An ODMG-compatible Testbed Architecture for Scalable Management and Analysis of Physics Data [1]

David M. Malon [a], Edward N. May [a]

[a] *Argonne National Laboratory*
9700 South Cass Avenue
Argonne, Illinois USA
malon@anl.gov, may@anl.gov

This paper describes a testbed architecture for the investigation and development of scalable approaches to the management and analysis of massive amounts of high energy physics data. The architecture has two components: an interface layer that is compliant with a substantial subset of the ODMG-93 Version 1.2 specification, and a lightweight object persistence manager that provides flexible storage and retrieval services on a variety of single- and multi-level storage architectures, and on a range of parallel and distributed computing platforms.

Key words: Object-oriented databases; object persistence; multilevel storage management; ODMG

1 Introduction

Understanding scalability for physics data analysis requires investigating approaches to data organization and clustering, caching and migration, replication, multiple data access paths, nonuniform data access and multilevel storage, parallelism, and more. The roles of parallel file systems, mass storage architectures, non-dedicated parallel computing platforms, and concurrent use of a heterogeneous mix of storage devices must also be understood. To undertake such studies, we have developed and implemented a flexible, lightweight object persistence manager that meets the following design criteria:

[1] The submitted manuscript has been authored by a contractor of the U.S. Government under contract No. W-31-109-Eng-38. Accordingly, the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

- access to every persistent object from every query node;
- support for efficient reorganization of data, including striping and reclustering, without knowledge of object schemata;
- extensible support for a variety of storage mechanisms, including local and remote disk, raw RAID, Unitree file systems, raw device access to DD2 and 8 mm tape, parallel file systems such as IBM's Vesta and PIOFS, and Internet data access via standard FTP and HTTP mechanisms or cgi-bin scripts;
- support for data replication;
- support for multiple access paths to data;
- portability to heterogeneous distributed architectures.

We have tested the software on a range of platforms, including Argonne's IBM SP PowerParallel system, and on a heterogeneous collection of UNIX workstations. Experiments have been conducted using both Fermilab D0 data and the output of ISAJET Monte Carlo simulations.

While investigations of the sorts outlined above could not have been undertaken with commercial database software, we have tried nonetheless to provide an interface that does not needlessly inhibit coexistence with (and, perhaps, eventual migration to) commercial object-oriented databases. To this end, we have defined an interface layer that is compliant with a substantial subset of the ODMG-93 Version 1.2 specification. In the process, we have learned a great deal about potential deficiencies and scalability implications of the ODMG specification in general, and of its C++ binding in particular. We have also sought to define a minimal interface that any lightweight object persistence manager should support in order that an ODMG-compliant system be buildable above it.

2 The Storage Services Architecture

The ODMG-93 Release 1.2 C++ binding defines the user interface to our persistence manager—briefly, persistent objects reside in a database instantiated by the ODMG-defined class `d_Database`, and references to persistent objects of class `T` are made via a template class `d_Ref<T>`. Details appear in the current ODMG draft standard [1]. Argonne's portable ODMG-compliant frontend to object persistence managers is summarized later in this manuscript, and described in greater detail in [2]. Internal to the Argonne object persistence manager, data are organized logically into *stores*. Within each store, objects are allocated in contiguous blocks of bytes called *segments*. Segments are the fundamental units of data transfer to and from a query process—when a reference to a persistent object is dereferenced, the corresponding segment containing the object is located and moved or mapped into memory.

For a variety of reasons, segments are generally inappropriate as the highest-level units for storage management—there are too many of them, and they are far too small for efficient storage and retrieval on high-latency devices (unless they are far too large to be appropriate for memory caching when a single object is touched).

Segments may instead be aggregated arbitrarily to form *folios*, which are the units in which storage devices will deal with the data. Examples of folios include an ordinary Unix file containing one or more segments, a raw DD2 tape partition containing n consecutive segments of a store, a raw RAID partition contiguously containing every k^{th} segment of a store, and a Unitree file containing the k most frequently accessed segments. Folios have appeared in other lightweight object persistence managers such as PTool [4], but the design differences between PTool and the Argonne software are substantial. In PTool, a persistent pointer names the folio in which the object is contained. In our software, folios are *orthogonal* to persistent pointers. We can reorganize segments into arbitrary folio arrangements, and all persistent pointers will be unchanged and valid. A storage server is free to rearrange segments without knowledge of segment contents, and without concern for external references to objects within the segment.

Physical location and access method identification for segments are maintained in metadata represented by an ordered list of retrieval rules. Dereferencing a pointer to a persistent object identifies the store number and segment number containing the object data. A Segment Locator, which is in general replicated on each compute node, provides a *whohas()* method to map this ordered-pair segment identification into a segment retrieval rule, customarily the first retrieval rule in the list matching the segment id. A *getServer()* method in turn accepts a retrieval rule as an argument, and returns a pointer to a segment server capable of reading and writing the corresponding segment.

An abstract SegmentServer base class defines the interface to the handful of methods that all segment servers must provide, such as segment creation, retrieval, and updating. Segment servers for particular devices are derived from this base class. To add a new storage medium to the list of supported devices, one need only implement the SegmentServer methods for the particular device, add a new reserved word to the retrieval rule lexicon, and update the getServer method to associate the new reserved word with the new Segment Server—all other code continues to operate unchanged.

What does a retrieval rule look like? For a single segment, it may be as simple as associating a hostname (or localhost), a device type, and an address (for example, a Unix file name, an offset for a raw device, or a partition number and an offset for a tape) to a (store, segment number) pair. A single rule may define a placement convention for all the segments in a store by use of a wildcard character in place of the segment number. Such a rule might specify, for example, that each segment k should be stored in Unitree with the pathname /mss/username/MyDatabase.k). If a particular segment is to be handled differently, perhaps because it has been locally cached, all that is necessary is to place a rule corresponding to that particular segment number *earlier* in the rule list than the default rule for segments of that store. Retrieval rules may also define segment-level striping (e.g., place segments in contiguous blocks of four, in round-robin fashion, into three Unitree files, repeating the process until the files are 32 segments long, then build three new Unitree files and repeat the process).

In practice, one often begins with a master retrieval rule list that describes the location of data segments (often in mass storage). When a segment is cached or copied or moved to another storage location, a new retrieval rule list is derived, either by replacing the original rule, or by adding a rule for reaching the new location earlier in the retrieval rule list. A segment locator using the new rule list will match the segment id to the retrieval rule it encounters first.

Rearranging data is straightforward: a utility for that purpose can be built essentially by incorporating a segment locator to read the current retrieval rules, and an array of segment servers to read segments and write them elsewhere. The point is that no knowledge of the underlying object schemata is required—storage can be managed orthogonally to the data store's content.

Local Caching and Replication: When a segment is copied, a corresponding updated retrieval rule list reflects its new location. Since each query node may have its own segment locator, each may use its own retrieval rule list. The consequence is that there is a choice—one could have each node i retrieve data from node j rather than from mass storage, for example, by sharing node j's rule list, or have each node talk only to its own disks and to mass storage by not sharing retrieval rule lists.

Preloading Local Disks: Recommended policy on many massively parallel architectures is to preload data, particularly shared read-only data, onto local disks before running the computational portion of a job. This is important in order to avoid serial bottlenecks (often especially paralyzing when hundreds of nodes are trying to read the same AFS- or NFS-mounted file, or even different files from the same file system). Systems often provide utilities to copy data to P parallel nodes in $\log P$ time. When data segments are preloaded, corresponding retrieval rule lists are built to reflect the new locations. As noted above, these lists may be different on every query node, but when the same segment is replicated, things are generally simpler. If segment k of a certain data store contains, for example, calibration data needed in the analysis of each event, segment k may be replicated on each query node's local disk, and a corresponding retrieval rule matching the segment id to, for example, a disk file named /tmp/scratch/username/MyDatabase.k on localhost, would likely be identical on every node.

Multiple Access Paths: The architecture allows multiple retrieval rules for a single data segment. The design is intended eventually to support, for example, finding the first matching retrieval rule for a segment, trying it, and if it fails to return in an acceptable amount of time, trying the next matching rule, or even associating estimated costs with each candidate retrieval rule and optimizing the choice. We have not taken advantage of either of these approaches to date. Different processes may today, however, follow different access paths to the same data by using different retrieval rule lists.

3 The User Interface Layer

The Object Database Management Group (ODMG) is an industry consortium of database vendors and others who have come together to agree upon aspects of a common specification for object databases. These efforts have resulted in an emerging standard (currently ODMG-93 Release 1.2 [1]) whose components include: an object model; an Object Definition Language (ODL); an Object Query Language (OQL); a C++ binding for ODL and OQL, and a C++ Object Manipulation Language; a Smalltalk binding for ODL and OQL, and a Smalltalk Object Manipulation Language. While ODMG-93 is an object database specification, a significant subset of it can be supported in a natural way by many lightweight object persistence managers.

Our primary goal in defining an ODMG-aware interface layer has been to provide high-performance access to the functionality of the underlying persistence manager, while maintaining compatibility wherever possible with the ODMG-93 standard's C++ binding. Where this has not been possible, we have striven to document carefully the differences and the reasons for them.

Along the way, we have tried to define and maintain a clear and consistent boundary between the ODMG-aware interface layer of our software and the underlying persistence manager. To this end, we have asked the question, "What is the *minimal interface* that any persistence manager should support in order that it be possible to build an ODMG-compliant database on top of it?" We have evolved an interface that we believe is a viable first draft of an answer to this question. One measure of our success is that it should be possible to implement our ODMG-aware software on top of a wide range of lightweight object persistence servers other than our own, as long as they are capable of supporting this minimal interface. To date, we have tested this idea in two implementations, one using the Argonne Lightweight Object Persistence Manager as the underlying persistence service, the other using a locally enhanced version of the PTool[4] software from the University of Illinois at Chicago.

In our definition of the user interface layer, we have endeavored to provide, in an ODMG-compliant way, the functionality that users of our scientific data stores demand. Salient features are object schema definitions that do not require extensions to the C++language, databases that may be opened in read-write or read-only mode, Ref-based access to persistent data consistent with C++ pointer usage, support for object naming, support for string storage and retrieval, and provision of collection classes and their associated iterators.

Beyond the requirements of functionality, our aim has been to enable use of as much of the ODMG-93 C++ binding as possible without requiring query language parsing or preprocessing object schemata. We support

- the `d_Database` class;
- the templated collection facilities `d_Collection<T>`, `d_Bag<T>`, `d_List<T>`, `d_Set<T>`, and `d_VArray<T>`, and their auxiliary iterator class `d_Iterator<T>`, except for the four `d_Collection<T>` methods that require parsing OQL query strings;
- the time utility classes `d_Date`, `d_Time`, `d_Timestamp`, and `d_Interval`;
- the `d_String` class;
- reference-based object access via the template class `d_Ref<T>` and the class `d_Ref_Any`;
- a rudimentary `d_Object` implementation (but classes need not derive from it to be persistence-capable);
- the semantics of `d_Transactions`, and the use of `d_Transactions` either as scoping rules only (for efficiency), or as a means to allow checkpoint, commit, and abort operations.

Because we do not currently parse OQL, the ODMG-93 class `d_OQL_Query` is not supported, nor is the global `d_oql_execute` function.

The lack of schema preprocessing has a number of implications. A beneficial consequence of this approach is that users may allocate any object in persistent memory without our software being aware of the object's schema. There are, however, ODMG non-compliance consequences as well: for example, while ODMG-93 prescribes that a `d_Error` object be thrown if an assignment of a `d_Ref<B>` to a `d_Ref<A>` is attempted when a B is not an A, we do not detect this problem. More significantly, we do not yet support *Relationships*. Maintaining referential integrity of this sort is currently the responsibility of the user.

We have attempted to be quite parsimonious in what we require of the underlying persistence server. We require the existence of two classes, which we denote by `Store` and `Pptr`, as underlying implementation classes for ODMG-

93's `d_Database` and `d_Ref<T>` classes, respectively. We assume that we can open and close a `Store`, and allocate contiguous blocks of bytes therein. We assume that the persistence server can convert a persistent pointer (a `Pptr`) that refers to an object in a `Store` into a valid memory address of that object's image. Only a few additional features are required; these are described in [2].

4 Status and Directions

We have implemented the architecture described above on the Argonne 128-node IBM SP system, and on networks of Unix workstations, including the Parallel Distributed Simulation Facility at the National Energy Research Scientific Computing Center. We have developed segment servers for all of the storage devices mentioned in the text. We have used these facilities both to provide access to storage on parallel and distributed platforms with a heterogeneous mix of storage media, and as a testbed to begin study of the complicated issues involved in physical data organization—alternative striping strategies, caching, replication, use of multilevel storage, and the role of parallel file systems. The ability to rapidly reconfigure our storage utilization without worrying about data store contents has proven invaluable. Future work on the storage server will involve tools for managing retrieval rule lists, and smart (e.g., configuration-aware) tools for automating storage reorganization.

Development efforts on the user interface layer are directed toward supporting a more extensive subset of ODMG-93, and toward providing parallelism and access to the persistence server's underlying multilevel storage allocation and management without compromising ODMG-93 compliance.

References

[1] R.G.G. Cattell et al, *The Object Database Standard: ODMG-93 Release 1.2* (Morgan Kaufmann, San Francisco, 1996).

[2] D.M. Malon and E.N. May, On persistence interfaces for scientific data stores, *ANL-HEP-96-CP-09* (submitted for publication, 1996).

[3] D.M. Malon and E.N. May, Flexible storage services for parallel data mining, *ANL-HEP-96-CP-40* (submitted for publication, 1996).

[4] R. L. Grossman and X. Qin, "Ptool: a scalable persistent object manager," *Proceedings of SIGMOD 94* (ACM, 1994) page 510.