard Template Library (STL) in future releases of the ODMG standard - V2.0, scheduled for late 1997, will offer much more complete support than is available in V1.2. Today, `d_Iterator<T>` objects conform to the STL specification of constant iterators of the *bidirectional_iterator* category.

In practice, this is still an area which is developing rapidly - the C++ standardisation process is not yet complete, there are still platforms for which compilers with adequate template support is not yet available, the ODMG standard is still evolving, and conforming products are waiting for the evolution of these standards before committing to a specific implementation. As a pragmatic interim solution, the use of Rogue Wave (persistent-capable) collection and container classes is recommended.

4.3 Use of an ODBMS with Rogue Wave Tools.h++

Rogue Wave Tools.h++ has been de-facto standard foundation class library for C++ programmers for many years. Recently, Rogue Wave have introduced a new version, V7, restructured to make full use of the draft standard C++ library, and the Standard Template Library (STL) component in particular.

Tools.h++ provides many "building-block" classes, including string, date and time, I/O and collection classes, the latter being of particular interest. It is not possible to use the standard Tools.h++ class library with an ODBMS (if one wishes to have persistent-capable instances of the Rogue Wave classes). However, versions of Tools.h++ exist for a number of ODBMS products, including Objectivity/DB.

At the time of writing, the current version of Tools.h++ from Rogue Wave is 7.0, whereas that supplied by Objectivity is 6.1. This is currently a significant inconvenience, but we have been assured by Objectivity that they will address this problem in the coming months.

The version of Tools.h++ for Objectivity/DB implements a subset of the standard product, which allows persistent versions of the Tools.h++ classes to be used to create objects stored in an Objectivity/DB federated database.

Objectivity/Tools.h++ offers the following persistent classes:

Class Category	Rogue Wave Tools.h++	Objectivity/Tools.h++
List, doubly-linked	RWTPtrDlist<T>	RWODTRefDlist<T>
Hash dictionary	RWTPtrHashDictionary<K,V>	RWODTRefHashDictionary<K,V>
Hash set	RWTPtrHashSet<T>	RWODTRefHashSet<T>
Hash table	RWTPtrHashTable<T>	RWODTRefHastTable<T>
String	RWCString	RWODCString
Vector, ordered	RWTPtrOrderedVector<T>	RWODTRefOrderedVector<T>
Vector, plain	RWTPtrVector<T>	RWODTRefVector<T>
Vector, sorted	RWTPtrSortedVector<T>	RWODTRefSortedVector<T>

In addition, the following embedded classes are defined. These classes may *only* be used as embedded member data in persistent classes, as with d_Varray.

Class Category	Rogue Wave Tools.h++	Objectivity/Tools.h++
Bit vector	RWBitVec	RWEmbBitVec
String	RWCString	RWEmbCString
Vector, ordered	RWTPtrOrderedVector<T>	RWEmbOrderedVector<T>
Vector, plain	RWTPtrVector<T>	RWEmbVector<T>
Vector, sorted	RWTPtrSortedVector<T>	RWEmbSortedVector<T>

Finally, there are also transient classes for containing persistent objects and performing operations such as sorting, ordering and iterating.

Class Category	Rogue Wave Tools.h++	Objectivity/Tools.h++
List iterator, doubly linked	RWTPtrDlistIterator<T>	RWODTRefDlistIterator<T>
Hash dictionary iterator	RWTPtrHashDictionary Iterator<K,V>	RWODTRefHashDictionary Iterator<K,V>
Hash table iterator	RWTPtrHashTable Iterator<T>	RWODTRefHashTableIterator<T>
Vector, ordered	RWTPtrOrderedVector<T>	RWTRefOrderedVector<T>
Vector, plain	RWTPtrVector<T>	RWTRefVector<T>
Vector, sorted	RWTPtrSortedVector<T>	RWTRefSortedVector<T>

In addition to these naming differences, which are clearly an impediment to converting transient applications to persistent ones, there are a number of functional differences between the two versions of Tools.h++, that are too detailed to list here. Suffice it say that it is not possible to migrate a transient application that makes heavy use of Tools.h++ to be persistent in a completely transparent manner.

Version 2.0 of the ODMG standard will include a subset of the STL containers. As a long-term strategy, it is clear that STL-based container libraries should be used, and standardisation in this area will help to minimise the changes that are required.

4.3.1 Porting Tools.h++ to ObjectStore

A discussion of some of the important issues related to porting container libraries to an ODBMS can be found in a Rogue Wave paper, available via

- http://www.roguewave.com/products/tools_os/ostorept.html.

4.4 Use of an ODBMS with OpenInventor

Open Inventor, from Silicon Graphics, is an object-oriented toolkit for developing interactive, 3D graphics applications. It also defines a standard file format for exchanging 3D data among applications. Open Inventor also serves as the basis for the Virtual Reality Modeling Language (VRML) standard.

Open Inventor applications, such as IRIS Explorer, may be used transparently with an ODBMS. To store persistent Open Inventor classes in an ODBMS, rather than in Open Inventor files, requires subclassing of the Open Inventor classes, inheriting from `d_Object`, as usual.

Open Inventor scene graphs can be stored in machine-independent (SGI-format, i.e. big-endian, Unix line-end for ASCII files) Open Inventor binary or ASCII files.

The possibility of storing Open Inventor scene graphs is clearly an interesting one, although producing and maintaining a fully persistent-capable Open Inventor toolkit is clearly a daunting task. Nevertheless, we have made some preliminary studies to understand the implications of such a toolkit and have passed the suggestion back to Silicon Graphics.

4.5 Use of Objectivity/DB with IRIS Explorer

IRIS Explorer is a toolkit for data visualisation. It is a modular system - users can select from a wide range of modules supplied with the package, public domain modules written at various user sites, or generate their own modules. These modules are then linked together graphically, to provide the required functionality.

IRIS Explorer comes with a number of modules to read industry-standard file formats, and a simple module to read formatted or unformatted files according to a user-provided description. Data read from the appropriate source is then passed to another module, such as a histogramming module.

At the time of writing, IRIS Explorer does not come with a module to read from an ODBMS, and so a small module was built to access an Objectivity/DB database, browse the objects in that database, select objects and then pass the selected data onto further modules.

In the current prototype, objects are exchanged between different modules by passing the 64-bit Objectivity OID (in fact, shared memory is used for passing the OID, unless the modules are run on different machines, in which case TCP sockets are used), and the database provides concurrency control between the different modules.

In the current version of IRIS Explorer, each module corresponds to a separate process. As Objectivity/DB currently uses a separate cache for each client, this implies a small overhead when passing objects between different modules. A future version of IRIS Explorer may use threads rather than processes for the different modules, which would circumvent this problem. In the meantime, we are discussing with Objectivity the possibility of a shared client cache, as is implemented in some other ODBMSs, such as Versant.

5. Design Hints for the Development of Persistent-Capable Applications

We present below a list of guidelines and recommendations for writing persistent-capable applications, based on the experience in NA45, GEANT-4, BaBar and other projects. These recommendations are presented as a series of guidelines and examples, where appropriate.

The guidelines have been grouped into three main categories.

- Consequences of the ODMG object model,
- Consequences of the incomplete ODMG implementation by the current Objectivity/DB release,
- Guidelines to simplify switching between different releases of Objectivity/DB.

5.1 ODMG Object Model

5.1.1 Persistent Objects Should Inherit from the `d_Object` base class

As described in section 3.2 on page 7, this base class provides the functionality needed to make an object persistent-capable.

5.1.2 Use of `d_Ref<T>` Smart Pointer

References to persistent objects are implemented using the `d_Ref<T>` template.

- `d_Ref` objects can be used as transient variables or as embedded part of persistent objects (associations).
- A shortcut `d_Ref_Any` is provided for references to any persistent object.
- The shortcuts `HepDatabase` and `HepContainer` are provided for references to data bases and containers.


```
// example:

class Track : public d_Object {
public:
    d_Double phi;
    d_Double theta;
    HepRef(Hit) hit[]; // use of HepRef as association
// more definitions
    ....
};

void foo()
{
// use a transient d_Ref to point to a new persistent Track
    d_Ref(Track) tr = new Track;
// access simple attributes
    tr->phi = 0;
    tr->theta = 0;
// associate a new persistent hit with the track
    tr->add_hit(new Hit);
}
```

- HepRefs will support to switch between Objectivity ooRef/ooHandle types to the ODMG compliant d_Ref templates. For the current version of objectivity they will be based on the ooHandle(T) macro. Starting from version 4 of objectivity we will use the d_Ref<T> template.
- HepRefs will support to switch to a “transient only” mode for developing code on “persistent” objects without having access/ having to link to the database.
- Only a subset of the d_Ref methods is supported by the TransRef smart pointer class:

```
operator*          : dereferencing the smart pointer
operator ->        : accessing a data member or method through
a pointer
operator == (int)  : comparison with NULL
operator != (int)  : comparison with NULL
```

- TransRefs do not provide a conversion operator to a pointer to a base class.

5.1.3 ODMG Fixed Length Types

In heterogeneous environments an ODMG compliant ODBMS may only provide a portable access to the following types with fixed value range are used. The C++ language defines for build in types a minimal value range rather than a fixed range. Therefore, persistent objects should contain only data members of ODMG defined fixed length base types or compounds these types.

Type Name	Range	Description
d_Float	32 bit	IEEE Std 754-1985
d_Double	64 bit	IEEE Std 754-1985
d_Short	16 bit	signed integer
d_Ushort	16 bit	unsigned integer
d_Long	32 bit	signed integer
d_Ulong	32 bit	unsigned integer
d_Char	8 bit	ASCII
d_Octet	8 bit	no interpretation
d_Boolean	0 or 1	defines false(0) and true(1)

Table 1: ODMG fixed length types

Type Name	Description
d_String	variable length string of characters
d_Date	date
d_Time	time
d_Timestamp	date and time
d_Interval	duration in time

Table 2: non-atomic ODMG types

5.1.4 Avoid Transient Memory References as Part of Persistent Objects

Raw C++ pointers or references which are embedded in persistent objects will become invalid when reloaded from the database. These pointers must be revalidated by the application before use, e.g. using the *d_activate()* and *d_deactivate()* methods defined by the ODMG³.

5.2 Objectivity/DB Implementation

5.2.1 Persistent Capable Objects Cannot Contain Other Persistent Capable Objects

Persistent capable objects, i.e. those that inherit from *d_Object*, cannot contain other objects which are also persistent capable. As a work-around, inheritance from *d_Object* can be added at the leaf class level, as shown below.

³ These methods are not yet implemented in the current version of Objectivity/DB, but have been requested for a future release.

```
// define a persistent track class
class Track : public d_Object {
public:
    d_Double phi;
    d_Double theta;
    // more definitions
    ....
};

// this will *NOT* work: Vertex is persistent and contains
// persistent capable Tracks
class Vertex : public d_Object {
public:
    Track first;
    Track second;
    // more definitions
    ....
};
```

Attempting to embed one persistent-capable object in another

```
// define a transient track class
class Track {
public:
    d_Double phi;
    d_Double theta;
    // more definitions
    ....
};

// define a persistent track
class PersistentTrack : public d_Object, public Track {
};

// no problem now: Vertex is persistent and contains simple
// transient Tracks
class Vertex : public d_Object {
public:
    Track first;
    Track second;
    // more definitions
    ....
};
```

Adding persistence at the leaf class level

5.2.2 Objectivity/DB References and Transient Objects

The smart pointer types implemented in the current versions of Objectivity /DB (ooHandle/ooRef/d_Ref) are not yet capable of referencing true transient objects that have been allocated on the normal program heap. It is therefore currently not possible to write code that can be called either with persistent or transient objects as arguments. Where such polymorphic behaviour is needed, transient objects may be simulated by creating persistent objects in a special database container which is deleted at the end of each transaction.

The HepODBMS package contains utility methods to implement this pseudo transient objects.

5.2.3 Avoid Creating Persistent and Transient Objects of the Same Class

Consider adding persistency through multiple inheritance on a separate leaf class, as shown in section 5.2.1 on page 22.

5.2.4 Passing of Persistent References and C++ Pointers

Avoid converting persistent references (HepRefs) to C++ pointers and then passing them to a method or subroutine. We have seen problems of dereferenced pointers from persistent references becoming invalid because the Ref destructor is called too early and the object has been removed from the DB cache. A simple way to prevent this is to pass the Ref(T) directly to subroutines.

```
bar(Track *t)
{
    cout << "phi=" << t->phi << endl;
}

foo()
{
    HepRef(Track) tr = HepNew(Track);

    tr->phi = 0;

    bar(tr); // implicit conversion to Track *
}
```

5.2.5 Initialisation of Persistent References from C++ Pointers

Persistent references (HepRefs) should not be initialised from C++ pointers other than those obtained from a persistent *new*.

```
// example:

void foo(Track *outerTrack)
{
    HepRef(Track) myTrack = outerTrack;

    ... more code which uses myTrack
}
```

- This example will not give any compiler warnings or errors
- The only exception of this rule is the initialisation from a HepNew operator.

```
//
// example :
//
HepRef(Track) newTrack = HepNew(Track);
```

5.2.6 Default Constructors of Embedded Classes

Default constructors of classes embedded in persistent capable classes should not have global side effects. Objectivity/DB creates one object of each persistent class known to an application to register it with the Objectivity/DB runtime system. This means that all constructors of embedded objects will be called before the first statement of `main()` is executed.

5.3 The HepODBMS Portability Layer

The HepODBMS package is intended to provide an ODMG-compliant coding style using the current, incomplete, ODMG implementations. In a limited way, the package also allows ODMG-compliant applications to be run in a fully transient mode, i.e. without access to an ODBMS at all. The latter has proven to be useful during the development of algorithms which do not need database access or on systems for which a ODBMS implementation does not yet exist.

Currently the package is still in early beta stage and has only been applied to RD45 internal porting projects. Nevertheless our positive experience with this package shows that an additional software layer between database and user code adds an important degree of freedom that helps isolating user code from database vendor and database version specific features and porting transient code to the database.

The functionality of the HepODBMS prototype package is described in more detail below.

5.3.1 Support for a Consistent Type Naming Scheme

The implementation of the ODMG base classes and templates is controlled by the database vendors. To have some additional independence of a vendor or a release of a database we suggest to introduce an additional naming layer on top of the ODMG classes. This will allow to respond centrally on problems or deficiencies of a particular implementation. Since this layer is typically implemented by *typedef* statements, it does not introduce any performance drawbacks.

5.3.2 Support Classes to Implement Clustering and Locking Strategies

The class `HepClusteringHint` provides a simple way to implement a consistent clustering strategy for a persistent class without introducing large code changes. See section 3.2.3 for more details. At the same time, this class may also be used to implement lock-free writing from multiple parallel processes into a single database.

5.3.3 Use of the `HepHintDeclare`, `HepHintInit` and `HepHintSet` Macros

- `HepHintDeclare` - declare a clustering hint member (as a class variable) within a class definition
- `HepHintInit(class)` - allocate the class variable and initialise it. Should appear once per program.

- `HepHintSet(class,location)` - Set the clustering hint for the class to a new location (e.g. another container or another object).

```
// in Track.ddl
class Track : public d_Object{
...
public:
  HepHintDeclare; // declare a clustering hint for the track class
...
};

// in Track.C

HepHintInit(Track); // initialise the clustering hint outside all
functions or methods
....

// in main.C

void DbApp::DbInit() // define the clustering policy in one central
location
{
  // create or open ReconstructedData data base from the federation
  db("ReconstructedData");

  // all Track objects created by HepNew will reside in the Tracks
  container
  HepHintSet(Track,container("Tracks"));

  // store all vertices together with the tracks
  HepHintSet(Vertex,Track::clusteringHint);

  // switch to RawData data base for following classes
  db("RawData");
  // all Hits objects created by HepNew will reside in the Hit container
  HepHintSet(Hit,container("Hit"));
}
```

5.3.4 Use *HepNew* and *HepDelete* Instead of the C++ *New* and *Delete* Operators

The new operator as defined in the ODMG standard takes additional arguments for clustering hints and type information. The `HepNew` macro allows these arguments to be supplied in a compliant way but also to allow switching back to a transient only version without code changes.

```
//
// example:
//
HepRef(Track) tr = HepNew(Track); // was: Track *tr = new Track;
...
HepDelete(tr); // was: delete tr;
```

`HepNew/HepDelete` will expand to a C++ `new/delete` if the data base is not available or deselected by the C++ preprocessor. `HepNew` supplies the additional clustering information for creating persistent objects clustered by class or close to another object.

5.3.5 Encapsulation of Database Session Control

There are a number of important functions related to database session control that are not standardised by the ODMG. To maintain vendor independence, we therefore provide an encapsulation of these functions in a database session class, which provides the following functionality:

- create or open a database by name,
- to set the application locking behaviour: waiting time, access mode,
- begin, abort and commit transactions,
- creation and retrieval of database and container objects.

The HepODBMS package provides the HepDbApplication base class as an interface to these services.

5.3.6 Support for Container Libraries

The parametrised types HepVector(T) and HepRefVector(T) provide variable length arrays containing persistent capable objects or references to persistent objects. Both vector classes can be used either as transient containers referencing persistent objects or as containers embedded in persistent objects. The interface of these vectors emulates the RWValOrderedVector<T> and RWPtrOrderedVector<T> templates in order to simplify the migration of code based on Tools.h++ to platforms which are not able to use the Objectivity version of the Tools.h++ library, because of lack of template support in the C++ compiler.

```
// in the TrackingDetector.ddl
declare(HepVector,Track);

class TrackingDetector : public d_Object {
public:
    HepVector(Track) itsTrack; // the detector contains its tracks
};

// in the TrackList.ddl file
declare(HepRefVector,Track);

class TrackList : public d_Object {
private:
    HepRefVector(Track) track; // the track list stores only
    references to tracks
};

// for example in the main.C file

#include "TrackList.h"

main()
{
    // ... some init code

    HepRef(TrackingDetector) tpc = HepNew(TrackingDetector);
    HepRef(TrackList) selected = HepNew(TrackList);

    // fill tpc with three tracks
    tpc->itsTracks.append( HepNew(Track) );
    tpc->itsTracks.append( HepNew(Track) );
    tpc->itsTracks.append( HepNew(Track) );

    // access one of them
    tpc->itsTrack[2].phi = 0;

    // put tcp track 2 into selection list
    selected->track[0] = tpc->itsTrack[2];

    // loop over all tcp tracks
    for (short t=0; t < tpcTracks.length(); t++)
        cout << tpc->itsTrack[t] << endl;

    // loop over all selected tracks
    for (short s=0; s < selected.length(); s++)
        cout << selected->track[t] << endl;
}
```


6. Conclusions

We have investigated and reported on how the use of an ODMG-compliant ODBMS for the storage and management of HEP event data impacts the object model, physical data organisation, coding conventions and the use of 3rd party class libraries. Although the use of an ODBMS clearly has a number of implications, we have developed, together with members of the user community, a series of conventions, guidelines and recommendations, described in this report, which not only minimise the impact of using an ODBMS, but also make concrete suggestions in a number of areas that can significantly increase the benefits of using such technology. These recommendations will continue to be developed, as the technology evolves, and made available through the Web.

7. Glossary

ADSM - A storage management product from IBM

AFS - the Andrew (distributed) filesystem

CORBA- the Common Object Request Broker Architecture, from the OMG

CORE - Centrally Operated Risc Environment

DFS - the OSF/DCE distributed filesystem, based upon AFS

DMIG - the Data Management Interface Group

GB - 10^9 bytes

HPSS - High Performance Storage System - a high-end mass storage system developed by a consortium consisting of end-user sites and commercial companies

KB - 2^{10} (1024) bytes - normally referred to as 10^3 bytes

IEEE - the Institute of Electrical and Electronics Engineers

MB - 10^6 bytes

MSS - a Mass Storage System

NFS - the Network Filesystem, developed by Sun

ODBMS - an Object Database Management System

ODMG - the Object Database Management Group, who develop standards of ODBMSes

OMG - the Object Management Group

OQL - the Object Query Language defined by the ODMG

ORB - an Object Request Broker

OSM - Open Storage Manager: a commercial MSS

PB - 10^{15} bytes

SQL - Standard Query Language: the language used for issuing queries against (relational) databases

SSSWG - the Storage System Standards Working Group

TB - 10^{12} bytes

VLDB - Very Large Database

VLM - Very Large Memory

VMLDB - Very Many Large Databases

XBSA - the draft X/Open Backup Services Application Program Interface

8. References

- [1] RD45 - A Persistent Object Manager for HEP, LCRB Status Report, March 1996, CERN/LHCC 96-15
- [2] RD45 - A Persistent Object Manager for HEP, LCB Status Report, March 1997, CERN/LHCC 97-6
- [3] Object Databases and Mass Storage Systems: The Prognosis, the RD45 collaboration, CERN/LHCC 96-17
- [4] Object Databases and their Impact on Storage-Related Aspects of HEP Computing, the RD45 collaboration, CERN/LHCC 97-7
- [5] Object Database Features and HEP Data Management, the RD45 collaboration, CERN/LHCC 97-8
- [6] Using and Object Database and Mass Storage System for Physics Analysis, the RD45 collaboration, CERN/LHCC 97-9
- [7] Where are Object Databases Heading? CERN/RD45/1996/4
- [8] Why Objectivity/DB? CERN/RD45/1996/6
- [9] Objectivity/DB Database Administration Issues. CERN/RD45/1996/7
- [10] Object Data Management. R.G.G. Cattell, Addison Wesley, ISBN 0-201-54748-1
- [11] DBMS Needs Assessment for Objects, Barry and Associates (release 3)
- [12] 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.
- [13] Object Oriented Databases: Technology, Applications and Products. Bindu R. Rao, McGraw Hill, ISBN 0-07-051279-5
- [14] Object Databases - The Essentials, Mary E. S. Loomis, Addison Wesley, ISBN 0-201-56341-X
- [15] An Evaluation of Object-Oriented Database Developments, Frank Manola, GTE Laboratories Incorporated
- [16] Modern Database Systems - The Object Model, Interoperability and Beyond, Won Kim, Addison Wesley, ISBN 0-201-59098-0
- [17] Objets et Bases de Donnees - le SGBD O₂, Michel Adiba, Christine Collet, Hermes, ISBN 2-86601-368-9
- [18] Object Management Group. The Common Object Request Broker: Architecture and Specification, Revision 1.1, OMG TC Document 91.12.1, 1991.

- [19] Object Management Group. Persistent Object Service Specification, Revision 1.0, OMG Document numbers 94-1-1 and 94-10-7.
- [20] The Object Database Standard, ODMG-93, Edited by R.G.G.Cattell, ISBN 1-55860-302-6, Morgan Kaufmann.
- [21] ADAMO Reference Manual, CERN ECP
- [22] HBOOK - Statistical Analysis and Histogramming Package - CERN Program Library Long Writeup, Y250
- [23] PAW - the Physics Analysis Workshop - CERN Program Library Long Writeup, Q121
- [24] ATLAS Computing Technical Proposal, CERN/LHCC 96-43
- [25] CMS Computing Technical Proposal, CERN/LHCC 96-45