# BUILDING PETABYTE DATABASES WITH OBJECTIVITY/DB

*Leon Guzenda*
Objectivity, Inc., Mountain View, California, USA.

**Abstract**
Objectivity, Inc. has been working with the CERN RD45 Project for several years to investigate the possibility of using a commercial Object Database Management System (ODBMS) as the basis for storing data from the Large Hadron Collider and other projects. This paper explains the fundamental concepts behind Objectivity/DB in the context of the problems inherent in building, using and maintaining multi-petabyte databases.

## 1. BACKGROUND

Since its inception in 1988, Objectivity, Inc. has concentrated its efforts in two main markets: becoming embedded in a wide range of specialized software and hardware products; and large projects, primarily in the telecom and Very Large Database (VLDB) arenas.

In 1997 the CERN RD45 Project selected Objectivity/DB for further study as a candidate for storing data from the LHC and other HEP projects. The desirability of moving to an object-oriented programming environment had quickly shown the need for a mechanism other than structured files or RDBMSs for storing and manipulating the data. Several projects are now moving into production with Objectivity/DB. This paper looks at the main challenges inherent in building multi-petabyte databases, some current solutions and some future work.

## 2. THE CHALLENGES

The most obvious feature of the data from large HEP projects is the sheer volume of data produced. Much of the raw data is essentially simple, or is distilled into commonly understood representations. It is not currently economically feasible to keep all of this data, or even all of the distilled data, online at all times. This presents a problem of tape management that is rarely seen outside of a few commercial data warehouses.

Storing the data is one problem. Manipulating it efficiently in compute intensive algorithms puts an extra stress on the DBMS that is similar to the requirements of advanced CAD applications. The DBMS has to attempt to access data at speeds comparable to those provided by RAM, yet it is confronted with disks that typically have 5 to 15 mSec response times. There is a 10 ** 6 disparity between RAM and disk access and this gap may widen as the years progress.

Once large bodies of data have been collected from repetitive or similar experiments, there remains the problem of being able to make ad hoc searches across the data. This problem is not unique to the HEP community. Multidimensional searches are equally important to the data warehousing community and to other physical and biological experimenters.

There is both a need to gather and store data over very long periods (25 years, or more) at virtual central locations and a need to keep isolated data stores that may be added to the “central” store at an appropriate time. Similarly, it is convenient to be able to extract datasets for local (private) processing. These local datasets may or may not be returned to the “central” database.

## 3. CURRENT SOLUTIONS

### 3.1 Scalability

Objectivity/DB is a scalable, distributed Object DBMS. It is scalable both in the number of users that may be storing or manipulating data at a given time and in the total amount of data that may be addressed in a single transaction.

Objectivity/DB holds objects (C++, Java or Smalltalk) in standard files, which are internally and individually structured, into a hierarchical storage structure. The highest level of this structure is the Federated Database (often simply known as the Federation). The Federated Database holds schemas, catalogs, security information and housekeeping for backups. This information may be replicated into Autonomous Partitions. A Partition is a logical subset of the Federation. Each Partition controls one or more Databases. These Databases may be on different machines and may hold objects created by heterogeneous operating system, platform or compiler/language environments. Each Database may itself be replicated across many Partitions (no more than one image per Partition). The combination of Partitions, with their own system resources, and Database Replicas provides a very fault tolerant environment.

Each Database is further sub-divided into one or more Containers. A Container may hold one or more Objects, which need not be of the same class. Each Object looks like a standard C++, Java or Smalltalk object with a few extensions to cover varying length arrays (VArrays) and named relationships between object and/or containers (Associations).

Each object is given a 64-bit Object Identifier (OID) which is retained for its lifetime. On current filesystems, the OID can locate an object anywhere within a 64 Exabyte (10 ** 18) byte address space. Users never manipulate OIDs directly. They use REFs, which are smart pointers, to refer to an existing object.

The data is transferred to and from disk by local or remote Page Servers. These may be industry standard nfsd daemons, making it possible to use a wide range of fileservers, or Objectivity's Advanced Multithreaded Server (AMS). A catalog of databases (kept in the Federated Database and synchronized across Partitions) makes it possible for the Storage Manager, which runs in the client process space, to locate any object. This distributed and parallel architecture means that unrelated processes or threads may use their own processing and I/O resources without interfering with one another. As an example, two users may be running applications on their own laptop PCs, totally disconnected from the rest of the federation. Or, two powerful workstations within the same LAN or WAN may be caching local copies of read-only data while creating other data, eventually to be shared, from that data. This is particularly important in the data acquisition phase of a typical project. Individual processors may write to dedicated disks and maintain a very high throughput. They need not be constrained by a central server.

The AMS may be interfaced with the High Performance Storage System (HPSS) via a layer called oofs. This layer can be re-implemented by users to allow it to access non-HPSS equipment. HPSS is an implementation of an IEEE standard Mass Storage Device controller. It holds a great deal of promise for managing the physical media involved in petabyte-plus systems.

Concurrency is handled with the traditional ACID transaction paradigm, implemented via simple Lock Server processes, which provide hierarchical Gray-code locking. Each Partition has its own Lock Server, to avoid having single points of failure in the Federation. The Lock Servers handle distributed deadlock detection.

### 3.2 Fast Access to the Data

Objectivity/DB provides several standard mechanisms for finding data rapidly. These include B-tree indices, hash tables, collection classes (C++ STL bags, sets, lists etc.) and navigational access via named associations. Objects may also be named and versioned. Objectivity/DB provides iterators for

finding groups of data. The iterators may be initialised with a simple or compound predicate (which may even be a valid SQL SELECT statement!). There are standard iterators for moving down the containment hierarchy, scanning a layer (or nested layers) of the hierarchy; and for traversing named associations. The main difference between an iterator and the SELECT plus move cursor arrangement provided by SQL is that an iterator starts returning qualified objects as soon as any are found. This is obviously useful if extremely large numbers of objects are to be examined. An RDBMS will generally assemble a complete view or table to satisfy the query before sending any response back to the client.

Objectivity/DB provides both uni-directional and bi-directional associations. Their cardinality may be 1:1, 1:N, N:1 or M:N. Associations may link objects in different containers or databases. This is transparent to the user, as Objectivity/DB provides a single address space within the federation.

It is often feasible to cluster data that will generally be used together into the same logical/physical pages on disk. This is done by providing a clustering directive at the time that an object is created. The clustering directive may be a REF of an object, container or database; or it may be a function that implements some policy. It may even be declustering rapidly arriving objects onto different disk drives to keep up with the incoming stream. It is possible to recluster objects at a later date, using the moveObject() method. This changes the OID, but all bi-directional associations, indices, hash-tables etc. are automatically adjusted. This mechanism was chosen in preference to a logically separate table of OIDs in order to reduce storage overheads and performance bottlenecks. It is interesting to note that immediately after committing a newly created object to disk there is no single place that holds the actual OID of the object. This means that a semi-smart process working through physical pages could theoretically reconstruct a badly corrupted database.

The physical page size is set by the database designer and is the basic unit of transfer between disk and RAM and across the network. Objectivity/DB differentiates between pages with many small objects in them and objects which span many physical pages. It only transfers the pages that are actually used into the client's cache. As a process or thread acquires locks to access each container of objects it picks up information about the status of its previously cached pages for that container. In good cases, the previously cached pages have not been changed by any other thread/process, so they may be reused. This gives a considerable performance boost in a surprisingly wide range set of circumstances. In the worst case, out of date pages are refreshed automatically by the Storage Manager, so cache management is not a burden to the application programmer.

We decided early on in the development of Objectivity/DB that it would be too expensive to lock individual applications. Experiments confirmed our view, particularly with large sets of complex data and relationships. Queuing and deadlocks impacted performance and complicated the error recovery in individual methods and programs. So, we decided to lock at the container level. After all, the container stores objects that are frequently accessed together, so maybe we can exploit this behaviour. We implemented an extra transaction mode, called MROW, for Multi-Reader, One Writer to overcome the objection that too many objects would be made inaccessible if an updater locks a container for an extended period. MROW allows a writer to create a new version of a container while readers view the currently committed version. This mechanism reduces lock requests (which typically run at around 400 RPCs per second across an Ethernet), reduces queueing and deadlocks; and increases system throughput. The outcome is that object level locking is still low on the list of user group requirements after almost ten years!

### 3.3 Ad Hoc Queries

As mentioned above, Objectivity/DB provides iterators, which may be initialised with simple or compound predicate conditions. The iterators will take advantage of any relevant indices. The Object Database Management Group included an Object Query Language as a part of the ODMG'93 standard for Object Databases. After careful consideration, Objectivity decided to stick with our commitment to provide both an ANSI standard SQL 2 interface to Objectivity, called Objectivity/SQL++ rather than try to implement the parsers and APIs of OQL. We also added ODBC

interfaces. Objectivity/SQL++, released with Objectivity Version 2.0, became the first implementation of standard SQL based on a pure ODBMS. The strategy has paid off. There are over 600 ODBC-compliant tools that will work with RDBMSs and Objectivity/SQL++. There are no independent (i.e. provided by a vendor other than an ODBMS vendor) OQL tools on the market today, after five years of potential market development. So, it now looks likely that OQL will converge with SQL3 and 4, ratifying our original strategy.

The current release of Objectivity/SQL++ can execute simple methods within the SQL++ server as a part of a query. For instance, it could execute a getAge() method to compute a value from a Date_of_Birth field in a Person object. Later, as the syntax for doing this within SQL4 becomes stable, we hope to extend SQL++ to handle any existing C++, Java or Smalltalk method/message.

### 3.4 Distributed and Private Databases

The Federation plus Partition and replication mechanism is a very powerful paradigm for providing a distributed data store within a collaboration. Everything within a federation is potentially visible from anywhere on the network. Individuals may create their won Partitions and replicate any databases they wish. They may also create new databases and then either delete them or make the available to the community. It is also possible to copy a database and attach it to a compatible Federation. It must retain the same database name and identifier and the schemas must be identical for this to be successful.

C++ environments are currently compiled and programs consist of binary images. Objectivity/DB must present the C++ objects in the format expected by the local compiler. So, if a class definition changes there are immediate ramifications across the system. Programs must be recompiled and object instances must be treated appropriately. Objectivity/DB provides a variety of ways of dealing with the object instances. For any class that is changed, the application designer may opt to have all instances atomically migrated to the new format; or have instances migrated as they are located; or have instances converted every time they are encountered. This still leaves the problem of migrating the C++ binary forward, but this is a code control problem with or without Objectivity/DB being present.

Given the pace of technological developments and the life span of some of the projects, it is completely impracticable to assume that the hardware, operating systems and languages will be constant throughout a project. Fortunately, Objectivity/DB transparently converts objects on demand to help cope with these issues. It does this by building a state table at the time that the object classes are first defined to Objectivity/DB (through C++ DDL or Java/Smalltalk environments). Only the objects or VArrays that are actually used are converted. The state machine only touches the fields that need conversion or movement. This is more efficient than adopting a single neutral format, which necessitates double conversion of almost all objects whenever they are actually used. This means that a few CPU cycles may be used, but the time taken is insignificant compared with the I/O which has immediately preceded the first conversion.

## 4. FUTURE WORK

We have provided named associations, hash tables and multi-key B-tree indices. We are currently experimenting with a multi-dimensional R-tree geospatial index. If this proves successful then it may be generalised for other markets. We are also tracking work by researchers at CERN and various Universities around the world to provide a wider variety of indexing techniques as a standard part of the product.

The initial experiments with data replication have been quite encouraging. A database was replicated between CERN (Geneva) and CalTech (Pasadena, California) and subsequent updates were successfully applied either immediately or by automatic resynchronization. However, we will

continue to provide improvements in this area, for example, by multicasting pages into the network whenever possible.

The current release of Objectivity (V5) only supports one file per database. This will be changed at the next release to meet the architectural requirement that a file may hold one or more containers, each container coming from the same database. We also need to remove the restriction that a copied database may only use its existing name and OID if it is attached to an existing Federation

Initial work with HPSS has been encouraging. However, there is further work to do in order to provide a random access disk cache and to implement Kerberos authentication. HPSS has a powerful concept of policies, or "Class of Service" for managing the staging of files. This looks very promising. We are seeking ways to exploit this mechanism, possibly by sending hints between the client and the AMS, or between the AMS and the HPSS.

### 4.1 References


1. R.G.G. Cattell, Editor, "ODMG'93 – The Object Database Standard" Revision 2, Morgan Kauffmann, 1998.
2. Objectivity, Inc., "A Technical Overview of Objectivity/DB", http://www.objectivity.com, 1998.