CERN-LHCC-97-8

SCP

EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

CERN/LHCC 97-8
LCB/RD45
February 3, 1997

OBJECT DATABASE FEATURES AND HEP DATA MANAGEMENT

The RD45 collaboration
CERN, *Geneva, Switzerland*

We present an overview of a number of Object Database (ODBMS) features, such as replication, schema evolution and object versioning, their implementation in Objectivity/DB and other commercial databases and an analysis of how these features could be used to solve data management problems typical of the HEP environment. We also describe a number of prototypes that have been built to demonstrate the usability of these features in the HEP domain. This document has been produced in response to the second milestone set by the LCRB for the second year of the RD45 collaboration, namely:

"*Investigate and report on ways that Objectivity/DB features for replication, schema evolution and object versions can be used to solve data management problems typical of the HEP environment.*"

TABLE OF CONTENTS

1. Executive Summary

At the request of the LCRB, we have investigated and report below on a number of object database features (replication, object versioning and schema evolution), their implementation in existing ODBMS products and their applicability to HEP. Despite the fact that none of these features are currently covered by the ODMG standard for ODBMSs and the fact that only one ODBMS (Objectivity/DB) supports all three capabilities, it is our conclusion that all of these features provide significant advantages in the area of HEP data management, and can even be considered mandatory.

We continue to work with the ODMG to extend the standard in these areas.

2. Introduction

As shown in the table below, the LHC experiments will take vast volumes of data - approximately 1PB per experiment per year. This will result in extremely demanding data management requirements - far exceeding those of existing experiments.

Time interval	**ATLAS/CMS**	**ALICE**
	1MB/event, 100Hz	**40MB/event, 40Hz**
1 second	100 MB	1.6 GB
1 minute	6 GB	100 GB
1 hour	360 GB	6 TB
1 day	8.6 TB	140 TB
1 week	60 TB	1 PB
1 month	260 TB	
1 year (100 days)	**1 PB**	
TOTAL LHC (15 yrs)	**75 PB**	

LHC Data Volumes and Rates

In response to the 2nd milestone set for the RD45 project by the LCRB at the 1996 review, we have investigated a number of features of ODBMS products; the theory, how these features are implemented in existing products and how these features might be used to solve problems typical of HEP data management. We also describe prototypes that have been built to investigate the various scenarios described.

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 1 [5] and 3 [6]. In particular, performance related aspects of the database features described in this report are covered in [6]. Finally, this document assumes a working knowledge of object-oriented methods and object data management, as described in [10].

3. Object Database Features and Standards

None of the features described in this report are covered by the current version, V1.2, of the ODMG-93 standard for Object Database Management Systems (ODBMS). It is possible that they will be added to a post-V2.0 revision of the standard, version 2.0 being scheduled for completion during 1997. However, even if they are included in future revisions, it is unlikely that they will be implemented in a standard way in commercial products before the next revision of the ODMG standard, which one might expect during 1999 at the earliest. Although in some cases, such as with the recently-announced Java binding, vendors have responded very quickly to the evolving standard, we believe that it would be unwise to assume a standards-conforming implementation of the features described in this report before 2001, if at all.

In consequence, any implementation of these features is necessarily vendor-specific. We have, therefore, chosen to compare at least two implementations, typically that implemented in Objectivity/DB, the product being used for all of the prototyping in RD45, and another ODBMS[1].

[1] At the time of writing, Objectivity/DB is the only ODBMS that supports all three features.

4. Schema Evolution[2]

To store objects (e.g. instances of C++ classes) in an ODBMS, the class definition, or *schema*, must be made available to the database. In the ODMG standard, schema are defined using the Object Definition Language (ODL), which is based upon the Interface Definition Language (IDL) of the Object Management Group (OMG), which in turn is based upon C++ syntax. Thus, ODL has a strong C++ flavour and ODL files are closely related to C++ header files.

Schema evolution operations include changes to classes, class contents and relationships between classes, such as associations, references and inheritance. In an Object Database environment, changes are made to the schema by modifying the corresponding ODL files and then processing these files using the appropriate database tool, which then generates the C++ header files which need to be included in database applications that wish to access or create instances of such classes. Schema evolution is extremely flexible and supports the following operations:

- delete a data member
- add a data member with system default or user defined default values
- rename a member
- change the type of a data member
- reorder data members
- add/change/delete method definitions
- change the size of array data members
- add/delete/replace a base class
- add/delete/change associations or references
- change the cardinality of references
- etc.

In addition to changes to the database schema, the affected objects, i.e. the persistent instantiations of the schema, may be converted in a number of ways, e.g.

- **immediate** - all affected objects within a federation are changed using a special upgrade application,
- **deferred** - affected objects are changed only when accessed. This has the advantage that only those objects that are used are changed, but does mean that access times can be unpredictable during the conversion period,
- **on-demand** - affected objects in specific containers, databases, or within the entire federation are updated upon user request.

[2] See also: Appendix: *Schema Evolution in Objectivity/DB* on page 35.

It is clear that the data models of the various LHC collaborations will evolve with time. Support for such evolution by the database provides a very powerful and convenient mechanism for coping with such change.

4.1 Schema Evolution in Objectivity/DB

Objectivity/DB provides a powerful interface to change the federated database schema. The set of schema evolution operations supported includes changes to classes, class contents, as well as relationships between classes, namely associations, object references and inheritance. The objects of the changed classes, called *affected objects*, are also converted to reflect the effect of these changes.

Users specify schema evolution operations simply by changing their DDL files. The modified DDL files include the new definitions of the evolved classes. In certain cases, e.g. when a default value is to be assigned to a new data member, additional information may be given in the form of #pragmas. The modified DDL files should then be processed by the DDL preprocessor, which updates the federated database schema and generates a new set of header files.

Schema evolution operations affect existing objects, the internal class representation in the federated database schema, and existing applications. As a result of schema evolution operations, the following changes may take place:

1) existing objects may have to be converted to conform with their schema,
2) existing class representations may have to be modified,
3) existing applications may have to be rebuilt using a new set of generated files.

In order to address the possible side effects of schema evolution operations, Objectivity/DB categorizes schema evolution operations into two types with respect to their effect on persistent objects and on existing applications, namely ***persistent changes*** and ***non-persistent changes***. Persistent changes create a new representation of the modified class, while in the case of a non-persistent change, no new class representation is created for the affected classes. *Persistent changes* require modification of the affected objects, while *non-persistent changes* affect only the generated code from the DDL preprocessor and the corresponding schema that are stored in the federated database. *Non-persistent changes* allow existing applications to continue running on the affected federated database, while *persistent changes* prevent existing applications from accessing the federated database. *Persistent changes* require that existing applications are rebuilt - applications should be recompiled and relinked with the newly generated files. Without rebuilding, applications that use only classes that have not evolved can continue to access the federated database.

Instead of rebuilding all applications which use affected objects, conversion can be performed *"on the fly"* each time that an affected object is accessed, without storing the changes in the federated database. In this case overall performance can slow down, but as long as new applications access affected objects in read-mode only, the affected objects still remain accessible to existing applications.

The great majority of schema evolution operations are persistent. Only a few, like renaming a class or a data member, changing the access of a member, or adding a non-inline association are non-persistent. The last option allows for extra data members to be logically added to existing classes without affecting existing applications. New data members are stored in an additional class, and are accessed transparently using *accessor* functions, which traverse the association from old to new class. An example of such a scenario is given in section 4.7 on page 12.

Some schema evolution operations (such as renaming a class or a data member) are performed through the use of special #pragmas. The provision of additional information, such as the specification of the default value of a new data member, are also performed using #pragmas.

The majority of schema evolution operations are supported using a single pass of the DDL preprocessor. However, certain operations, such as adding a class as a non-leaf class or deleting a non-leaf class, require an additional pass.

The conversion of objects of *persistently changed* classes can be performed using either ***data conversion*** or ***function conversion***. *Data conversion* does not require the user to write any program to convert the affected objects. *Function conversion* enables the user to provide a function for each persistent-capable class that has been changed. This function is called automatically in the process of converting an affected object. Using such a function, the user is able to access the data members of existing objects both before and after data conversion. If a transaction is read-only, the converted objects are not stored in the federated database.

A function conversion is mandatory in only two cases: when adding a class to, or removing a class from the middle of the inheritance hierarchy. In addition, function conversion can be useful for initializing new values of data members from the values of existing data members. It can also be used to change the value of existing data members[3].

In order to convert the affected objects, Objectivity/DB supports immediate (eager), deferred (lazy), and on-demand methods. Respectively, three different interfaces can be used: *upgrade application*, *single-object-at-a-time*, and *on-demand*. The first interface, using an upgrade application, supports complex schema evolution operations (such as deleting a non-leaf class or replacing a base class). The second interface converts an affected object only when it is accessed; during one approach only a single object can be converted - if there are multiple objects that require conversion, the *upgrade application* interface must be used.

Objectivity/DB also provides administration tools for downloading schema into files and for the loading of schema from files. Schema dump/load permits changes to production databases without the need for distribution of header files.

[3] In a future release, the default constructor will be called for new data members.

4.2 Schema Evolution in O_2

As is the case with Objectivity/DB, O_2 also defines two types of schema evolution operations: persistent and non-persistent changes (called respectively *updates which affect objects*, and *updates with no effect on objects*). After changing a schema, all modifications must be **confirmed**. When an application accesses an object, O_2 checks its structure and automatically performs an update if required. The values contained in the object are kept, whenever possible, in the new object representation. Object conversion is carried out by the system, but users can supplement an implicit conversion with their own conversion procedure. O_2 provides automatic conversion of the following types: atomic (primitive), object references, collections and tuple types (structures).

Changing between *primitive types*, such as *int, float* etc., is implemented through the use of functions: sprintf(), atoi(), atof() and some additional tricks, which allows O_2 to provide conversion from integer, real, boolean, char, bytes and string to another primitive types, often with a partial lost of data. This is in contrast to Objectivity/DB, which only permits meaningful conversions - in other cases, no attempt is made to retain the value of the old data member.

When changing a *reference*, if the new class is a super class of the old class, the reference is unchanged; in any other case it becomes nil. The *collections* managed by O_2 are list, bag (multi-set) and set (unique set). When the new collection and the old one are equal or a standard conversion can be applied, data from the old collection is used to initialize the new one. When no conversion is detected the resulting collection is empty. When a *tuple* is converted into another tuple, each tuple field is converted successively, and a field is converted recursively. A tuple cannot be directly changed into a collection, and vice versa. If you need to provide such a conversion you need to delete the source class and replace it with the new one with the same name.

Issues such as changing the order of data members, changing the access of a member, or more sophisticated features like adding/deleting a class as a non-leaf class and so on, are not covered in the current O_2 documentation.

O_2 provides two methods: deferred and immediate conversion, both performed in a standard way.

Function conversion can be performed instead of the standard O_2 conversion if needed. To perform function conversion, one should update the classes without deleting attributes and add new attributes with different names; then write and run a program to access all modified objects and initialize the new attributes with the required values. Finally, the old attributes can be deleted and if necessary renamed. Function conversion acts as immediate conversion, preventing other applications from accessing database whilst the conversion is being performed.

The schema manager stores information about the evolution of a class structure in a *class history*. When an application is being designed, schema definitions typically evolve, but it may not be desirable to retain the complete history of all updates. As mentioned above, all schema modifications must be **confirmed** and a special command is provided for this purpose. A class can only be updated explicitly using this command. By not confirming a schema change, it is possible to return to the previous definition without any modifications to the database. To perform this behaviour, O_2 defines two different types of database: database in controlled mode and in test mode.

Controlled mode means that object updates are always possible and carried out by the system. For the system to carry out the updates, all the classes definitions must be confirmed. *Test* mode means that the database can be used even if its class definitions are not confirmed. However, in test mode, non-confirmed evolution is not supported. *Controlled* mode can be converted to *test* mode, but not vice versa. When a database is created, its default status is *controlled.* However, if a schema change is not confirmed, the system implicitly creates a database in *test* mode. To change back to controlled mode, it is necessary to delete the entire database, confirm the schema changes and recreate the database.

According to the O_2 documentation, an upgrade to existing schema should not be performed directly on the target system. A new version of a schema should be prepared first on the source system, and then installed on a target site. Two special tools are provided to perform this operation: *o2schema_load* and *o2schema_dump* (in Objectivity/DB this functionality will be provided in a future release). These tools use dump files to store and retrieve schema information. They can also be used for storing and recovering the contents of a database.

4.3 Schema Evolution and HEP Data Management

In order to investigate the usefulness of schema evolution in solving data management problems typical of the HEP environment, we have performed the following investigations:

- a comparison of the functionality provided in Objectivity/DB with the requirements from ALEPH and L3,
- the practicalities of the various types of data migration offered (lazy, immediate)
- the impact of schema evolution and object instance migration on user applications.

4.4 Schema Evolution in Existing HEP Experiments

Data management packages, such as those traditionally used in HEP, offer little or no support for schema evolution. Typical cases of schema evolution include:

- change of bank contents, e.g. extend a bank/table;
- change of data structure, e.g. add a new bank/table and/or reference link.

Normally, it is left to the application to handle the changed schema, e.g. by checking a version number in the data structure itself, and invoking conditional code. This is not only unsafe - code that does not correctly handle data generated according to different schema can give wrong results or, preferably, crash, but also requires applications to carry "baggage" corresponding to all versions of the schema that are developed over the lifetime of an experiment.

In practice, it is not uncommon for old data to be reprocessed, so as to be converted to new schema, or for spare fields to be reserved for new information.

4.5 Schema Evolution in L3

The L3 experiment changed its storage strategy in 1995. Prior to this date, the whole data structure, called DRE, was saved. From 1995 on, only a compressed version of this data structure, called DSU, was stored persistently. The DRE structure is recreated in memory from the DSU structure using an ad-hoc routine on input. In ODBMS terms, this is the approximate equivalent of an *activator.*

Schema evolution is handled by special routines in the production software - general cases are handled by the system software and special cases are handled by specific routines for each bank.

In general when an event is read a new "fanout" is created and then individual banks are checked against the present schema (hard coded in a common block) before being shunted to the new fanout. If the "schema" is different, a conversion routine is called.

Typical code is shown below:

```
*
* *** Check the EVNT bank fanout first
*
   CALL ZSHUNT (IXSTOR, LBEVNT, LBAAAA, 2, 1)
   LBEVNT = 0
   CALL UTEFAN (LBEVNT)
   LREFRB(1) = LBAAAA
   LREFRB(2) = LBEVNT
   CALL RECMPR (LREFRB, 1, NRECRB, CRECRB, IDRERB, IOKFRB)
   NWDS   = MIN (IQ(LBEVNT-5), IQ(LBAAAA-5)) - 1
   CALL UCOPY (IQ(LBAAAA+1), IQ(LBEVNT+1), NWDS)
   DO 10 IL = 0, NPEVNT
    IF (LQ(LBAAAA-IL).GT.0)
  +   CALL ZSHUNT (IXSTOR, LQ(LBAAAA-IL), LBEVNT, -IL, 1)
 10 CONTINUE
   CALL RESHUN (LREFRB, NRECRB, CRECRB, IDRERB, 0)
   CALL MZDROP (IXSTOR, LBAAAA, 'L')
   LBAAAA = 0
```

where:

```
*
******************************************************************************
*
*     SUBR. RECMPR (LREF, IFLG, NCBK*, CHBK*, IDESC*, IOK*)
*
*  Compares two fanout banks (xREC or EVNT) and checks the list of
*  constituents and their properties
```

```
*
*  Arguments :
*
*    LREF    Reference links of the two fanout banks
*    IFLG    0 for xREC; 1 for EVNT
*    NCBK    Number of constituent banks [xOBJ(xREC) for xREC(EVNT)]
*    CHBK    Names of the constituent banks
*    IDESC   Description of the two sets of constituents (KL/NS/NF/
*            NW/NC). If the constituent does not exist in one case,
*            corresponding KL would be -1
*    IOK     If agrrement between the 2 fanouts (first element refers
*            to overall agreement; second element refers to PP's)
*
*  Called by RECHCK
*
*************************************************************************
*
```

User conversion routines are supposed to detect that a bank is different because the number of words or links is different using code like this:

```
*
* *** Loop over all banks in the old fanout and shunt appropriately
*
      LBTREC = LQ(LBEVNT-KLTREC)
      LBZZZZ = 0
      IF (LBAAAA.GT.0.AND.LBTREC.GT.0) THEN
        DO 10 IBK = 1, NCBK
          CHC    = CHBK(IBK)
*           They must exist in old and new setups
          IF (IDESC(1,1,IBK).GT.0.AND.IDESC(2,1,IBK).GT.0) THEN
*           They should have same NS/NW/NF
            IF ((IDESC(1,2,IBK).EQ.IDESC(2,2,IBK)) .AND.
     1          (IDESC(1,3,IBK).EQ.IDESC(2,3,IBK)) .AND.
     2          (IDESC(1,4,IBK).EQ.IDESC(2,4,IBK)) .AND.
     3          (IDESC(1,5,IBK).EQ.IDESC(2,5,IBK))) THEN
*           Shunt it if they have same NS/NW/NF
              LBNK   = LQ(LBAAAA-IDESC(1,1,IBK))
              IF (LBNK.GT.0) THEN
                CALL ZSHUNT (IXSTOR, LBNK, LBTREC, -IDESC(2,1,IBK), 1)
              ENDIF
            ELSE
```

after the ELSE some ugly code usually follows.

For example this is a code due to V.I. to add more words and links to a MUon TracK due to the addition of forward chambers:

```
        NWT   = IQ(LBMUTK-1)
        ipush = 0
        Nsold = IQ(LBMUTK-2)
        ipush = NSMUTK-NSold
        nlold = IQ(LBMUTK-3)
        iver = 200
        if (Nsold.eq.3) then ! version < 190
          iver = 180
        elseif (Nsold.eq.4) then ! version 19n
          iver = 190
        elseif(NSMUTK.ne.Nsold) then  ! unknown to us
           print *, 'in MUTK number of structural link is not correct!'
        endif
*
......
```

skipping code where it goes on computing various number of words NWx

```
.....
        NIW   = IPOIN + NWZ + NWF + NWR + NWB + NWM - LBMUTK - NWT
        CALL MZPUSH(IMDVRA,LBMUTK,ipush,NIW,' ')
        if (ipush.ne.0) then   ! I hope J.Z. does not kill me
*          start to move reference links
```

```
        do il=nlold,nsold+1,-1
         LQ(LBMUTK-il-ipush) =  LQ(LBMUTK-il)
        enddo
*        change the number of structural links
        IQ(LBMUTK-2) = IQ(LBMUTK-2) + ipush
*         and shunt all the banks around
        if (iver.eq.180) then
         nsl = 2
        elseif (iver.eq.190) then
         nsl = 3
        else
         print *,' MUBFIX: unknown version',iver
        endif
        do il=nsold,nsl,-1
          lq(lbmutk-il-ipush) = 0
         if (lq(lbmutk-il).ne.0) then
           call ZSHUNT(IMDVRA,lq(lbmutk-il),lbmutk,-il-ipush,1)
         endif
        enddo
       endif
* -------        p-seg
      IPOIN = LBMUTK + IQ(LBMUTK+1)
      if (ipush.ne.0.and.IQ(IPOIN).gt.100) then   ! v < 200
       iloc = ipoin
       do ic = 1,IQ(IPOIN)/100
        iq(iloc+1) = iq(iloc+1) + ipush  ! refernce link position
        iloc = iloc + MOD(IQ(IPOIN),100)
       enddo
      endif
* -------
... and so on so forth for the rest of the bank
```

4.6 Schema Evolution in Adamo

The following text is extracted from the following sections of the Adamo reference manual:

- http://www.cern.ch:80/Adamo/guide/Section-9-2.html
- http://www.cern.ch:80/Adamo/guide/Section-10-6.html

More information can also be found in:

- http://www.cern.ch:80/Adamo/refmanual/Chapter-4-8.html
- http://www.cern.ch:80/Adamo/refmanual/Section-4-8-3.html

4.6.1 Generic ADAMO Files

In ADAMO, all data manipulation and navigation operations take place in memory. GAF routines allow the user to store all the tables and dataflows present in memory at a given moment, with any or all of the associated indices and selectors if required.

4.6.1.1 The GAF Record

A GAF contains records, the contents of which must be either a table or a dataflow. Thus, several tables may be stored on the same record as long as the tables belong to the same dataflow.

As suggested by Figure 9.2, the set of tables stored in a record are loaded into memory when the appropriate routine is called. As an example, if GAFs are being used to record high energy physics events then all the tables describing an event are stored in a record. Every version of the event is stored in a separate record. Access to a particular version of an event is through a GAF reading routine which drops the previous contents of the tables and replaces it by the data in the record or the desired version.

4.6.1.2 Storage of the Data Model

Records store not only data but also the corresponding data model. This feature allows GAF routines to deal with the versions of the continuously evolving data model; tables, attributes and relationships may be added, deleted or modified during the life time of an experiment. When such changes occur, the data stored within old GAFs are no longer conformant with the new data definition. Such data definition mismatches are automatically resolved by the selection and fetching of the conformant parts of the old data. This is explained in [Section 10.6] the following section.

4.6.2 Reading Old GAFs

When debugged and tested, the new release of the application is ready for production. But what about backward compatibility with old data? In most data management systems, this problem is handled by converting old data files to conform to the new data definition. An alternative solution to this may be to keep in production all successive versions of the application. Both solutions are unsatisfactory because they imply further programming and bookkeeping work. The Generic ADAMO File system offers “on the fly” conversion of old data.

4.6.2.1 Storage of the Data Definition

The ADAMO programmer needs to do absolutely nothing to be able to read old GAFs. GAF records store the data definition along with the data. Since the data definition of an application is also stored in memory, GAF routines are able to compare the data definition of an application in memory with those on a GAF record and then to simply read from a GAF only the data that conform to the application data definition.

4.6.2.2 Tap Warning Message

This is the case with the modified eventrec application; the old GAF (rawgaf.ie) can still be read without the need of making any changes. When reading the GAF, the TAP generates the following message:

TAP Warning - projection generated for: Track

to tell that there has been a rearrangement of the data within the new structure of the table Track.

4.6.2.3 The New Track Table

The new table is shown in Figure 10.6. The column P has been renamed Momentum and two new columns appear corresponding to the new attribute and the new relationship: Length and ProducedAt.

4.7 Testing Schema Evolution

For detailed understanding of all schema evolution aspects, we have built a small prototype for testing schema operations, consequences of schema changes, its influence on the performance and many others. We decided to use very easy model of schema consisting just of a few classes, with several associations of different types between them. The model, we decided to start with contained some "mistakes", not used fields or inaccurate types. During next steps, we tried to change our schema in order to fix all inconveniences, putting most effort on **understanding** what are the consequences of operations being performed. We wanted to make clear, which operations really need rebuilding all existing applications, and for which operations this can be avoided. As it is not necessary, we did not put much effort on the interface part of the prototype: the communication with a user is as easy as possible: just in text mode. Below, we present a complete list of all schema evolution operations available in the version 4.0.1 of Objectivity/DB system, distinguishing clearly *conversion* from *non-conversion* operations.

4.7.1 Operations which do not affect existing applications

The following operations have no affect on existing application:

1) Adding a non-inline association,
2) Renaming a data member / an association,
3) Changing the access control of a data member / of an association, e.g. *public, protected, private,*
4) Changing the behaviour of associations, as described below[4],
5) Changing the order (position) of a non-inline association,

As is clear from this list, the schema changes which do not affect existing applications are relatively minor.

4.7.2 Operations which affect existing applications

The following operations require that existing applications are rebuilt:

1) Adding:
   - data member
   - object reference
   - inline association
   - the first virtual member function
2) Deleting:
   - data member
   - association
   - object reference

[4] In the Objectivity/DB, behaviour specifiers are used to specify how the association to the old object will be handled during following operations:
- lock -> lock(propagate)
- delete -> delete(propagate)
- copy -> copy(copy), copy(move), copy(delete)
- versioning -> version(copy), version(move), version(delete)

3) Changing the order (position) of:
   - data member
   - inline / non-inline association
   - object reference
4) Removing the last virtual member function
5) Change the size of a fixed-length array data member
6) Changing an object reference from short to long and vice versa
7) Changing the representation (inline - non-inline) of an association
8) Changing an inline association from short to long or vice versa
9) Changing an array size for an array of object references
10) Changing a primitive data member type:

Convert from...	**Convert to ...**	
	may not preserve the value	preserves the value
int8	int16, int32, float32, float64	uint8
int16	int32, float32, float64	int8, int16[x]
int32	float64	float32, int8, int16, int32 [x]
float32	float64	int8, int16, int32
float64		int8, int16, int32, float64

[x] - Values not preserved only if converting from signed to unsigned and vice versa

Although some schema changes will always require that existing applications are rebuilt, it is clearly important to carefully investigate ways that this can be avoided. Of the above items, perhaps the most significant is the addition of new data members. As the addition of a non-inline association is a non-conversion operation, that is, does not require that existing applications are rebuilt, additional data members can be "added" by adding a non-inline association and storing the additional data in a new class. This is discussed in more detail below.

4.7.3 Simulating a New Member

Below, we consider a concrete example where a new data member is "added" by using a non-inline association to a new class.

Suppose, that a new member "*int nrOfStudents*" should be added to a persistent class *Teacher,* for storing the number of all students taught by that teacher. Assuming that only a small fraction of applications will need to access this new field, we want to perform this operation without interfering all existing applications, even if the use of an association to a new class implies a small performance penalty.

To make such a change, we first need to create a new persistent class, e.g. *TeacherSupplement*:

```
class TeacherSupplement : public ooObj {
     public :
          int nrOfStudents ;                                  // a member

          TeacherSupplement() { nrOfStudents = 0 ; } ;        // the constructor
} ;
```

and add to the class *"Teacher"*:

1) a non-inline association to a class *TeacherSupplement:*
   ooRef(TeacherSupplement) supplementClass ;

2) methods:

```
inline int getNrOfStudents() { return ooThis()->supplementClass->nrOfStudents ; } ;
inline void incrNrOfStudents() { ooThis()->supplementClass->nrOfStudents ++ ; } ;
```

We must now rebuild the schema stored in our federated database passing to the DDL preprocessor our ddl file(s) containing a definition of both classes: *Teacher* and *TeacherSupplement*. The preprocessor should be run with the *-evolve* flag, as we are adding a new association to a *Teacher* class.

Now we can "attach" our "new, external member", for example:

```
Teacher::Teacher( /* parameters */ )
{
     ooThis()->supplementClass = new( ooThis().containedIn() ) TeacherSupplement ;

     //     other constructor's methods
}
```

The consequences of executing this constructor areas follows:

For every object of class *Teacher*:

- an object of a class *TeacherSupplement* is created automatically:
  - with a member *nrOfStudents* = 0 (set in the constructor of *TeacherSupplement* class)
  - in the same container, where the parent object is located,
- an association *supplementClass* is set to point to the newly created object of the *TeacherSupplement* class.

Existing instances of the class *Teacher* must now be changed. This is performed using a conversion function. To understand how this function works, the definition of several classes must be given, as shown below:

```
class Teacher : public ooObj {
  public:
        Teacher( /* parameters */ ) ;
        inline ooRef(Student) myStudents[] <-> teachers[] ;
        inline ooRef(TeacherManager) tManager <-> allTeachers[] ;
        // other fields and methods ...
  // simulation of a new member:
        inline int getNrOfStudents() { return ooThis()->supplementClass->nrOfStudents ; } ;
        inline void incrNrOfStudents() { ooThis()->supplementClass->nrOfStudents ++ ; } ;

        ooRef(TeacherSupplement) supplementClass ;
} ;

class Student : public ooObj {
   public:
        inline ooRef(Teacher) teachers[] <-> myStudents[] ;
        // other fields and methods ...
} ;

class TeacherManager : public ooObj {
   public:
        void initAfterAddindNewMember() ;
        inline ooRef(Teacher) allTeachers[] <-> tManager ;
        // other fields and methods ...
} ;
```

Now we can define the function:

```
void TeacherManager::initAfterAddindNewMember()
{
     ooItr(Teacher) teachI ;
     ooThis()->allTeachers(teachI) ;          // set iterator for traversing through all teachers

     while ( teachI.next() )                  // traverse through all teachers
     {
         teachI->supplementClass = new( ooThis().containedIn() ) TeacherSupplement ;

         ooItr(Student) studI ;
         teachI->myStudents(studI, oocNoOpen) ;      // set iterator for traversing through all students
         while( studI.next() )                       // count students, set new member
              teachI->incrNrOfStudents() ;
     }
}
```

The function *initAfterAddindNewMember* should be run **only once**, before using the new member.

Now we can rebuild applications which want to use the “new” member. The details of accessing the “new” member via the association are hidden in the functions *getNrOfStudents()* and *incrNrOfStudents()*.

The procedure presented above is fully transparent to other applications, they can continue accessing federated database without any limitations but, clearly, they do not know about the existence of the “new” member.

The side-effect, and probably the main shortcomings of such procedure are:

a) one additional object must be created for every object of “affected” class,
b) accessing this object through a non-inline association will influence the performance.

4.7.4 Complex Operations

Complex class content changes can be performed by combining multiple basic operations. Complex operations include:

1) Adding a data member and copy its value from another member,
2) Changing the type of a data member to either a base class or derived class,
3) Changing the class of origin of a data member,
4) Changing the dimensions of a array data member,
5) Changing a fixed-size array to a variable-size one,
6) Changing the type of a non-primitive data member,
7) Changing an object reference to an association,
8) Changing the cardinality of an association,
9) Changing the domain class of an association.

4.8 Conclusions

Schema evolution is clearly an important facility for large and/or long-lived projects. Further investigations, using a large federation, need to be made of the different conversion possibilities and of the use of additional classes to simulate the addition of new data members in existing classes.

The schema evolution capabilities offered by ODBMSs greatly exceed those of existing packages. However, it is clear that guidelines need to be established so that this capability is used in an efficient and manageable fashion.

5. Object Versioning

Object versioning is a capability that permits an application to create separate versions of individual objects, or in some cases also versions of collections of objects. The requirement for support for object versioning in ODBMS products originally arose from applications such as CAD or CASE, where access to various states of an object during the course of its evolution are a fundamental part of the application.

Object versioning may be used for a number of different purposes, including:

1) to keep track of the history of an object,
2) to enable concurrent access to the same object,
3) to deal with the problem of changing type definitions.

Object history

Calibration objects may be considered to be an example of the first case. These are typically time-varying objects, although in some cases are accessed via a logically equivalent key, such as run and event number. The use of object versioning allows the complete history of the different calibrations to be maintained, including branches. Implementations of object versioning typically provides support for a default version. In the case of calibration objects, however, a default version has no obvious meaning.

Concurrent access

There are times when different users need to work concurrently with their own set of data stored in the same object, or need to update the same object independently. Examples include the testing of new algorithms, such as track fitting, on production data. Even though one may wish the new objects to be persistent, they are in some sense private, and should not be visible to the rest of the collaboration until such time as the new algorithm is officially approved and becomes part of the production software. Equally, there may be several people testing alternative algorithms in parallel. Object versioning provides a clean solution - it enables concurrent access to different versions with no locks or collision. Every user can maintain their own versions, including branches (which can later be merged if required), and sharing of versions between different users is also possible. One may also define a default version, which can be redefined at any time. However, the creation of versions clearly requires write permission to the database and thus cannot be safely granted to the entire collaboration.

Changing type definitions

Examples of the third case are currently restricted to the field of schema evolution - no known ODBMS mixes schema evolution with object versioning. Such a behaviour could be applied in some applications probably with a greater success than schema evolution alone. The possibility of using different schema for different versions of the same object is much more powerful than changing the schema of a base object.

The ODBMS may provide a *mechanism* or a *policy* for version management. Existing products do not support built-in *policies* in their products, preferring to provide a complete *mechanism* for building *policies* instead. Having that *mechanism*, users can easily provide the appropriate *policy*. Both Objectivity/DB and O_2 provide a *mechanism*, leaving the management problem to be solved by the user. A very easy example of a policy, that could be provided by vendors, would be keeping track of the object creator. This could be implemented to allow individual users to see only object versions that they themselves had created, without even begin aware of the existence of other versions. This could diminish the problem of access control and inadvertent data loss or corruption due to conflicting updates by different users. Naturally this can be supported in the user code using the above mentioned *mechanism* - about 10 lines and that is all!

Both Objectivity/DB and O_2 implement a new version of an object by making a copy of the parent object *(versioning-by-copy)*. A second approach exists and is called *versioning-by-difference* (or *delta versioning*); this approach is not popular among modern ODBMS products in era of large and cheap data storage systems. This unpopular approach stores only differences between versions instead of entire objects. It requires a lot of additional time for reconstructing a version, saving a little disk space instead. This approach could be applied successfully for large objects, where differences between successive versions are small.

The ODBMS may provide a protection of versions from unauthorized or inadvertent access. This feature is not currently available either in Objectivity/DB or in O_2. O_2 provides two different polices for accessing objects: writable and read-only, which can be regarded as a security feature.

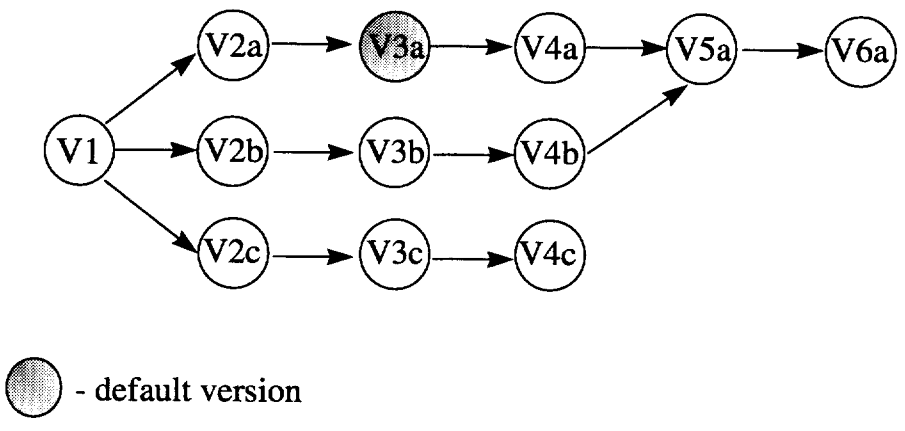
5.1 Object Versioning in Objectivity/DB

5.1.1 Version Genealogy

The complete set of versions of an object in Objectivity/DB's terminology is called a *version genealogy*. Creation a new version of an object simply means adding a new object to the genealogy. Depending on the *VersioningMode* attribute associated with every object, an object can be versioned in *linear* mode, *branch* mode, or versioning can be simply disabled. The *VersioningMode* flag can be changed at any time, hence switching between modes is very easy. The most basic functionality required for versions is creation and deletion of versions. A user can create a new version of an object explicitly, or a new version can be created implicitly whenever the object is modified. Within a *version genealogy* one version can be singled out; then it is called a *default version*. Working with a default version is extremely fast and flexible: dedicated associations enable convenient locating the version, and special tools help to maintain it easily. A default version is not created automatically - it has to be set explicitly by the user. Yet another nice feature of

Objectivity/DB's versioning is possibility of merging versions; recollect an example from the preceding subsection called *"Concurrent access"* to consider importance of that feature.

The figure below shows an example of a *version genealogy* with a default version marked, cases where both linear and branch versioning are applied, and where two versions are merged.

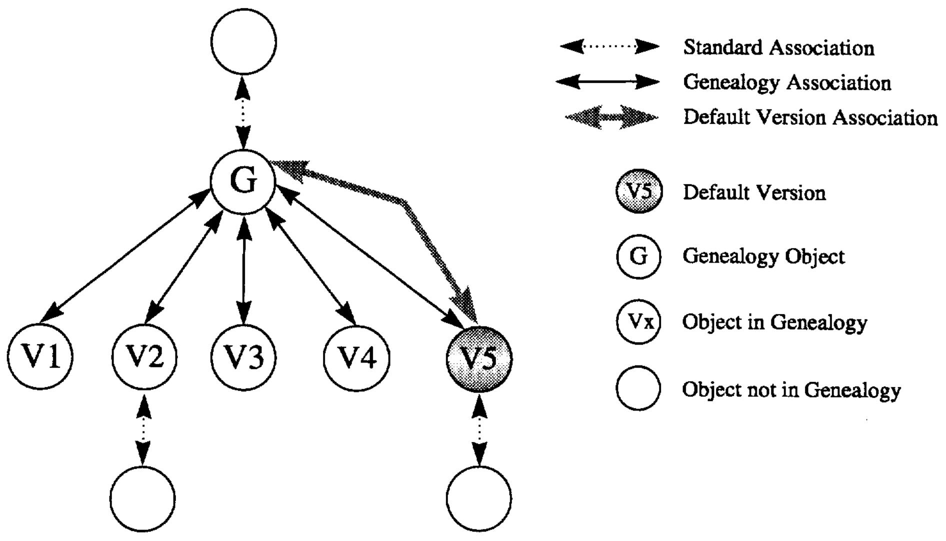


Example of a "Version Genealogy"

While creating of a new version of an object, Objectivity/DB performs a bit-wise copy of the existing object, possibly adding some associations. Whenever a new version of an object is created, all associations concerning this object remain untouched. Whether the semantics of creating a new version should be the same as that of a copy constructor or an object assignment depends on the application. For this reason, an empty virtual function *ooNewVersInit()* is called automatically just after the new version creation; the function does nothing but return a success status. Overloading this function in a user's application's persistent class gives him a broad influence on the initialization of a new version. Using the function, user can specify the exact semantics that fits the version copy requirements.

5.1.2 Genealogy Object

The versioning features of Objectivity/DB are implemented through the use of system defined association links and member functions that are automatically defined by the DDL preprocessor. Every version keeps track to his parent and child versions (*prevVers* and *nextVers* links); if needed, the order parent-child can be inverted. For further simplification of maintaining any number of versions, Objectivity/DB uses a *genealogy object*, which is an additional object created automatically when a default version is set. The *genealogy object* is an instance of a class *ooGeneObj* or of a user-defined subclass of *ooGeneObj.* Every *version genealogy* can have associated one *genealogy object*. The *genealogy object* serves as a multiplexer, allowing objects to associate to the genealogy as a whole (*allVers* link); it can be customized to hold information that pertains to the genealogy as a whole; it is also used to support a default version of an object. Within a genealogy, the default version may only be accessed from the default version itself and from those versions created after the initial default version was designated. The figure below shows a position of *genealogy object* inside a sample version genealogy. The genealogy object is "attached" to the version genealogy through associations: one per version. Standard associations between genealogy object or versioned objects and any other objects which are not in the genealogy and may be in different database or even off-line are also supported.



The "Genealogy Object" in Objectivity/DB

The Objectivity/DB-defined association links to support versioning are confined to the classes *ooObj* and *ooGeneObj*. The following paired bi-directional association links are available:

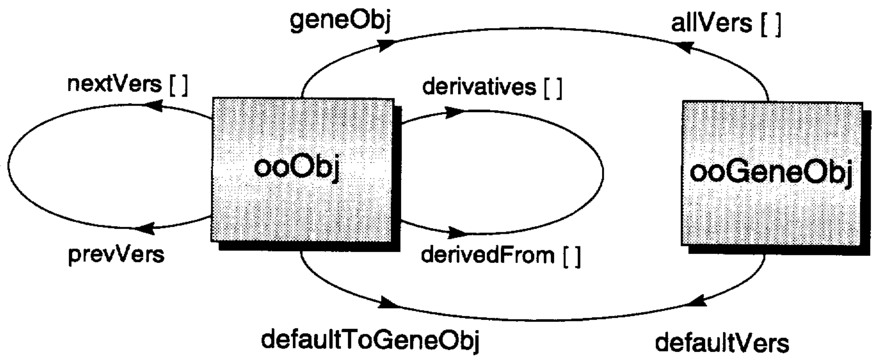
a) association between all versions[5] and the genealogy object[6] *(ooObj::geneObj* and *ooGeneObj::allVers[])*

b) association between default version[5] and the genealogy object *(ooObj::defaultToGeneObj* and *ooGeneObj::defaultVers)*

c) association between parent and child versions[5] *(ooObj::nextVers* and *ooObj::prevVers)*

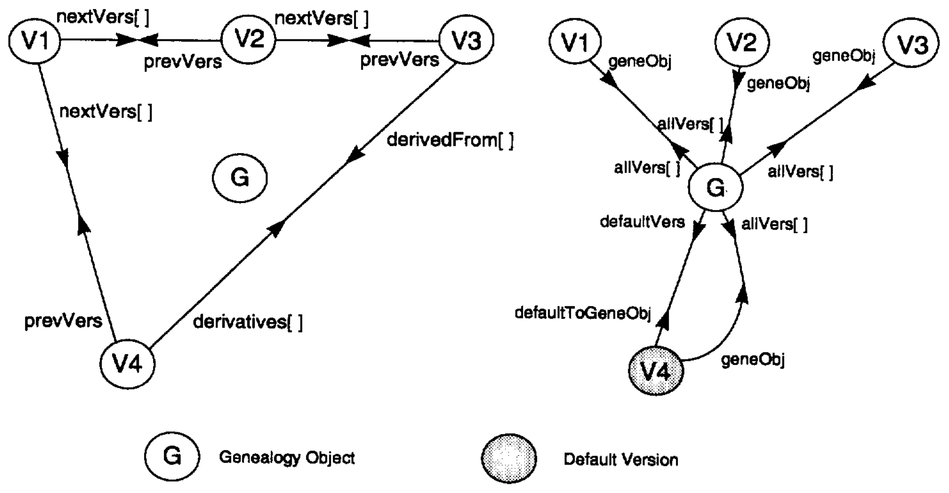
d) like c), if a version branch is being merged *(ooObj::derivatives[]* and *ooObj::derivedFrom[])*

[5] Instance(s) of *ooObj* class or a user-defined class(s) derived from *ooObj*.
[6] Instance of *ooGeneObj* class or a user-defined class derived from *ooGeneObj*.



These associations may be modeled as shown in the figure above. The figure below shows an example of how these association links might be used within a simple version genealogy. Every version within the genealogy accesses genealogy object via *geneObj* association, From the other side: *genealogy object* has the access to every version via *allVers* association. An additional link between genealogy object and default version exists: genealogy object locates default version using *defaultVers*, beeing located itself by default version through *defaultToGeneObj* link.

Version V1 and V3 are being merged. The V4 version is actually created from V1 version, and derivatives / derivedFrom links generally indicate that a version branch is being merged.

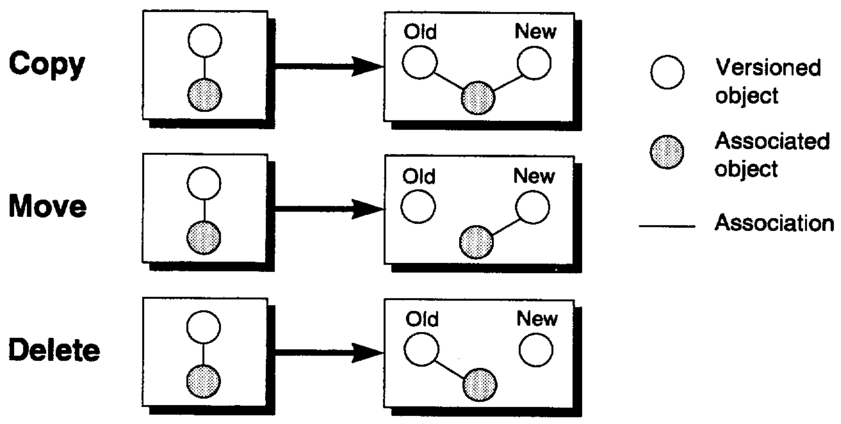


5.1.3 Versioning Behaviour for Associations

When declaring in a DDL file an association between objects, at the same time the user can also define versioning behaviour for the association, choosing between three alternatives:

– copy
– move
– delete.

The figure below explains fully how this works in practice.



Delete behaviour is the default - this will be applied unless otherwise specified. As the next line shows, defining the behaviour is extremely easy:

ooRef(myClass) assocOne[] <-> assocTwo : **version(move)** ;

and needs only two additional words. This example set the behaviour of association *assocOne* to "move".

5.1.4 Others Versioning Features

Objectivity/DB provides complete set of tools for creating, deleting and efficient locating of versions. To create a new version, one should:

1. enable versioning
2. close a "parent" object
3. open an object in update mode
4. optionally set a new version as a default one
5. optionally disable versioning

All needed associations are set by the system - the user does not need define them. This approach is very simple, but does not give the full power of Objectivity/DB's versioning. To use the *genealogy object* to its full potential, the user must create it manually and use the association interface to versioning. A customized genealogy object class can be defined simply by creating a subclass of *ooGeneObj.*

Also the precise use of the association links is fully up to the application developer. By directly manipulating and managing these associations user can:

- create customized versioning semantics
- use customized genealogy object classes.

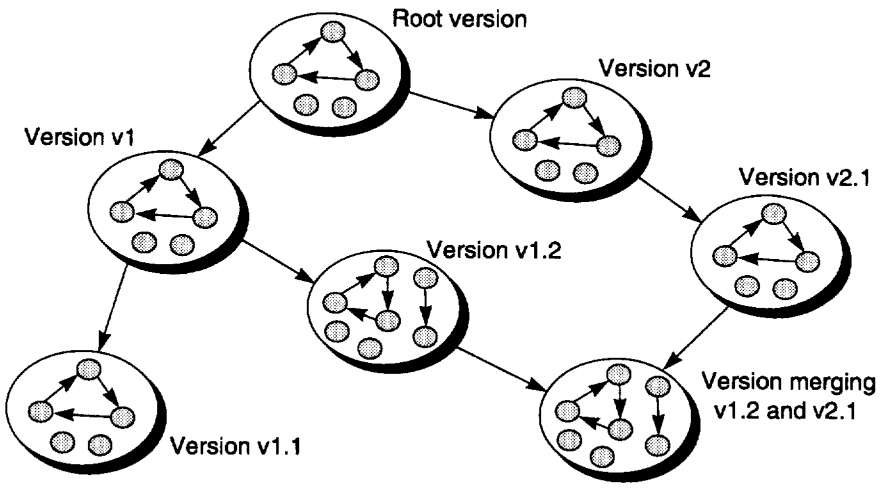
In addition to standard tools for locating versions, system names or scope names can be used whenever it simplifies the identification of versions. System name uniquely identifies an assigned object within entire federation. A scope name is a name that uniquely identifies an object within the *name scope* of itself or other objects. A *name scope* is an individual object's set of scope names; therefore, there are as many name scopes as there are individual objects.

Objectivity/DB supports only versioning of basic objects (objects of class *ooObj* or user-defined classes derived from *ooObj*); versioning of "nested" or composite objects is unsupported. In order to create and work with versions of collection of objects, one can provide the appropriate code in his class which derives from *ooGeneObj.*

5.2 Object Versioning in O_2

The following information was obtained from a pre-release of the O_2 versioning manual - it was not available in the released product at the time of writing.

For versioning purposes O_2 defines a ***configuration*** of objects, which is a collection of derivatives from *o2_Version* class - class used for maintaining versions. Any kind of object can be put into configuration, including objects which are collections any versionable object can be referenced to or from non versioned objects. The only restriction is, that collection involved in versioning can not be indexed. In general it means that O_2 allows to create and manipulate versions of **groups** of objects, regardless if the group consist of the same class of objects or not. When you create a new version of an object, which consists of other objects (lets call them *subobjects*), all *subobjects* will appear in a new version automatically.



Unlike Objectivity/DB, first version called *root version* is automatically sets as a default version. O_2 provides a lot of pre-defined functions, which simplify the development of applications using versions; the same can be achieved in Objectivity/DB by writing additional users routines. Of particular note is that each version can be easily labeled. Comparing to Objectivity/DB, there are some additional function for locating versions, (for example locating root version, retrieving by label). O_2 optimizes the size of a version by not copying the values of the objects in a configuration when a new version is derived. The copy is postponed to the time of updating a new version. O_2 provides also a function for finding difference between two versions.

5.3 Possible Uses of Object Versioning in HEP

There are a number of areas in HEP where one might consider the use of object versioning:

- to handle calibration objects, where multiple versions are expected to exist, typically accessed by time stamp,
- to manage different versions of event data objects, e.g. track objects, perhaps corresponding to reprocessing with improved algorithms and/or calibrations/alignment etc.,
- to handle different versions of persistent event selections.

To investigate the usefulness of Object Versioning in HEP, we have designed and/or developed prototypes corresponding to each of these cases, which are described in more detail below.

5.4 A Calibration Database Using Object Versioning

A possible application for object versioning in HEP is that of a calibration (or *constants*) database - a set of parameters for various elements of the detector are stored in a database, typically retrieved by time instant or run and event number (e.g. give me the pedestals for the electromagnetic calorimeter valid at instant *t*). Typically, many different calibrations for a given subdetector will exist, and there may even be more than one set of calibrations valid for a given time instant.

Such a scenario seems to map well to object versioning, particularly as implemented in Objectivity/DB, which offers both linear and branch versioning, as described above.

Unfortunately, versioning does not offer any clear advantages in terms of access to a specific calibration - there is no obvious meaning to be attached to a *default version* of a calibration, nor do *next* and *previous* provide useful functionality for this application.

Finally, object versioning does not solve the primary problem associated to calibration data, namely efficient access to the most appropriate calibration.

In summary, whereas object versioning *could* be used to manage calibration constants, they offer no clear advantage, and we have therefore not developed a prototype in this area.

5.5 Object Versioning and Event Data

In this prototype, we consider the use of object versioning for handle multiple versions of a track object. We use the word *version* when identity is preserved. Thus, refitting an existing track object results in a new version of that object, whereas re-running the track finding algorithm does not.

An important restriction regarding the use of object versioning is that one must decide a priori which objects are to be versioned and which are not. This is because applications must handle versioned objects explicitly, via the geneology object. Thus, one may decide on specific classes within the event that are to be versioned, such as the HepEvent object, and HepEventTag and so forth.

The use of object versioning would assist the management of various production runs. The default version would be set explicitly, following some quality control, and users would subsequently access this version of the objects in question, unless they explicitly requested an different version, such as the previous one.

5.6 Object Versioning and Persistent Event Selections

The definition of event selections is often an iterative process. An initial selection is made, then refined many times, and often compared with a previous selection. In this context, concepts such as "next" and "previous" would appear to be convenient ways of handling, for example, multiple versions or refinements of a named-event selection.

To investigate how versioning features could be applied to this field, we have built a prototype application. In this prototype, we implement versioning of selections, as well as versioning of the cuts defined by the user. To give the user full freedom, there are no restrictions on the number of cuts, the types[7] used in the cuts or their naming. The user is simply required, prior to running the application, to define the appropriate cuts in a C++ header file, simply by specifying a type and a unique name for every cut. In addition, a "match" function must be provided in a user class, which must be derived from the base class *HepPersistentEventCollectionPredicate*. There are no restrictions on the contents of the match function - it is fully user dependent. This function receives as a parameter an event and should return *false* or *true*, depending if the event is accepted or not, based upon the cuts that have been defined or indeed upon any other criteria. In the constructor of the user class, initial values can be specified. After compiling and linking the user code with the appropriate RD45 class library (which will become a component of CLHEP once fully mature), the user can now start selecting events. Every selection that is created can be given a name unique within a user scope. In other words, there is no requirement that the selections have names that are unique across the entire federated database - multiple users can work simultaneously, each using, for example MySelection, without interfering each other. (It is also possible for multiple users to access the *same* collection simulatenously, e.g. in the case of analysis group or even collaboration-wide named collections.) Also versions of a selection (i.e. predicate genealogy) can be given a name[8]: during the creation of a version or at any time later.

In this prototype, Tcl/Tk has been used to provide the graphical user interface. A window is provided from which event selections can be made, in which the user is able to see the

[7] All standard C++ types can be used.
[8] Also unique within a user scope.

various cuts that were defined in the corresponding header file, and edit the value of these cuts.

While running the program the first time, an object for keeping track of all persistent selections of the user is created (instance of a class *VerManager*). Only one such an object per one user exists inside all the federation. When a new selection is created:

1) it becomes associated with *VerManager* object,
2) a "root" version of selection is automatically created, with a "root" version of the associated predicate.

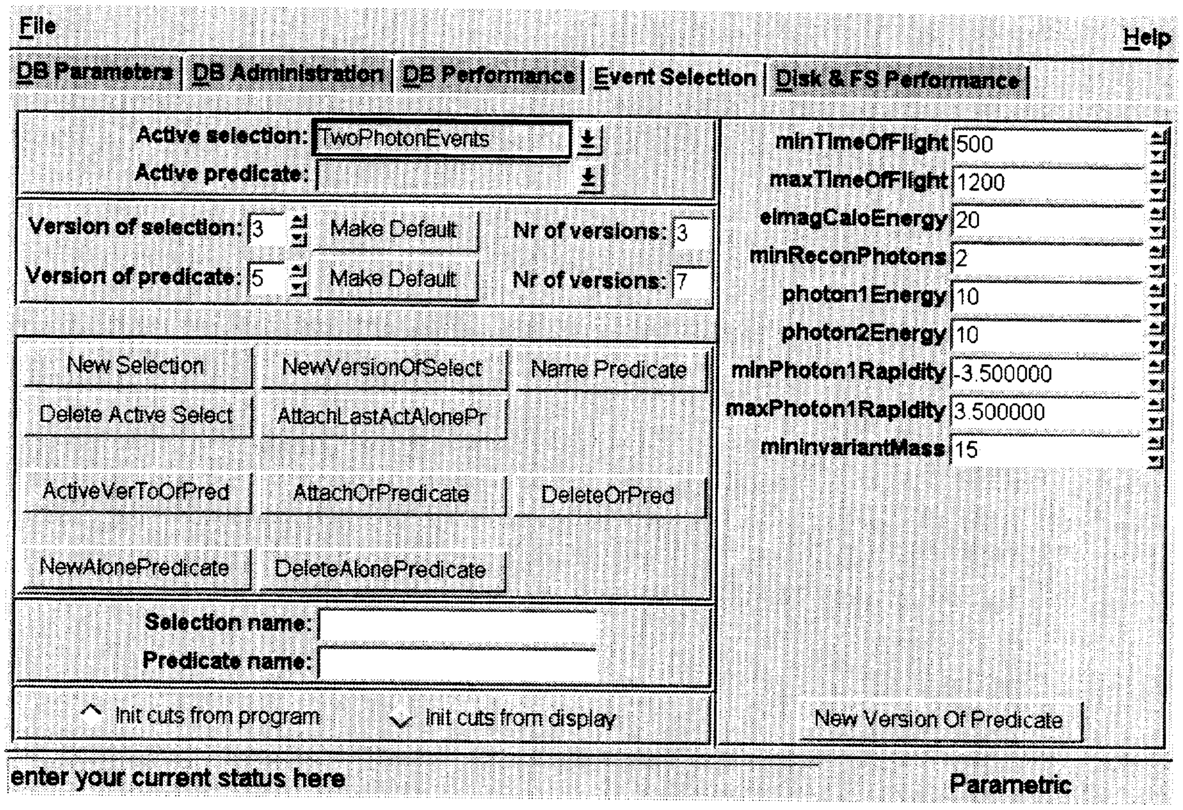
The set of cuts in a predicate can be initialized from the values given by the user in the constructor, or simply from the edit fields provided by the GUI. The user may now create new versions of the predicates associated with a given selection, and also create new versions of the selection itself and again provide multiple versions of the predicates for each version of the selection.

An active version of the selection is always marked as default inside a the selection's *version genealogy*. Similarly, an active version of the predicate is marked as default inside a predicate's *version genealogy*. Unless otherwise specified, by clicking on the appropriate button, the latest version becomes the default one. Event selection is performed using a set of default cuts from a default version of selection.

Keeping many versions of the cuts inside one version of the selection is very efficient and space-saving, as single predicate version is typically very small, consisting only of the values to be used for the cuts in question together with a few "system" fields. In contract, a version of a selection can become very large, as it contains references all events that correspond to the selections.

Inside one named selection user can create up to 2^{32} different versions of selections, and the same number of different versions of cuts inside one version of the selection. Every new version is registered immediately, hence the full history of changes can be retrieved at any time. Obviously one user can see only his selections, without even being aware of existence of others, although once again, public or group selections are also possible.

In the current version of our prototype, versions are located in two different ways: if the neighbour version need to be located, we use the associations "previous" or "next", and for locating any other version, we use "lookup" function, which traverse through all versions until it founds the right one. For large number of versions inside one genealogy, more clever system of decision taking could be probably more efficient. Use of the associations "previous" and "next" could be extended for far-away neighbours, as traversing several times through successive neighbour versions could be faster then looking for one version in the large set of versions. For systems with hundreds of versions, a version could be also located using an index. As an index sorts objects of a particular class according to the value in one or more fields of the class, creating an index which would use versions' numbers could increase greatly accessing versions.



Active selection: TwoPhotonEvents
Active predicate:
Version of selection: 3 Nr of versions: 3
Version of predicate: 5 Nr of versions: 7
Selection name:
Predicate name:

minTimeOfFlight	500
maxTimeOfFlight	1200
elmagCaloEnergy	20
minReconPhotons	2
photon1Energy	10
photon2Energy	10
minPhoton1Rapidity	-3.500000
maxPhoton1Rapidity	3.500000
minInvariantMass	15

enter your current status here
Parametric

5.6.1 *"Alone"* Predicates

The term *alone* predicate is used for a predicate which is not connected to any selection and exists independently. In fact, an alone predicate is a genealogy object, and it can consist of any number of real predicates (versions), with one version default. After creation an alone predicate, possibly with a number of versions attached such a predicate can attached to a chosen selection as one of the versions of the selection. After being attached, there are no differences between a standard version of a selection and attached alone predicate. Every "not connected" alone predicate will disappear at program termination, hence alone predicates can be also considered as temporary predicates.

For convenience reasons, all alone predicates must, in the current prototype, be given a name (unique in the user scope). It would also be possible to manage such predicates simply by keeping track of the appropriate object reference.

5.6.2 OrPredicates

Apart from standard predicates, which consist of a set of cuts defined by the user and a *match* function, predicates of some special types could exist, for further helping in refining events. Firstly, as the most useful, OrPredicate has been implemented. If this shows usefulness of this type of predicates, others (AndPredicate, ...?) will be implemented in a future version of the prototype.
The idea of these predicates is simple: a predicate has no cuts of its own and instead it consists of some associations to other predicates (i.e. versions of predicates, not genealogy

objects). In place of the *match* function defined by the user, a specific *match* function exists; this function defines features of that predicate. In case of OrPredicate, its *match* function accepts an event if any of associated predicates accept this event, otherwise the event is rejected. In case of AndPredicate an event is accepted only if every associated predicate accepts that event.

Creating an OrPredicate can be divided into two steps:
1) creating an object,
2) adding any number of associations between this object and any version of predicate from selections.[9]

After an OrPredicate has been created, it can be attached to any version of a selection (i.e. predicate genealogy), as a version of predicate and then becomes a last and a default version inside the genealogy; it cannot be attached to an alone predicate. As an OrPredicate (unlike standard predicates) has no cuts, in the interface of our prototype for any OrPredicate, in the entry/view fields all values of cuts are set to zero.

5.7 Conclusions

Detailed tests of versioning selections and user cuts proved, that features of versioning can be applied to these field with a great success. Advantages of versioning such as existence of a default version, possibility to see all versions as one object, or easy location of "previous" or "next" version inside a large genealogy fit fine with requirement of applications designed for iterative refining selection. We now believe that object versioning is the best way of maintaining vast number of cuts of hundreds of users. What is not clear now is, if user could have possibility to delete some selected versions: if it is regarded as a history, once created versions probably should be kept for ever as read-only. In current version of our prototype user can delete only entire selection (all versions of selection with all versions of predicates), he cannot delete only a part of selection. If this approach will be continued in the future is not decided yet, but clearly implementation of tools for deleting a part of selection is fairly easy.

We believe that object versioning can be used to advantage in HEP to help manage *persistent selections* of events, and their associated predicates, plus also different versions of event (sub-)objects. Although they could also be used to handle calibration objects, we see no particular advantage (or disadvantage) in this area, as the primary problem is rapid access to the appropriate calibrations, for which object versioning, in this instance, offers no benefit.

We feel that we have identified a need for "user-versions", and have outlined a strategy for how they could be implemented. We have fed this requirement back to Objectivity, and await their response.

[9] As OrPredicate is a kind of predicate, it can be also easily associated with other OrPredicate.

6. Replication[10]

Replication, or more specifically replication of user data[11], is a technique whereby multiple "copies" of individual objects are automatically maintained by the system.

Replication[12] is an important technique that can be used to improve both performance and fault-tolerance. By definition, replicated objects are those objects for which there is more than one *implementation (cf. "copy")*. Typically, the different implementations will be stored on separate servers, often in more than one location. The *degree* of replication may vary within the system - objects that are replicated in all physical instances of a single, logical database are termed *completely replicated*. The set of all replicas of a given object is called the *replication set*.

Replication offers a number of important advantages, particularly in the distributed environment:

- Performance: read access to an object can be satisfied by a local replica, without accessing a remote server,
- Availability: remote users can continue to work with objects, even if the network or remote server becomes available,
- Autonomy: individual sites and/or network segments can continue to work independently.

There are, of course, a number of drawbacks, mainly related to write access. If objects are frequently updated then the amount of synchronization that is required may be significant. In addition, if the objects may be updated from many different locations, a strategy for avoiding conflicts, particularly in the case of network partitioning, is essential.

Thus, replication makes most sense when:

- the data is largely read-only, or the update rate is low compared to the read rate,
- performance and availability are high priorities.

Both of these are true for HEP event data.

[10] See also: Appendix: *User Data Replication in Objectivity/DB* on page 50 and the *Versant Fault Tolerant Server* on page 63.

[11] Replication of *system data*, including the federated database catalogue, schema etc., is also essential. In Objectivity/DB, this is provided by the Fault Tolerant Option (FTO), upon which user data replication (DRO) is layered. We do not discuss the FTO further in this report.

[12] Information concerning replication in ODBMSs is not currently available online. A discussion of the concepts of data replication may be found in **http://www.oracle.com/info/products/symrep/chapter5.html**. See also the discussion in Loomis[14].

6.1 Replication in Other ODBMS Products

At the time of writing, we are not aware of any ODBMS product, other than Objectivity/DB, that offers flexible data. Only Versant appears to offer any sort of "replication", as described in

http://www.versant.com/versant/products/ftserver.html.

and reproduced in the Appendix: *Versant Fault Tolerant(FT) Server* on page 63. The replication provided by the current version of Versant is not considered appropriate for usage in HEP.

6.2 "Tape Replication"

In the normal case, replication is performed over network connections. However, given the quantities of data involved and the uncertainty concerning the network bandwidths that will be available/affordable, the possibility of "tape replication" is also of interest.

Replication by tape may be divided into three stages:

- recording data on tapes from the source databases
- sending the tapes (post,...)
- retrieving data from the tapes onto the destination partitioned databases.

Clearly, for every-day data exchange between replicated parts of a federation this approach is not recommended, but if TB or more of data are involved, this "traditional" method of data import/export could again play a role. For example, if it is decided that a given set of databases that are already populated be replicated to an outside institutes, it may be much more efficient to distribute a few TB of data by tape, rather than attempt to replicate the data over the network.

A shortcoming of *tape-replication* is duration of this operation which, for remote institutes, can take a week or more. In many cases, however, it is not the time delay but rather the reliability of the solution that is the dominant factor. In the case of bulk data import/export, the use of tapes may be more reliable than a network-based solution, even though it does incur a significant management overhead, and typically requires that the database that is being exported remain "offline" during the duration of the export. Once the bulk data has been exported in this way, the databases can be reconfigured as online and minor changes may then take place over the network.

6.3 Use of Replication in the HEP Environment

There are a number of potential uses for user-data replication in the HEP environment. These include:

- performance,
- availability/reliability,
- distribution of data from a central site to regional centres, individual institutes and even end-users,
- "collection" of simulated data from outside institutes to regional centres and/or a central repository.

Prototype applications corresponding to each of these problem domains have been built, and are described further below.

6.4 Replication and Performance

In the case of read-only or read-mostly applications, replication offers potential performance benefits by allowing a read operation to be satisfied by a local replica, thus avoiding costly network transfers, and also by off-loading a central server. Write operations will typically be slower using replication, as the transaction cannot complete until the data involved is written to persistent storage for all currently accessible servers.

Fortunately, HEP data is read-mostly, and thus we expect significant performance benefits, particularly in the wide-area, from this feature. Indeed, it is at least possible, if not likely, that some degree of replication, particularly for objects that are frequently accessed, such as HepTag objects, will be needed simply to satisfy the performance requirements of the users.

6.5 Replication and Reliability

In addition to potential performance improvements, replication removes single points of failure and thus allows for highly robust systems. Not only are users protected against host or storage failure, but, more importantly, against network failure. Thus, a user working at a remote site does not require a permanent network connection to a central server, provided that the needed data is replicated locally.

6.6 Distribution of Data by Replication

Data export to remote sites has traditionally been performed by bulk export of tapes. Given the volume of data involved in LHC era experiments, it is not obvious that this solution remains viable, and it is clearly far from being an optimized solution. For example, some estimates call for the export of 100 TB of data to each of 10 regional centres approximately twice per year. This immediately triples the amount of data that is handled each year, and creates significant book-keeping problems.

An alternative solution would be to replicate the frequently accessed data to those sites where it is required - a strength of the Object Database solution is that *any* object can be accessed, regardless of its location in the federation, and thus it is not necessary to duplicate the entire data-set. Preliminary estimates suggest that between 10-100 KB/event will be required to perform most analysis - including even simple event displays! If one is able to combine these reduction factors with appropriate *streaming* or *tagging* of the data, one can potentially achieve reduction factors of 10^4 or more. This is shown in the table below, where the data rate required to replicate 1% of the total event sample, and 1% of each event in question, only 100KB/second are required. Certainly, for small data samples, such as the event tags themselves (which can also be streamed), the required network bandwidth for "real-time" replication is extremely modest.

Data Subset	Data Volume	Data Rate
All data for all events	1MB/event	100MB/second
10% of each event	100KB/event	10MB/second
1% of each event	10KB/event	1MB/second
1% of each event 10% of all events	10KB/event	100KB/second
1% of each event 1% of all events	10KB/event	10KB/second
Event characteristics	100B/event	100B/second

Rates Required to Replicate Various Data Samples

6.7 Collection of Data by Replication

In HEP, event simulation is typically performed in a distributed fashion - simulation is performed at many collaborating institutes and collected at a central site by means of tape (and often redistributed to other remote institutes). The data rates involved are typically extremely low, and data collection could be greatly simplified by replicating the events back to a central site as they are created.

In order to test the feasibility of using data replication for "collection" of simulated data, we have installed Objectivity/DB with the Data Replication Option (DRO) at two remote sites:

- Krakow, in Poland, to test replication over low bandwidth connections (a "worst-case" scenario) and
- KEK, in Japan, to test replication in the "wide-area".

As a fully object-oriented simulation tool-kit is not yet available, we have written collections of objects of various sizes to approximate the structure of simulated events. This "mathematical event generator" currently uses the draft ALICE data model.

This prototype has allowed us to exploit the weighting feature used by Objectivity/DB to determine write access to a given database within the federation. In the default condition, each partition to which a given database is replicated has equal votes. In the case of network partitioning, the partition with the majority of the votes is permitted write access to the database, whereas the partition with the minority is permitted only read access. By applying weights, one can configure the database such that each site involved in the simulation task always has write access to its "own" database. Data will be replicated asynchronously to all other sites as required.[13]

6.8 Conclusions

Data replication is clearly an important capability that will be required for both performance and availability reasons. In addition, its use for data distribution and collection potentially offers greatly superior data management capabilities over today's, ad-hoc, labour-intensive solutions. At the time of writing, data replication support in ODBMS products is still immature, and more tests need to be made over time to understand to what extent these capabilities can be used in a production system in the LHC time-frame. In addition, the real requirements in terms of network bandwidth and resilience to network failure need to be studied.

[13] This is another case where it may be convenient to "batch" updates, by simulating that the network connection is down for most of the time, and enabling replication of pending data at appropriate periods.

7. Summary and Conclusions

We have described the concepts of data replication, object versioning and schema evolution, together with their implementation in two ODBMS products. In each case, we have evaluated how these capabilities could be used to solve data management problems typical of those posed by HEP event data, and have built prototypes to verify the functionality in practice. We believe that all of these features offer advantages for HEP data management, and feel that data replication in particular will bring significant benefits to the problem of data management for LHC-era experiments.

We have identified a number of requirements that are not satisfied by the current implementation in Objectivity/DB, and are working with the vendor to ensure that a future version of the product will satisfy our needs in these areas.

At the time of writing, there are no plans to incorporate these features in a future version of the ODMG standard, although we have requested that they be added to the list of requirements for post-V2.0 releases.

8. Appendices

The following sections are included verbatim from vendor papers.

9. Schema Evolution in Objectivity/DB[14]

Objectivity/DB Version 4 supports high availability applications with a sophisticated schema evolution mechanism that allows developers to modify the structure of persistent data without requiring the database to be taken off-line. This increases the flexibility of the application development process, reducing the technical and business risks associated with modifying deployed applications.

As the requirements of a database application evolve over time, changes are made to the definition of the physical data structure, or *"schema"*, of the data elements stored within the database. Schema evolution is the process of redefining the persistent datatypes in a database application. Data conversion is the step in the process that converts the contents of the database from the old schema to the new schema.

Objectivity/DB provides a mechanism for implementing changes to existing applications with large, populated databases. This ability to modify database schema "on-the-fly" provides developers the option of making data structure changes to deployed applications that would not otherwise be feasible. This reduces the risk of application deployment and makes project management more flexible.

The rest of this chapter describes Objectivity/DB schema evolution in more detail. A general discussion of the restrictions placed on schema evolution by relational technology is followed by a description, including examples, of Objectivity/DB's schema evolution capabilities.

9.1.1 Relational Schema Evolution

Relational technology provides little support for schema evolution and data conversion, offering, at best, the ability to add a statically initialized column to a table. The bulk of the work for more complex schema changes is the conversion of the existing data, which must be performed after the database has been taken off-line.

9.1.1.1 Application Changes

Data is usually stored in a relational database in a normalized format to provide a common view of the data across multiple applications. Persistent data is defined strictly as rows and

[14] http://www.objy.com/ODB/WP/Schema/schema.html.

columns within tables. The translation of data from the logical data structures of the application into the tables that form the schema of the database is left to the application.

As a result, relational database applications must always be conscious of the tables, field definitions, alternate views of tables, and, most importantly, table joins that must be performed during normal execution. Since the database schema is defined by the same mechanism that provides access the database, SQL, every point in the application that accesses the database must be altered to reflect the change in the external data structure.

Of course, it is possible to encapsulate the physical data structure in a relational database application by providing alternate views of tables. The proper use of modular programming techniques can isolate the knowledge of the physical structure of the database. However, object oriented applications, built on object databases, do this encapsulation as a natural part of the development process, rather than as an extra step in the design, programming, and administration of the database.

9.1.2 Data Conversion

Assume for a moment that the application is sufficiently modular to be changed with a relatively small effort. What about the database that already exists that is filled with operational data?

The effort required to convert the data in an existing relational database to a new schema is one that is quite familiar. Traditionally, changing the definition of a table in a relational database requires shutting down the database, converting the contents of the database, updating the applications to use the new table definition, re-distributing the application, restarting the database, and allowing the users back on the system.

Making a copy of some or all of the database and doing the job offline allows the users to remain operational, but it raises the problems of disk space and data integrity. The converted data will be out of sync due to normal operations that occur during the conversion process, requiring some form of update conflict resolution to be performed.

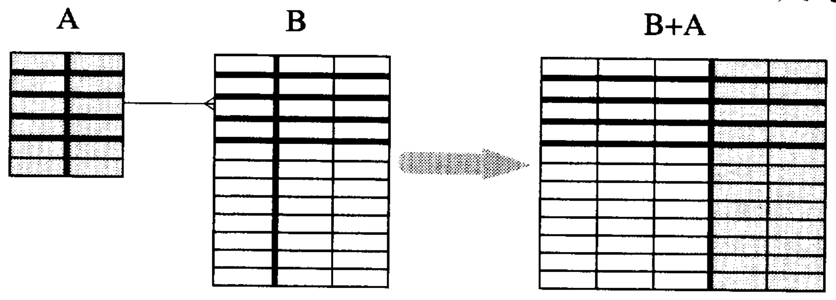
The key issue with changing the schema is that the on-line database must be converted at some point in time. There is no way to avoid inconveniencing the end-users during this phase of the data conversion process.

While the technique of schema evolution and data conversion described above is also applicable to some object databases, it is possible for an object database to provide assistance with the schema evolution process. In particular, the conversion of previously stored data is a process that is facilitated by Objectivity/DB's schema evolution capabilities.

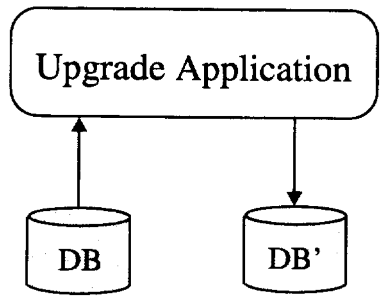
9.1.3 A Relational Example

Consider an example in which the performance of a relational database application is found to be limited by the normalization of the data model.

The administrators discover that one of the original assumptions in their system design is false. Table A was expected to be accessed independently of Table B most of the time, when actual usage indicated that A is only used when B is accessed first. This lookup of A for every B is performed through a join that is repeated over and over, causing extremely poor performance. Denormalizing the physical implementation of the data model, merging A information into each record of B, would improve performance by eliminating the unnecessary join.



RDBMS do not provide support for such schema modifications. In order to make the changes indicated in the example above, the development team must perform some variation of the following general steps:



- Modify the application to use the new schema
- Write a monolithic Upgrade Application that performs three steps:
    1. Reads the old data from the database DB
    2. Converts each record from the old schema to the new schema
    3. Writes the database with the new schema into DB'
- Kick all the users off and shut down the database
- Perform the monolithic data conversion
- Distribute the new version of the application
- Let users run the new application

9.1.4 The Problems

The problems with the monolithic data conversion described above are the lack of database availability during the data conversion process, excess disk space requirements, and general risk involved.

Availability

User inconvenience can be reduced by the steps outlined above, but it cannot be completely removed. The data conversion takes a finite amount of time. The larger the database, the longer it is unavailable during data conversion.

This puts a great deal of pressure on the development staff to make the conversion process go smoothly. It also requires access to the database when it is not being heavily used so that it can be shut down without disrupting operations. This is not possible for many applications, since they require the database to be available continuously.

Disk Space

During data conversion itself, the data is copied from one place to another, meaning that there will be two images of the database in storage. This could, in the worst case, double the disk space requirements.

If the entire database was copied during the conversion program, the disk requirements would double. If the tables are copied back into the same database, then obsolete versions of tables may exist that can be archived or deleted, as appropriate.

Disk space is a particular issue for schema evolution in object databases, since object databases are able to hold significantly more operationally useful data than relational databases.

Risk

Monolithic data conversion incurs tremendous risk to the schema evolution process in terms of the lost business opportunities during the data conversion time period. The business costs associated with the database being unavailable are entirely application dependent, but can be considerable in strategic applications.

The primary technical risk involves data integrity. At some point, the users will all change over to a new version of the end-user application to run against a new version of the database that was created with a separate upgrade application. The possibility of corrupting the database with two applications is greater than with a single application.

9.2 Schema Evolution with Objectivity/DB

Objectivity/DB provides a robust schema evolution mechanism that handles most schema changes quite simply, giving the developer control over the timing and the granularity of the data conversion process. A developers is able to alter the schema of a deployed application and convert the existing database without forcing end-users off the database during a lengthy off-line, monolithic data conversion process.

Rather than have to convert the entire database at once, only the objects whose definitions have changed are candidates for data conversion. Those objects are referred to as "***affected***" objects. They may be converted one at a time, or in various size groups. When converted, affected objects are written back into the space in which they existed before, allocating or freeing incremental disk space according to the type of change being made to the schema.

During the data conversion process, and in stark contrast to relational databases, the database remains on-line for the business function it supports, minimizing business risk. The technical risk is also minimized because in most cases the "conversion program" and the "end-user application" are the same. Data is converted automatically by Objectivity/DB in the end-user application, which greatly reduces the risk of programming errors.

The remainder of this discussion revolves around the type of schema changes that can be made, and how the timing and granularity of the data conversion is controlled by the developer.

Types of Schema Change

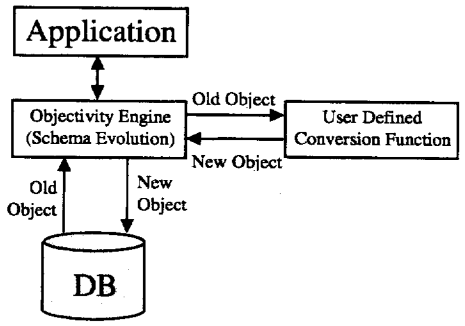
Many schema changes are possible, ranging from purely logical changes (such as changing the name of a data member) to inheritance changes. The basic types of schema changes supported by Objectivity/DB are:

- Logical changes
- Class member changes
- Association and reference changes
- Class changes
- Inheritance changes

Basic schema changes of each of these types can be handled automatically by Objectivity/DB, with the optional use of Conversion Functions as required.

Automatic Conversion

Objectivity/DB handles many types of schema changes automatically, such as the conversion of one primitive datatype to another, the addition or deletion of new class members, and the modification of the access control of a base class.


Application
Objectivity Engine (Schema Evolution)
Old Object
New Object
User Defined Conversion Function
Old Object
New Object
DB

Conversion Functions

Conversion Functions are developer defined call-back functions that provide an opportunity for application dependent processing to be applied at the point of data conversion. The Conversion Function is executed by the database engine during the automatic conversion of affected objects. Each time an object of the old schema is accessed, the Conversion Function is executed. When objects of the new schema are accessed, the Conversion Function is not executed.

9.2.1 Conversion Modes

After the type of schema change has been specified, the issues of timing and granularity of data conversion must be addressed. In other words, we have to decide when to convert the existing data, and how much of it to convert at a time.

Relational databases, and some object databases, only provide monolithic data conversion. In object database terms this is called "Immediate Mode" conversion, because all the data has to be converted immediately before any user application can be given access to the database. This makes the database completely unavailable to the users. By comparison, Objectivity/DB does not limit access to the database during data conversion.

In addition to Immediate Mode, Objectivity/DB also offers alternative conversion modes that allow the application requirements to dictate the timing and granularity of data conversion. Data conversion can either be deferred until objects are physically accessed by an application, or performed when the developer demands. These are known as Deferred and On-Demand schema conversion, respectively, which never limit database availability. Even Immediate Mode data conversion leaves the database available, because only the affected objects are made unavailable.

Mode	Granularity
Deferred	Object
On-Demand	Container
	Database
	Federated Database
Immediate	Federated Database

Deferred Mode Conversion

Deferred Mode Conversion leaves the affected objects in the database in the old form until they are required for use by the end-user application. Objectivity/DB converts each affected object as it is used in the course of normal end-user operations.

Deferred Mode, which encompasses the majority of schema changes, is the easiest form of conversion from the developer's standpoint: the end-user application is simply modified to use the new schema. The process of changing the schema in the application source code will automatically set the program up to convert affected objects as they are encountered; i.e. in Deferred Mode. If a Conversion Function is required to augment the data conversion, it would be added to the end-user application.

The end-user simply receives a new version of the application and operates it as before. The conversion takes place in the database engine automatically. There is no need to stop all the users from using the system for an extended period of time, because down-time for an individual end-user is limited to the amount of time it takes them to restart their application.

On-Demand Mode Conversion

On-Demand Mode Conversion does the same type of conversions as Deferred Mode, defined above, but to groups of objects explicitly indicated by the application developer at various points in an application.

On-Demand Mode is implemented by calling a member function for one of the data storage constructs in the end-user application. The function call would be placed at that point where the application encounters new containers, databases, or federated databases. Objects, and groups of objects, are flagged as they are converted, so that each affected object is only converted once. Unless On-Demand conversion is used for the entire federation, it is likely that there will be some unconverted objects in the database. This is not a problem, since they will simply be converted when the end-user application tries to use them.

Immediate Mode Conversion

The use of Immediate Mode data conversion allows schema evolution to be performed despite the presence of uni-directional associations and inherited references in the schema. Objectivity/DB offers a flexible implementation of Immediate Mode conversion that leaves the database on-line, making only the affected objects unavailable during data conversion. Note that Immediate Mode conversion is only required for two specific types of schema change; replacing base classes and deleting classes.

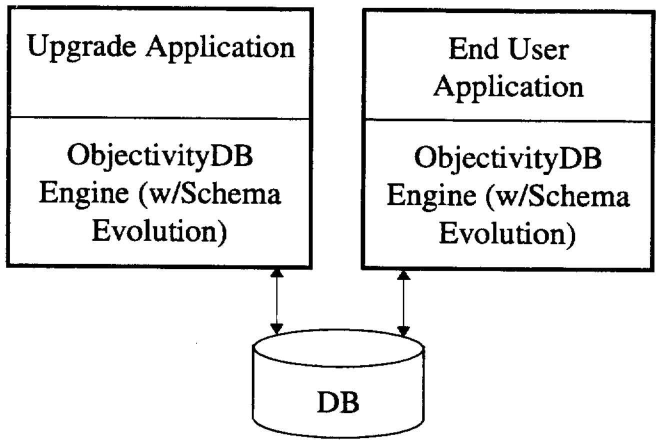
9.2.2 Other Schema Evolution Issues

Multiple Changes Over Time

Objectivity/DB can keep track of an arbitrary number of Deferred Mode schema changes to the same class. This becomes an important issue when multiple changes are made to the schema over time, and not all of the objects in the database have been converted.

For example, assume that a particular class is changed two or three times using Deferred Mode data conversion. The database will contain both converted and un-converted objects mid-way through a Deferred Mode conversion. Objectivity/DB allows the subsequent schema evolution processes to be started, even though all the data has not yet been converted from the earlier schema changes. As objects are accessed, they will be converted to the newest schema automatically.

Upgrade Applications


Upgrade Application
ObjectivityDB Engine (w/Schema Evolution)
End User Application
ObjectivityDB Engine (w/Schema Evolution)
DB

Upgrade Applications are primarily small programs that make one or more calls to the On-Demand Mode member functions to convert objects in containers, databases, or across the entire federation. Such an Upgrade Application can usually be run in parallel with the new version of the end-user application, with the knowledge that when it completes, all affected objects will have been converted.

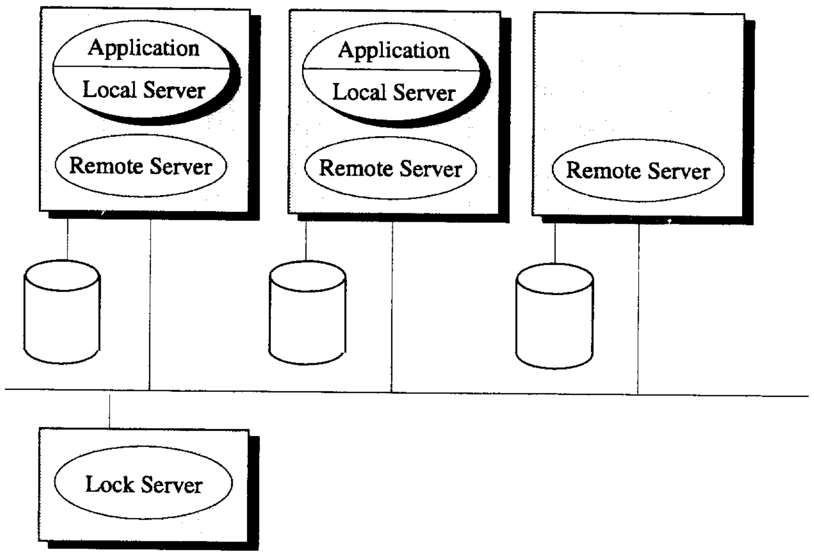
Of course, it is not possible to anticipate every type of schema change. Objectivity/DB schema evolution supports unanticipated schema changes through the sequential execution of multiple schema changes. Some of these multiple step schema changes will require an Upgrade Application to explicitly traverse all of the affected objects prior to moving on to the next step in the schema evolution.

In a similar fashion, schema changes that are complex in nature, such as those where Immediate Mode conversion is required, are dependent upon application-specific information to be provided in an Upgrade Application in order to be able to apply integrity constraints during the schema evolution process.

9.2.3 Schema Evolution Scenarios

Objectivity/DB In Centralized Client/Server Applications

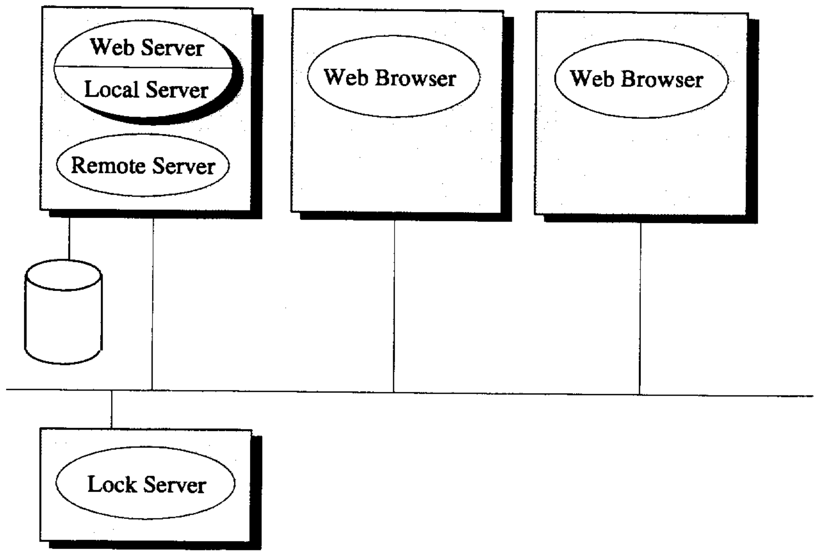
Take the example of a repository built using Objectivity/DB, where the end-users start and stop the client application each day. In this scenario, the client applications are able to be re-distributed as a normal part of operations. In an application in which Objectivity/DB resides in the client workstations, performing schema evolution simply requires updating the client workstation applications.



The only time that the database would be "unavailable" is during the brief moment when the client applications are being restarted.

Objectivity/DB In Server Application Only

Removing Objectivity/DB from the client workstations changes the situation.



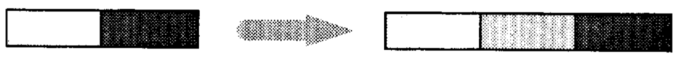
This might be an advanced Web server application built with Objectivity/DB, where the "client application" is an off-the-shelf Web browser. Since the client and server processes are effectively decoupled through the use of HTML, it is unnecessary to re-distribute the client portion of the application. The only time that the Web site would be unavailable is during the restart of the Web server application.
One way to prevent even this minor interruption of service is with Objectivity/DB Data Replication Option, which allows an individual server to be taken off-line for service, and brought back on-line again, without disrupting access to replicated data in a federated database.

Schema Evolution Examples

Adding Data Members

This example is the classic situation where a new piece of information needs to be maintained in an object.

Before		**After**
long FirstItem	→	long FirstItem
long LastItem		long NewItem
		long LastItem



The steps to performing the schema evolution are quite simple.

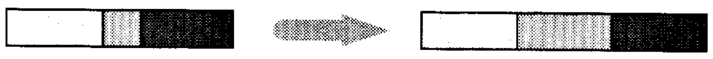
- Change the schema and application to add the new data type.
- Re-compile, re-distribute, and run the application as before.

The conversion of objects residing in the database will be deferred until they are accessed in the normal operation of the application. Objectivity/DB can automatically initialize a new data member to a predefined value. If the initial value must be calculated, a Conversion Function is required.

Conversion of Primitive Datatypes

In this example, the number of unique BufferIDs required was under-estimated. Converting BufferID from a short to a long will solve the problem. The physical conversion of the object is shown below.

Before	**After**
long FirstItem	long FirstItem
short BufferID	long BufferID
long LastItem	long LastItem



The steps to performing the schema evolution are quite simple.

- Change the schema and application to use the new data type.
- Re-compile, re-distribute, and run the application as before.

Logical Schema Change

In this example, no change to the physical structure of the persistent objects is required. An object has in-line references to other objects. New requirements dictate that when the object is locked or deleted, all the referenced objects should also be locked or deleted. Adding lock and delete propagation to the in-line references requires a logical schema change to the schema that can be implemented in Deferred Mode.

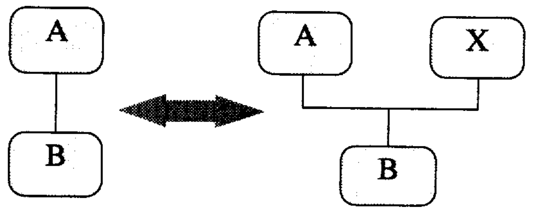
The steps to performing such an operation are as follows:

- Change the schema to propagate locks and delete properties across the in-line references.
- Re-compile, re-distribute, and run the application as before.

Modifying Inheritance

Objectivity/DB's schema evolution support is not limited to modifying the contents of a class. It is also possible to modify the inheritance relationships between existing classes in Objectivity/DB.

For example, adding a non-persistent base class to a persistent class is a schema change that can be implemented in Deferred Mode. The same is true for removing a non-persistent base class. In this example, we also wish to ensure that all the objects are converted in a finite amount of time.



The steps are the same as in the earlier examples:

- Change the end-user application to add or delete the base class.
- Re-compile, re-distribute, and run the user application as before.

Over time, most of the affected objects are likely to be converted. In order to force the remaining affected objects to be converted, an Upgrade Application can be written that calls the function to convert the remaining affected objects in the federated database. If Conversion Functions are used in the end-user application, they should also be used in the Upgrade Application.

- Create an Upgrade Application that also converts the objects in On-Demand mode against the entire Federated Database.
- Run the Upgrade Application to convert the affected objects simultaneously with the execution of the end-user applications.

9.2.4 Conclusion

Schema evolution is a key requirement in high availability applications. Objectivity/DB provides powerful and flexible schema evolution capabilities which clearly demonstrate our support of mission-critical application environments in which the database must remain available at all times.

Not only are application developers able to make schema changes that were not possible before, but they are able to do it easily with Objectivity/DB. The flexibility of Deferred and On-Demand Mode data conversion allows the developer to select the timing and granularity of the data conversion appropriate for the application.

Objectivity/DB's support of on-line data conversion minimizes the risk traditionally associated with making schema changes in deployed applications. Application developers are better able to plan incremental application modifications, reducing the risk of being locked into a deployed application that might be inadequate to meet future needs.

10. User Data Replication in Objectivity/DB[15]

Objectivity/DB Data Replication Option (DRO) extends Objectivity/DB Version 4 to include database replication capabilities that provide improved read performance and continuous availability, while ensuring data integrity in a manner that is transparent to applications.

Objectivity/DRO provides a method of data replication by which data integrity is maintained not by application specific code, but by the database engine itself. By comparison, traditional database solutions that offer asynchronous replication require data integrity to be maintained by the application, rather than the database.

The fundamental problem in maintaining data integrity in a set of replicated database images is propagating the changes from one database image to each of the others. With Objectivity/DRO, this is accomplished using a highly efficient voting mechanism to determine whether a particular access can successfully be granted to the database. If a majority, or "quorum", of database images agree, then access will be granted. If one or more of the database images are unavailable due to network or server failure, a quorum of database images is still sufficient to successfully modify the database. When the failed servers are repaired, or reconnected to the network, the database images that they contain will be automatically resynchronized to match the contents of the quorum database image.

Data replication with Objectivity/DRO provides continuous system availability in the event of either server or network failures. Since database images can be distributed geographically, Objectivity/DRO protects against physical and catastrophic site failures, as well as hardware and software failures at a given location.

The rest of this document describes Objectivity/DRO in more depth. Discussion will center around the various methods of data replication and present a detailed description of Objectivity/DRO. A number of data replication examples will be presented to demonstrate how to configure Objectivity/DRO in a variety of situations.

The fundamental issue in data replication is maintaining data integrity across multiple database images. Two primary solutions exist, synchronous and asynchronous replication.

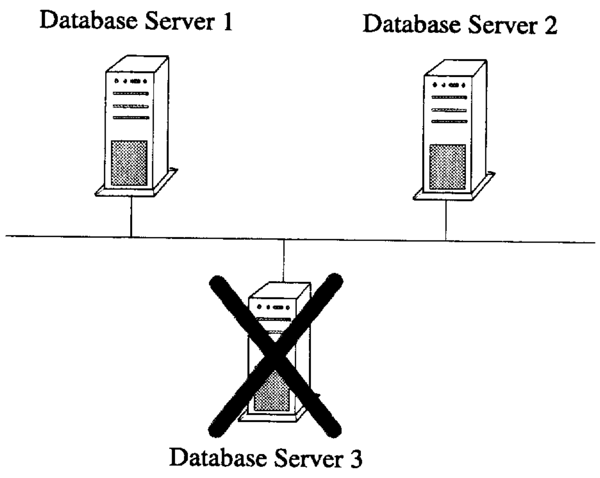
10.1 Synchronous Replication

Synchronous replication requires that every image of the database be written at once. Data integrity is maintained in traditional database solutions with a two-phase commit within a transaction that accesses every image of the database element to be written, and proceeds only when every image is available for update. In traditional database solutions,

[15] http://www.objy.com/ODB/WP/DRO/dro.html

synchronous replication eliminates the distinction between master/slave and peer-to-peer configurations.

However, traditional synchronous solutions are vulnerable to system failure. If one image of the database is unavailable due to server failure, the transaction is prevented from completing.



While synchronous replication assures data integrity, it does so at the expense of availability. System availability is greatly reduced with synchronous replication if the links between the database images are fragile. This vulnerability to network failures also prevents synchronously replicated database servers from being geographically distributed, leaving them vulnerable to location specific disasters, like fire or earthquake.

10.2 Asynchronous Replication

Asynchronous data replication provides two benefits, improved read performance and continuous availability. The failure of a particular server does not generally affect the operation of the other servers, but does force more work to be done in the application to maintain data integrity.

When changes are made, a single image of the database is updated, and then the changes are propagated to the remaining images. The two popular mechanisms for propagating changes to the other database images are imbedded triggers and passing change logs. Triggers are implemented inside the database and add to the overhead associated with performing transactions in the database. Passing change logs also increases network overhead during the replication of the update, but is less intrusive on the local database.

Asynchronous replication also allows database images to be isolated geographically, since each site has its own version of the data.

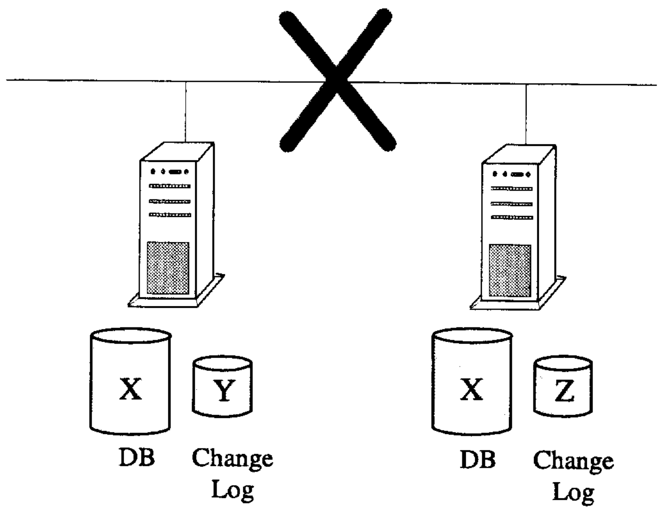
10.3 Conflict Avoidance, Detection, and Resolution

Conflict resolution is required when two images of the same database are updated at the same time, without knowledge of the other image. The simplest solution for this type of conflict is to have the application avoid such conflicts by ensuring that each data element belongs to one database image or the other. This forces a master/slave relationship between database images.

Once the conflict exists, traditional databases provide mechanisms to detect and resolve them. Conflict resolution algorithms assign priorities to the conflicting updates based on update requester status (master/slave), timestamps, or some type of application dependent algorithm. However, all applications do not fit these models. Certainly, a last-writer-wins model can cause problems for the second-to-last-writer. A master/slave relationship does not allow for a true peer-to-peer situation, where multiple servers have equal ability to update the same data.

Traditional Hot Fail-Over Solutions

Take the example shown below with two images of the same database on different database servers, separated by a failed network.



In this scenario, each database server thinks that it is the only one that exists, and that it has the ability to modify the database. Each server continues to process changes, creating the appropriate change log entries, not realizing that the failed server will come back on-line. When the connection is re-established, each will try to update the other.

The problem occurs when both have made changes to the same data element. In this case, both have updated element X, creating Y and Z, creating a conflict that may or may not be able to be resolved. Conflict resolution in this case is entirely application dependent.

10.4 Summary of Traditional Replication Solutions

Synchronous and asynchronous replication are summarized in the table below. With synchronous replication, data integrity is assured, but availability is not maintained. With asynchronous replication, availability is higher, but data integrity is not maintained.

Traditional database solutions have struggled to provide data replication, because their fundamental architecture assumes a centralized server that is designed to operate in isolation. Replication and distribution are then provided by connecting multiples of these stand alone servers.

The reason that traditional database solutions offer both synchronous and asynchronous solutions for the data replication problem is that they are unable to provide data integrity and continuous availability at the same time. They are able to provide one or the other, but not both.

In order to provide both availability and data integrity, a third alternative is required that provides a fundamental improvement to synchronous replication. Objectivity/DRO is an implementation of this third category: synchronous data replication with a dynamic quorum calculation mechanism that provides both availability and data integrity.

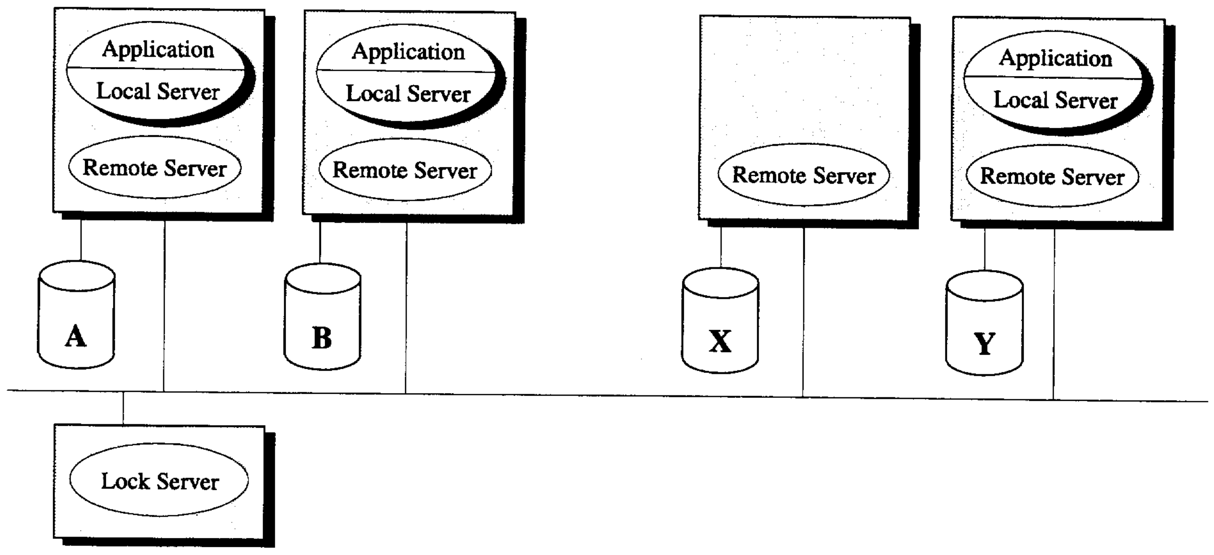
10.5 Objectivity/DB Data Replication Option

Objectivity/DB has been designed from the beginning to be a distributed database, efficiently maintaining data integrity across multiple database servers. Objectivity/DRO extends this distributed functionality to provide data replication. Before we can describe the technical aspects of Objectivity/DRO, it is necessary to review the key concepts of Objectivity/DB and Objectivity/FTO.

Review of Objectivity/DB Key Concepts

Objectivity/DB is built around the concept of the "federated" database, which is defined to be a set of individual databases that reside on a number of servers.

The architecture features Local Servers, Remote Servers, and Lock Servers. The Local Server is linked to the applications, and provides access to the local disk, while Remote Servers provide access to disks on other machines. The Lock Server manages concurrency among users. Caching is maintained on the Local Server to provide improved performance through efficient use of disk I/O. Cross-transaction caching is managed by the database.



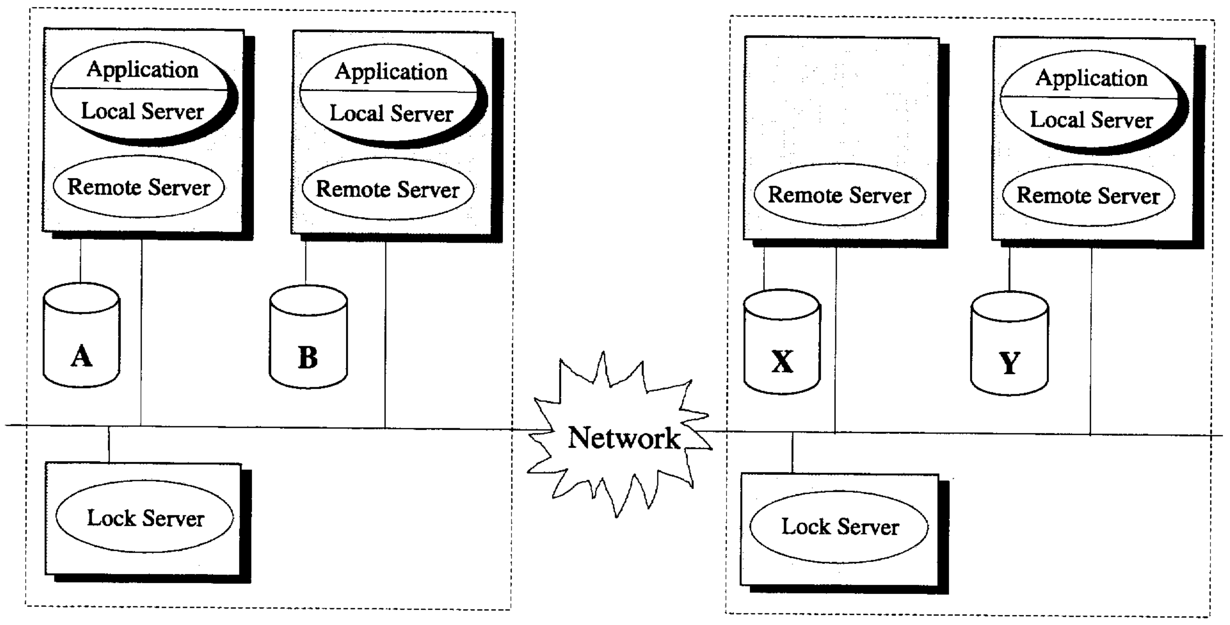
Objectivity/DB has the ability to distribute the databases across numerous servers, as shown above.

Distributing databases across multiple servers provides tremendous scalability, since there is no single bottleneck for data access. The lock server process is not a bottleneck, since it uses IDs to determine locking, and does not require accessing the object itself.

Review of Autonomous Partitions

Autonomous partitions operate as independent groups of databases within the context of a single federated database. Each autonomous partition has its own lock server process controlling access to its data. Autonomous partitions are provided by Objectivity/DB Fault Tolerant Option (FTO), which is a required option to implement data replication with Objectivity/DRO.

Objectivity/FTO allows Lock Servers to be replicated for each autonomous partition. Catalog information is replicated across autonomous partitions to provide schema definitions when partitions are isolated by network failures.



Placing a lock server in each autonomous partition protects one from the failure of another. Even if an autonomous partition fails, the others can continue to operate among themselves on their own data.

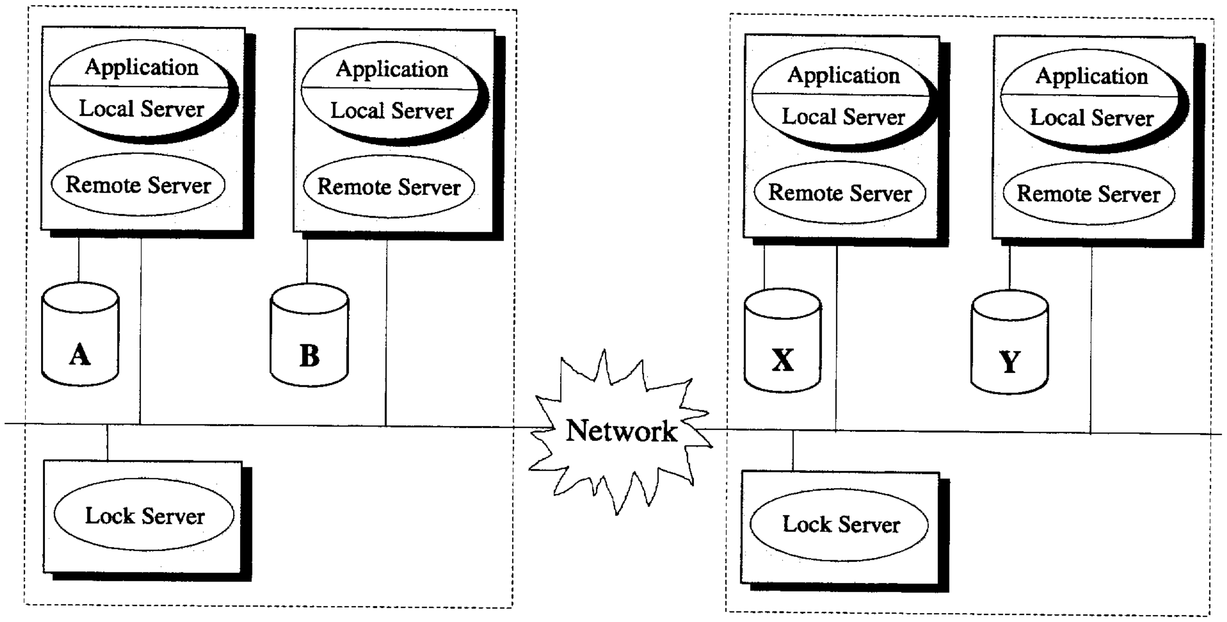
It is important to note, however, there is only one copy of any particular database. With Objectivity/FTO, if one autonomous partition fails, access to that data is interrupted until the autonomous partition is repaired or brought back on line.

Data Replication

Objectivity/DRO provides multiple images of a database, placing each one in a separate autonomous partition. The application requirements determine which databases are to be replicated, and how the database images will be configured.

Objectivity/DRO allows flexible replication of selected data. Since the definition of any individual database is determined by the application developer, and can be dynamically altered, the granularity of the data replication can be selected to meet the requirements of the application.

Updates to data elements are synchronous, so there is no concern of data integrity violations. When an image of a database is brought back on line, it will be automatically resynchronized with any updates that occurred to the other images while it was off-line.



Note that only one image of each database can reside in any autonomous partition. Adding Objectivity/DRO makes it possible to have continuous access to the data.

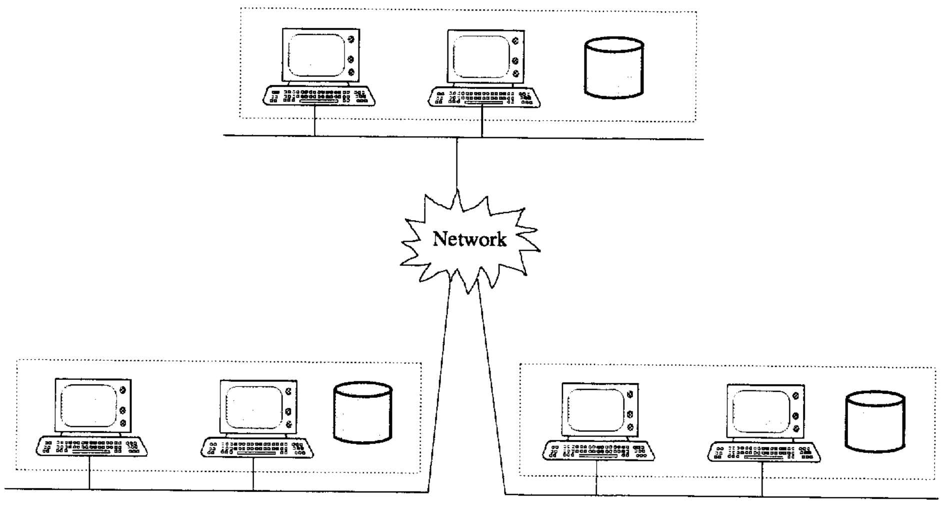
Creating a sufficient number of database images protects against having a single point of failure. Even multiple failures will not prevent access to the data, as long as enough database images exist.

Quorum Calculation

The number of database images required to complete a transaction is calculated implicitly at runtime. This is called the "quorum calculation". Simply put, the database images vote and the majority wins.

When a database image is accessed, and a lock is implicitly requested as part of that access, the application process running on the local computer contacts each lock server to determine which database images are available to vote. If a majority of the images of a database is available, called a "quorum", the access is permitted.

In this example, any two of the three database images form a quorum.



The images that are not available, for whatever reason, will be automatically resynchronized when they come back on line.

The quorum calculation uses a minimum of network traffic overhead, since only identification information is being passed. The data itself is not required to be passed, saving the network bandwidth for transmission of the data directly from database server to database server during normal operation of the application.

There is a trade-off to be made in designing an application with Objectivity/DRO, since write performance depends partly upon the number of images of a given database that is being updated. On one hand, the more images in the quorum, the longer it may take to modify the database. On the other hand, the more images of a given database there are, the more assurance there is that a quorum will be available.

Non-Quorum Access

The quorum calculation occurs on every inquiry and update transaction. A quorum is not required for inquiry transactions. The requirements of the particular application determine whether data is accessible for read access when a quorum is not available. While the system default does not allow non-quorum reads, it may be changed programmatically.

A quorum of images is required to successfully access the database for update. The quorum calculation ensures that enough database images are available to represent the true state of the database. Objectivity/DRO prevents non-quorum writes from ever occurring to ensure the integrity of the data, and prevents the need for merging multiple versions of a database together.

The bottom line is that if the application can write the data, there must be a quorum of database images available to ensure the availability of the modified data in the future.

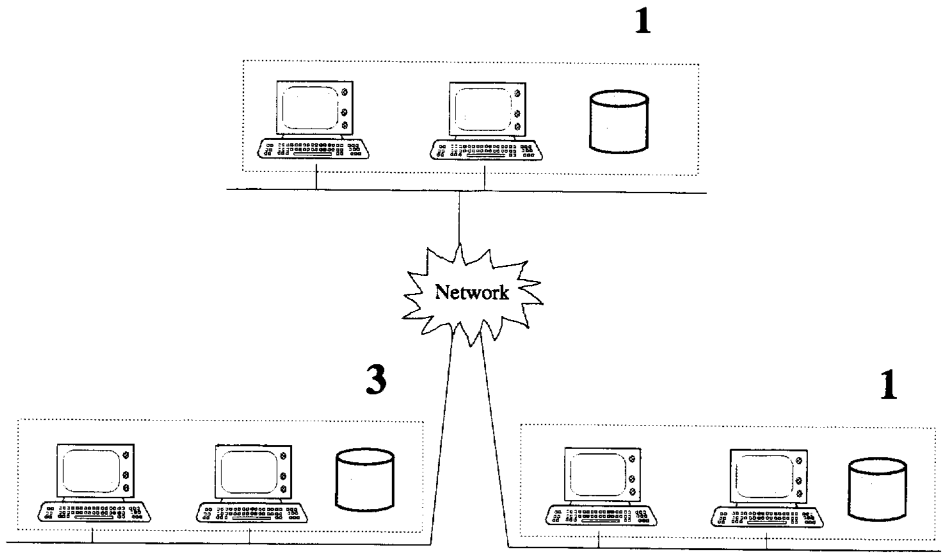
Non-Unitary Weighting

There are many replication scenarios in which some database images require special access or ownership of particular data, regardless of server or network failures elsewhere in the system. Objectivity/DRO addresses this by assigning voting 'weights' to each database image.

Previous examples have been simplified to imply a single vote per database image. The actual case is that, rather than merely counting the number of database images available, the quorum calculation compares the weight of the available database images to the total weight assigned to all the images of that same database.

An example of this is a master/slave configuration in which one particular database image must be available for update transactions to succeed. The other database images are used to provide improved read access performance.

Setting the weight of the first database image to be greater than half the number of total weights assigned has the effect of making the first database image the owner of the database.



This is a master/slave configuration, since it only allows the slave database images to be updated if the master database image is also available to provide a quorum.

Administration and Maintenance

Data replication with Objectivity/DRO is easy to maintain and administrate. There are no complicated database resynchronization procedures for out-of-date database images since Objectivity/DRO automatically re-synchronizes them when they come back on-line.

This capability allows easy platform maintenance. For example, in order to update the operating system on a computer, simply take it off-line, install the new version of the operating system, and put the computer back on line. Objectivity/DRO will automatically re-sync the database image, and the end-users will not be aware that maintenance was being performed.

The maintenance of multiple images of a database, including the assignments of weights, is primarily an administrative activity. Applications developed with Objectivity/DRO are not required to be aware that they will have their database replicated. This allows control of the application configuration to be maintained outside of the application development team.

Non-Quorum Recovery

No software can protect against every kind of failure. There may still be a situation in which a catastrophic event wipes out a quorum of the database images.

Take the example of a master/slave configuration in which all the master database images are located within a geographic region that is destroyed in an earthquake. All that remains after the earthquake is a non-quorum group of database images in some remote location. If the remaining, non-quorum database images were part of the quorum at the time of the earthquake, then the data in them is current. The data is inaccessible, however, since the remaining database images do not form a quorum. The database administrator will have to use Objectivity/DRO's administrative utilities to redefine the quorum to be within the remaining database images. Effectively, the administrator makes the federated database think that the database images destroyed in the earthquake never existed.

10.6 Business Scenarios

Objectivity/DB does not distinguish between database images on "client" workstations and database "servers". There is an inherent peer-to-peer relationship that can be modified to fulfill the requirements of the application. The database administrator creates a master/slave or peer-to-peer relationship with Objectivity/DRO by assigning the appropriate weights to the database images.

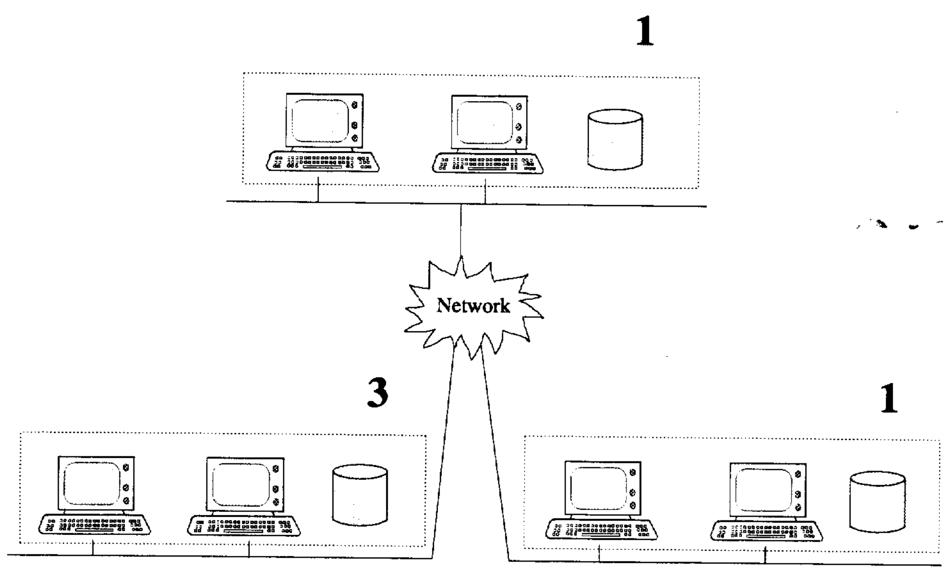
Telephone Call Routing

Protection Against Server Failure

Consider the conflict resolution example described earlier and apply it to telephone call routing; a situation that requires 100% availability of network configuration information. This application requires a hot backup of its data so that no interruption of service occurs.

In order to protect against the database server crashing, an image of the database is placed on a second server. If the first server fails, the second server continues processing normally.

The situation is pictured as follows:



In this configuration, two different servers contain images of the same information. If one of the servers fails, database reads are still possible through the other server. (Note that autonomous partitions may span servers, LANs, and WANs. For the sake of simplicity, this example assumes that each server contains its own autonomous partition.)

Simple Hot Fail-Over With Objectivity/DRO

Objectivity/DRO does not assign primary or secondary status to either of the database images. All images are equivalent representations of the same database. The only differentiator between images is the weight assigned to each which is used in calculating the quorum. If access is available to most of the weight of the replicated database images, authority is granted to make a change to the database.

In this example we have chosen to assign equal weight to each database image, which raises the issue of breaking ties. If one of the servers has failed and someone wants to write to the database, there is no quorum available since there is not a majority of weights available. It's a tie. There are as many weights available as there are unavailable.
The way Objectivity/DRO provides a quorum in the basic two image scenario is to create a third autonomous partition. A "tie-breaker" is placed into this third autonomous partition which casts the deciding vote in the quorum calculation. The tie-breaker is considered the equivalent to a database image for purposes of the quorum calculation, but it contains no data.

Any two of the three database "images" constitute a quorum, even though one of the images is really a tie-breaker.

Protection Against Network Failure

Locating each autonomous partition on a different network shows that this capability is not limited to use in single network configurations. Objectivity/DB, combined with Objectivity/DRO, provides a solution that is able to span networks to protect against network failure.

In the previous example, if all of the servers were on a single network, they would be vulnerable to network failures. In most environments, each geographic location has a number of LAN segments to isolate network failures. Objectivity/DRO handles this complexity transparently.

The configurations shown above both protect against network failure as well as server failure. Any two of the three database images will form a quorum allowing the database to be updated, as if they were on a single LAN.

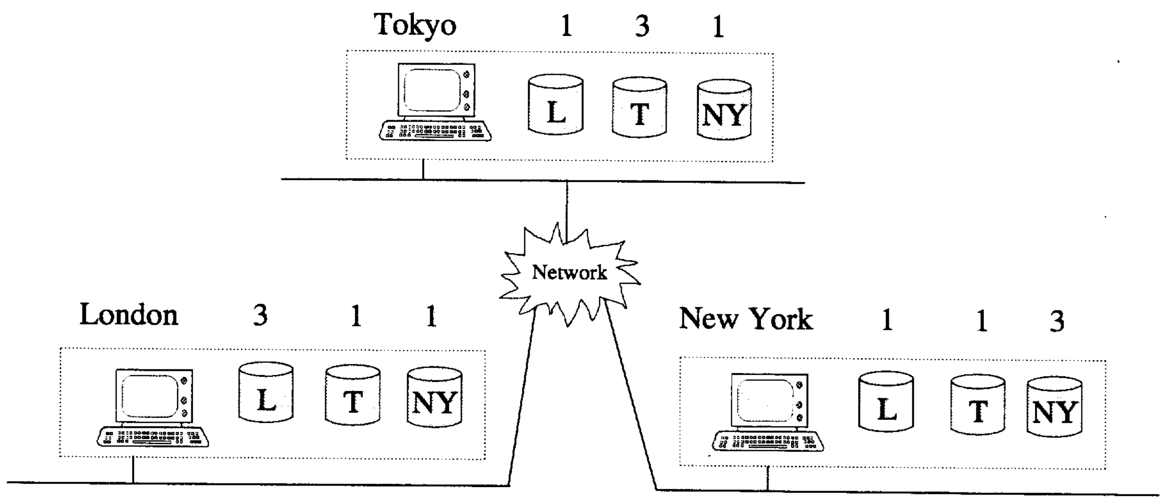
Should one LAN fail, or the internetwork connection to one of the LANs fail, the quorum is still available to client workstations on the remaining LANs.

When the network between servers is disconnected with Objectivity/DRO, data is not corrupted, since only those servers that form a quorum of databases are able to process updates. While the network connection stays broken, only the contiguous servers forming the quorum can write to the database, thus ensuring data integrity.

Afterwards, when the network is reconnected, Objectivity/DRO automatically re-syncs the out-dated database images, eliminating the worries of accessing stale data, or corrupting the database through simultaneous updates.

Airline Customer Profile

Another example is that of an airline providing customer profile information to each counter at airports that are geographically dispersed.



Consider a customer of good standing. A frequent flyer comes to expect a higher level of service from the airline. When dealing with a ticket agent the customer's status will be indicated on the ticket agent's screen.

Clearly, performance is important, since the customer could potentially miss a flight if they had to wait too long, or if their status was not known at a remote location.

For argument's sake, let's assume that there are so many customers that ownership of the customer profile database is divided regionally. (Only New York can update New York customers, etc.) Quorum weighting would be applied as shown above.

With this weighting, each region would effectively own it's data, but replicas would be maintained by the remote locations to address the needs of international travelers. If the internetwork fails, isolating New York from Tokyo and London, New York would still be able read the profile of a European customer flying from London. Updates would not be permitted unless the customer's "home" region was available.

This ability to read "stale" data - called a "non-quorum read" - is an option that may be enabled or disabled, according to the requirements of the application.

This example exposes an application design trade-off true for any database replication implementation in which access to stale ("non-quorum") data is not available for inquiry. When reading stale data is not allowed, data replication is only useful as a means to improve read performance at remote sites.

Flexibility

The primary reason for setting the weights in this manner is to point out the flexibility provided by weighting database images.

If regional ownership is required such that only the "local" region is able to update the customer profile, weights would be assigned as shown. The owner would be given ownership by assigning a weight of 3 to the local image, and a weight of 1 to the remote images.

If regional ownership of customer profiles is not a requirement, a simple change to unitary weights on each database image can alter the behaviour of the system to allow any two of the three regions to update a customer profile.

10.7 Web Server

Many situations require a large volume of data to be updated in a single location that needs to be accessed in geographically separated locations, such as with a Web server.

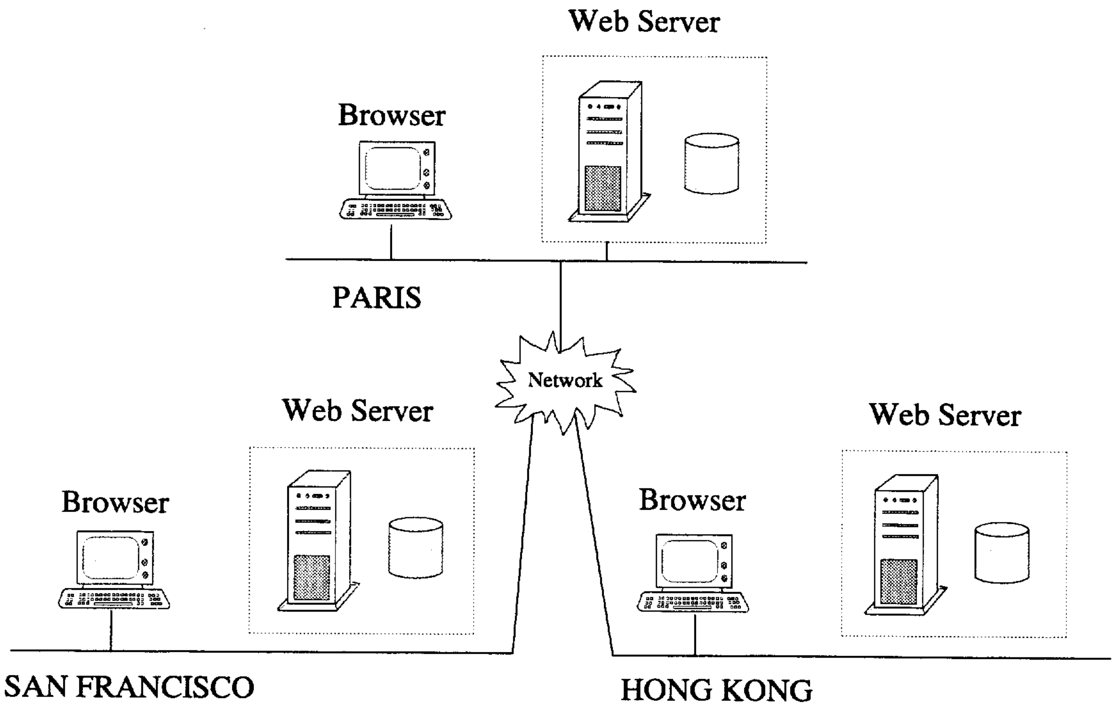
This scenario generalizes to most situations with the following characteristics:

- many reads / few writes

- geographic dispersal of information
- 100% availability required

As an exercise, consider the case of multi-media production facilities at three affiliated news organizations maintaining Web sites for the distribution of current event information. These Web servers would by definition contain a high concentration of audio, full-motion video and bit-mapped graphics.

Some of the information is accessed hundreds of times per day, other information is only accessed occasionally. Though the number of reads is very large in comparison to the number of writes, the data is written on a weekly, or sometimes daily basis.



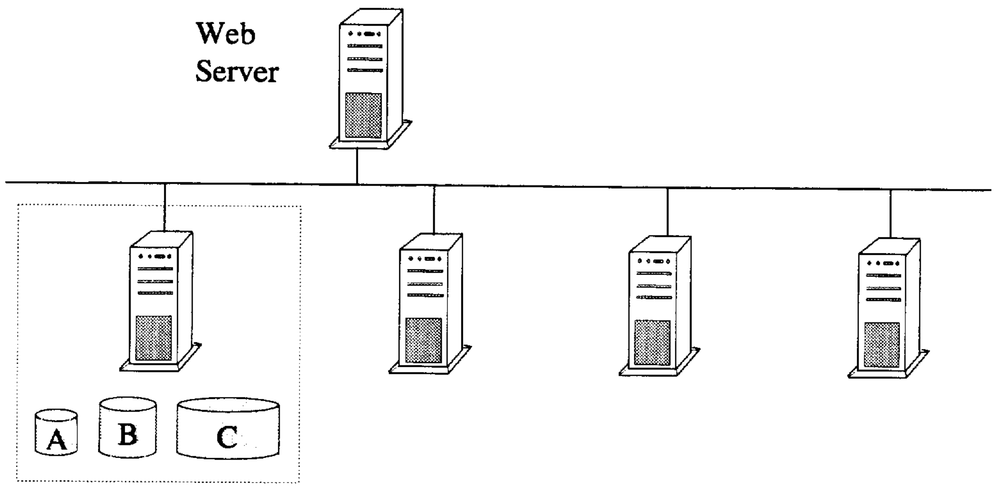
The problem in trying to access all the information from a single server location Hong Kong is that the data volumes are too high to be transmitted on demand through the WAN connecting sites in San Francisco, Paris and Hong Kong. The information still needs to be accessed in Paris and San Francisco, but it is extremely inconvenient if, at runtime, the staff at those facilities has to wait for transmission when reading the database.

The solution is to use Objectivity/DRO to maintain multiple images of the data. The information is locally held to improve read performance. When updates are made to the database, all the locations will be kept up to date with the latest information. If one of the sites is off-line, Objectivity/DRO will perform the automatic re-synchronization as soon as the problem is resolved.

A geographic distribution of database images will improve read performance by moving the data closer to the end-users. Simple reads can be performed against the local image of the database, eliminating transmission delays due to accessing remote Web servers.

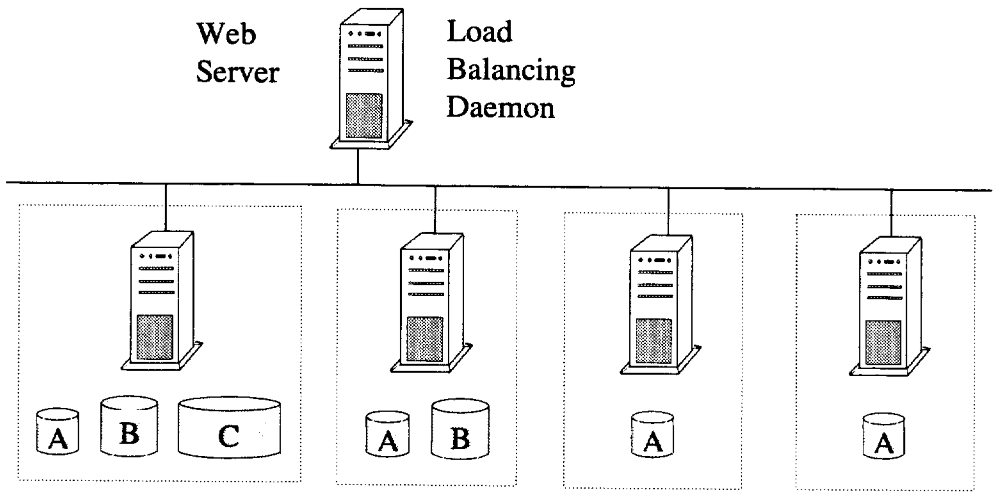
Load Balancing

Note that this application has an uneven distribution of data access. Some of the data is accessed frequently, while other parts of it are read in-frequently.



In this scenario, A represents the hot news item of the day. It is only a small portion of the total volume of the database, yet it is accessed by 90% of the database reads. B represents the background information on A. It is larger than A in terms of size, but is only accessed by 9% of the read traffic. C is only responsible for 1% of the read traffic, but it comprises the vast majority of the database.

Clearly it is desirable to distribute A as widely as possible to provide better response to the majority of the database accesses. B requires some replication, but not as much as A. C, however, can be left on a single server, since it is so infrequently accessed.



You could build a Web server using Objectivity/DRO that is able to dynamically distribute the load across multiple database servers. As Web page access profiles changes over time, the Web server would be able to dynamically re-distribute the data across the database servers according to an application specific load distribution algorithm.

10.8 Conclusion

Objectivity/DB Data Replication Option improves read performance and ensures continuous availability with a dynamic quorum calculation mechanism that maintains data integrity despite server or network failures.

Objectivity/DRO protects against failures in both servers or networks. There is no need for applications to perform conflict resolution because Objectivity/DRO's synchronous data replication ensures data integrity. When database images on servers or networks that have failed become available to the quorum, they will be automatically resynchronized to the quorum database image.

Where traditional data replication solutions force the designer into a particular server configuration, Objectivity/DRO offers the flexibility to select the configuration that best fits the application. Configurations can range from simple hot fail-over systems, to widely distributed systems that offer protection against multiple points of failure. By adjusting the weight of database images, Objectivity/DRO supports master/slave, peer-to-peer and mixed configurations.

11. Versant Fault Tolerant (FT) Server[16]

VERSANT databases are in production use in many application areas that require system availability 24 hours a day, 7 days a week (24 x 7) with no scheduled downtime. At a system level, hardware vendors have addressed this issue with such features as hot standby systems and disk mirroring. VERSANT addresses this need through synchronous replication. Synchronous database replication coordinates duplicate transactions distributed to multiple database servers within the standard VERSANT two-phase commit protocol.

11.1 Continuous Normal Operation

The VERSANT FT Server establishes a replicant pair and coordinates identical transactions to the pair of VERSANT databases. Database clients connecting to a replicated database are automatically and transparently connected to the replicated pair. Update transactions are coordinated using standard VERSANT distributed two-phase commit. Objects requested by an application are fetched from only one of the servers, but appropriate locks are acquired on both. During normal operation, identical transactions are processed simultaneously thereby maintaining logical equivalence of the two databases.

11.2 Redundancy is Application Transparent

The VERSANT FT Server is unique in its support for database mirroring. Since fault tolerance is implemented within VERSANT's standard two-phase commit protocol, both transaction and database integrity are preserved. Only VERSANT provides this functionality without any changes to applications or class libraries; fault tolerance is managed as a DBA function.

11.3 "MASTER-SLAVE" Operation

Should a replicated database or the network connection fail due to either software or hardware causes, failure of the connection between the databases is treated as a database failure and all changes from that point are recorded to a change log. To resynchronize the databases, this log is applied to the failed database when restarted. As with standard operation, the synchronization process is transparent to the applications. Client applications that use a "slave" database for primary access are not allowed to update objects within the slave database until the network connection is restored. This precludes the possibility of creating inconsistencies resulting from uncoordinated updates to replicated databases.

[16] This section is reproduced from http://www.versant.com/versant/products/ftserver.html.

11.4 Geographic Distribution

Just as VERSANT provides transparent database distribution, allowing objects residing in one database to transparently reference objects residing in any other database across the network, the VERSANT FT Server allows replicants to be widely separated across LANs or WANs. For example, an organization could place a system in California and its replicant in London to protect their operations. For some organizations, the ability to protect system operations from unforeseen occurrences is a standard part of disaster planning.

12. 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} (10^{24}) 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 databases

SSSWG - the Storage System Standards Working Group

TB - 10^{12} bytes

VLDB - Very Large Database

VLM - Very Large Memories (> 2GB, i.e. requiring 64-bit addressing)

VMLDB - Very Many Large Databases

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

13. 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 97-7

[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 an 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. Also appears in [17].

[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 O_2, 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