The OO7 Benchmark*

Michael J. Carey David J. DeWitt Jeffrey F. Naughton
Computer Sciences Department
University of Wisconsin-Madison

Abstract

The OO7 Benchmark represents a comprehensive test of OODBMS performance. In this paper we describe the benchmark and present performance results from its implementation in three OODBMS systems. It is our hope that the OO7 Benchmark will provide useful insight for end-users evaluating the performance of OODBMS systems; we also hope that the research community will find that OO7 provides a database schema, instance, and workload that is useful for evaluating new techniques and algorithms for OODBMS implementation.

1 Introduction

Builders of object-oriented database management systems are faced with a wide range of design and implementation decisions, and many of these decisions have a profound effect on the performance of the resulting system. Recently, a number of OODBMS systems have become publically available, and the developers of these systems have made very different choices for fundamental aspects of the systems. However, perhaps since the technology is so new, it is not yet clear precisely how these systems differ in their performance characteristics; in fact, it is not even clear what performance metrics should be used to give a useful profile of an OODBMS's performance. We have designed the OO7 Benchmark as a first step toward providing such a comprehensive OODBMS performance profile.

Among the performance characteristics tested by OO7 are:

- The speed of many different kinds of pointer traversals, including traversals over cached data, traversals over disk-resident data, sparse traversals, and dense traversals;
- The efficiency of many different kinds of updates, including updates to indexed and unindexed object fields, repeated updates, sparse updates, updates of cached data, and the creation and deletion of objects;
- The performance of the query processor (or, in cases where the query language was not sufficiently expressive, the query programmer) on several different types of queries.

By design, the OO7 Benchmark produces a set of numbers rather than a single number. A single number benchmark has the advantage that it is very catchy and easy to use (and abuse) for system comparisons. However, a benchmark that returns a set of numbers gives a great deal more information about a system than does one that returns a single number. A single number benchmark is only truly useful if the benchmark itself precisely mirrors the application for which the system will be used.

In this paper, we describe the benchmark and give preliminary performance results from its implementation in one public-domain research system (E/Exodus) and two commercially available OODB systems (Objectivity/DB, which is also available as DEC Object/DB V1.0, and Ontos). Due to tight space constraints, the descriptions here are necessarily sketchy, and not all of the results can be presented. A more detailed benchmark description, together with a full and final set of performance results for all of the participating systems[1] can be found in [CDN93]. Lastly, it should be mentioned that we had also expected to include results for another commercial system, the ObjectStore system from Object Design, Inc. Unfortunately, on the day before the camera-ready deadline for this proceedings, ODI had their lawyers send us a notice saying that they were dissatisfied with the way that we had run the benchmarking process and that we had to drop our ObjectStore

*DEC provided the funding that began this research. The bulk of this work was funded by DARPA under contract number DAAB07-92-C-Q508 and monitored by the US Army Research Laboratory. Sun donated the hardware used as the server in the experiments.

[1]We are currently finishing up the benchmark on another commercial system (O2), and its performance will be included in [CDN93]. We also invited Versant to participate in the benchmark, but they declined to participate until the next release of their system was available.

results from the paper or else face possible legal action. It is unfortunate that they chose to withdraw, as ODI's approach to persistence provided some interesting contrasts with the other systems.

The remainder of the paper is organized as follows. Section 2 compares the OO7 Benchmark to previous efforts in OODBMS benchmarking. Section 3 describes the structure of the OO7 Benchmark database. Section 4 describes the hardware testbed configuration we used to run the benchmark, and gives a brief overview of the systems tested. Section 5 describes the benchmark's operations and discusses the experimental results for each operation as it is presented. Finally, Section 6 contains some conclusions and our plans for future work.

2 Related Work

In this section, we briefly discuss the previous benchmarking efforts that are related to OO7, and we cite the reasons why we felt that there was a need for additional work in the OODBMS benchmarking area. A much more in-depth treatment of related work can be found in [CDN93].

2.1 Previous OODBMS Benchmarks

The OO1 Benchmark[2] [CS92], commonly referred to as the Sun Benchmark, was the first widely accepted benchmark that attempted to predict DBMS performance for engineering design applications. Because of its early visibility and its simplicity, OO1 has became a de facto standard for OODB benchmarking.

Another benchmark that OO7 is closely related to is the HyperModel Benchmark developed at Tektronix [And90]. Compared to OO1, Hypermodel includes both a richer schema (involving several different relationships and covering a larger set of basic data types) and a larger collection of benchmark operations (including a wider variety of lookup, traversal, and update operations).

There are several other OODB studies related to our work on OO7. Ontologic used the initial Sun Benchmark to study the performance of Vbase, their first OODB product offering [DD88]. Researchers at Altair designed a complex object benchmark (ACOB) for use in studying alternative client/server process architectures [DFMV90]. Finally, Winslett and Chu recently studied OODB (and relational DB) performance by porting a VLSI layout editor onto several systems [WC92]. However, only the file I/O portions of the editor were modified, so this work focused on save/restore performance rather than performance when applications are operating on database objects.

[2] Object Operations, version 1.

2.2 Why Another Benchmark?

OO1 and HyperModel both represent significant efforts in the area of OODB benchmarking. Why, then, did we feel a need for "yet another" benchmark in this area? As mentioned briefly in the introduction, neither of the existing benchmarks was sufficiently comprehensive to test the wide range of OODB features and performance issues that must be tested in order to methodically evaluate the currently available suite of OODB products. For example, both benchmarks lack any real notion of complex objects, yet these are expected to be the natural unit for clustering in real OODBMS applications. In addition, neither benchmark provides more than rudimentary testing of associative operations (object queries), and neither covers issues such as sparse vs. dense traversals, updates to indexed vs. non-indexed object attributes, repeated object updates, or the impact of transaction boundary placement [CDN93].

3 OO7 Database Description

Since the OO7 Benchmark is designed to test many different aspects of system performance, its database structure and operations are nontrivial. The most precise descriptions of the OO7 Benchmark are the implementations of the benchmark. These implementations are available by anonymous ftp from the OO7 directory of `ftp.cs.wisc.edu`. In addition, we have written a reference C++ implementation of the benchmark. This C++ implementation is also available. The informal description of the benchmark given here should suffice for understanding the basic results; a more detailed description of the benchmark, including a schema, is presented in [CDN93]. Anyone planning to implement the benchmark should obtain a copy of one of the available implementations.

The OO7 Benchmark is intended to be suggestive of many different CAD/CAM/CASE applications, although in its details it does not model any specific application. Recall that the goal of the benchmark is to test many aspects of system performance, rather than to model a specific application. Accordingly, in the following when we draw analogies to applications we do so to provide intuition into the benchmark rather than to justify or motivate the benchmark. There are three sizes of the OO7 Benchmark database: small, medium, and large. Table 1 summarizes the parameters of the OO7 Benchmark database.

3.1 The Design Library

A key component of the OO7 Benchmark database is a set of *composite parts*. Each composite part corresponds to a design primitive such as a register cell in a VLSI CAD application, or perhaps a procedure in a

Parameter	Small	Medium	Large
NumAtomicPerComp	20	200	200
NumConnPerAtomic	3,6,9	3,6,9	3,6,9
DocumentSize (bytes)	2000	20000	20000
Manual Size (bytes)	100K	1M	1M
NumCompPerModule	500	500	500
NumAssmPerAssm	3	3	3
NumAssmLevels	7	7	7
NumCompPerAssm	3	3	3
NumModules	1	1	10

Table 1: OO7 Benchmark database parameters.

CASE application; the set of all composite parts forms what we refer to as the "design library" within the OO7 database. The number of composite parts in the design library, which is controlled by the parameter *NumCompPerModule*, is 500. Each composite part has a number of attributes, including the integer attributes `id` and `buildDate`, and a small character array `type`. Associated with each composite part is a *document* object, which models a small amount of documentation associated with the composite part. Each document has an integer attribute `id`, a small character attribute `title`, and a character string attribute `text`. The length of the string attribute is controlled by the parameter *DocumentSize*. A composite part object and its document object are connected by a bi-directional association.

In addition to its scalar attributes and its association with a document object, each composite part has an associated graph of *atomic parts*. Intuitively, the atomic parts within a composite part are the units out of which the composite part is constructed. In the small benchmark, each composite part's graph contains 20 atomic parts, while in the medium and large benchmarks, each composite part's graph contains 200 atomic parts. (This number is controlled by the parameter *NumAtomicPerComp*.) For example, if a composite part corresponds to a procedure in a CASE application, each of the atomic parts in its associated graph might correspond to a variable, statement, or expression in the procedure. One atomic part in each composite part's graph is designated as the "root part."

Each atomic part has the integer attributes `id`, `buildDate`, `x`, `y`, and `docId`, and the small character array `type`. (The reason for including all of these attributes will be apparent from their use in the OO7 Benchmark operations, described in Sections 5.1 through 5.3.) In addition to these attributes, each atomic part is connected via a bi-directional association to several (3, 6, or 9) other atomic parts, as controlled by the parameter *NumConnPerAtomic*. Our initial idea was to connect the atomic parts within each composite part in a random fashion. However, random connections do not ensure complete connectivity. To ensure complete connectivity, one connection is initially added to each atomic part to connect the parts in a ring; more connections are then added at random. In addition, our initial plans did not specify a 3/6/9 interconnection variation. This variation was included to ensure that OO7 provides satisfactory coverage of the OODBMS performance space, as our preliminary tests indicated that some systems can be very sensitive to the value of this particular benchmark parameter.

The connections between atomic parts are implemented by interposing a connection object between each pair of connected atomic parts. Here the intuition is that the connections themselves contain data; the connection object is the repository for that data. A connection object contains the integer field `length` and the short character array `type`.

Figure 1 depicts a composite part, its associated document object, and its associated graph of atomic parts. One way to view this is that the union of all atomic parts corresponds to the object graph in the OO1 benchmark; however, in OO7 this object graph is broken up into semantic units of locality by the composite parts. Thus, the composite parts in OO7 provide an opportunity to test how effective various OODBMS products are at supporting complex objects.

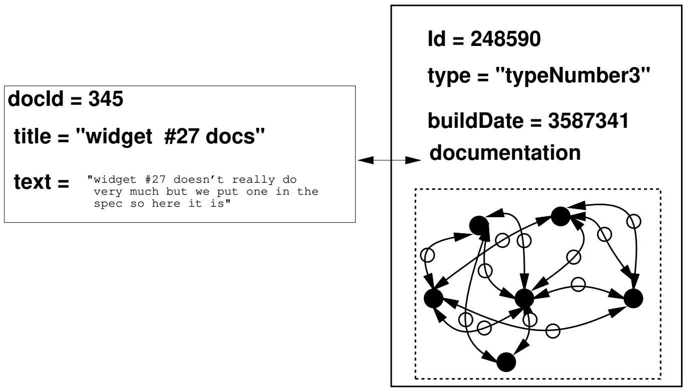

Figure 1: A Composite Part and its associated Document object.

3.2 Assembling Complex Designs

The design library, which contains the composite parts and their associated atomic parts (including the connection objects) and documents, accounts for the bulk of the OO7 database. However, a set of composite parts by itself is not sufficiently structured to support all of the operations that we wished to include in the benchmark. Accordingly, we added an "assembly hierarchy" to the database. Intuitively, the assembly objects correspond to higher-level design constructs in the application being modeled in the database. For example, in a VLSI CAD application, an assembly might correspond to the design for a register file or an ALU. Each assembly is either made up of composite parts (in which case it is a *base assembly*) or it is made up of other assembly objects (in which case it is a *complex assembly*).

The first level of the assembly hierarchy consists of *base assembly* objects. Base assembly objects have the integer attributes `id` and `buildDate`, and the short character array `type`. Each base assembly has a bi-directional association with three "shared" composite parts and three "unshared" composite parts. (The number of both shared and unshared composite parts per base assembly is controlled by the parameter *NumCompPerAssm*.) The OO7 Benchmark database is designed to support multiuser workloads as well as single user tests; the distinction between the "shared" and "unshared" composite parts was added to provide control over sharing/conflict patterns in the multiuser workload. This paper only deals with the single user tests; only the "unshared" composite part associations are used in the single user benchmark. The "unshared" composite parts for each base assembly are chosen at random from the set of all composite parts.

Higher levels in the assembly hierarchy are made up of *complex assemblies*. Each complex assembly has the usual integer attributes, `id` and `buildDate`, and the short character array `type`; additionally, it has a bi-directional association with three subassemblies (controlled by the parameter *NumAssmPerAssm*), which can either be base assemblies (if the complex assembly is at level two in the assembly hierarchy) or other complex assemblies (if the complex assembly is higher in the hierarchy). There are seven levels in the assembly hierarchy (controlled by the parameter *NumAssmLevels*).

Each assembly hierarchy is called a *module*. Modules are intended to model the largest subunits of the database application, and are used extensively in the multiuser workloads; they are not used explicitly in the small and medium databases, each of which consists of just one single module. Modules have several scalar attributes — the integers `id` and `buildDate`, and the short character array `type`. Each module also has an associated *Manual* object, which is a larger version of a document. Manuals are included for use in testing the handling of very large (but simple) objects.

Figure 2 depicts the full structure of the single user OO7 Benchmark Database. Note that the picture is somewhat misleading in terms of scale; there are only $(2^7 - 1)/2 = 1093$ assemblies in the small and medium databases, compared to 10,000 atomic parts in the small database and 100,000 atomic parts in the medium database.

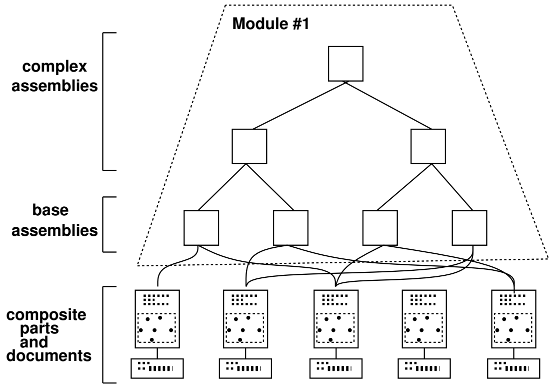

Figure 2: Structure of a module.

4 Testbed Configuration

4.1 Hardware

As a test vehicle we used a pair of Sun workstations on an isolated piece of Ethernet. A Sun IPX workstation configured with 48 megabytes of memory, two 424 megabyte disk drives (model Sun0424) and one 1.3 gigabyte disk drive (model Sun1.3G) was used as the server. One of the Sun 0424s was used to hold system software and swap space. The Sun 1.3G drive was used to hold the database (actual data) for each of the database systems tested, and the second Sun 0424 drive was used to hold recovery information (the transaction log or journal) for each system. The data and recovery disks were configured as either Unix file systems or raw disks depending on the capabilities of the corresponding OODBMS. Objectivity, for example, uses NFS to read and write non-local files, so the disks were formatted as Unix files for that system. Exodus, on the other hand, prefers to use raw disks to hold its database and log volumes.

For the client we used a Sun Sparc ELC workstation (about 20MIPS) configured with 24 megabytes of memory and one 207 megabyte disk drive (model Sun0207). This disk drive was used to hold system software and as a swap device. Release 4.1.2 of the SunOS was run on both workstations.

4.2 Software

E/Exodus

Exodus consists of two main components: The Exodus Storage Manager (ESM) and the E programming language. The ESM provides files of untyped objects of arbitrary size, B-trees, and linear hashing. The current version of the ESM (version 2.2) [EXO92] uses a page-server architecture [DFMV90] where client processes request pages that they need from the server via TCP/IP.

The E programming language [RC89] extends C++, adding persistence as a basic storage class, collections of persistent objects, and B-tree indices. The services provided by E are relatively primitive compared to its commercial counterparts. There is no support for associations, iterators with selection predicates, queries, or versions.

For these experiments, we used a disk page size of 8 Kbytes (this is also the unit of transfer between a client and the server). The client and server buffer pools were set to 1,500 (12 MBytes) and 4,500 pages (36 Mbytes) respectively. Raw devices were used for both the log and data volumes.

Objectivity/DB, Version 2.0

Unlike Exodus, Objectivity/DB, also available as DEC Object/DB V1.0, employs a file server architecture [DFMV90]. In this architecture, there is no server process for handling data. Instead, client processes access database pages via NFS. Since NFS does not provide locking, a separate lock server process is used. We placed this lock server on the same Sun IPX that was used to run the server process in the other configurations. The current release of Objectivity/DB provides only coarse grain locking, at the level of a container, and the current B-tree implementation cannot index objects distributed across multiple containers.

Objectivity, like Ontos (described next), employs a library-based approach to the task of adding persistence to C++. Instead of modifying the C++ compiler (the approach taken by E), persistent objects are defined by inheritance from a persistent root class. In addition to persistence, Objectivity/DB provides sets, relationships and iterators. Access to persistent objects is through a mechanism known as a handle. By overloading the "->" operator, handles permit the manipulation of persistent objects in a reasonably transparent fashion.

For the benchmark tests, the client buffer pool was set at 1,500 8K byte pages. Since the Objectivity architecture does not employ a server architecture, it was not necessary to set its buffer pool size. However, because SunOS uses all memory available to buffer file pages, the actual memory for buffering pages was roughly the same as for the other systems. As mentioned above, the database and shadow files were both stored as Unix files.

Ontos Version 2.2

Like ObjectStore and Exodus, Ontos employs a client-server architecture. However, Ontos is unique in its approach to persistence. Objects (which inherit from an Ontos defined root object class) are created in the context of one of three different storage managers. The "in-memory" storage manager manages transient objects much as the heap does in a standard C++ implementation. The "standard" storage manager implements an object-server architecture [DFMV90], with both the unit of locking and the unit of transfer between the client and server processes being an individual object. The third storage manager is called the "group" storage manager, and it implements a page-server architecture; the granularity for locking and client/server data transfers in this mode is a disk page.

All three mechanisms can be used within a single application by specifying a storage manager when the object is created (the C++ new() operator is overloaded appropriately). For the OO7 Benchmark, composite parts, atomic parts, and connection objects were created using the group manager. The standard object manager was used for the remaining classes of objects.

The features provided by Ontos are slightly richer than the other systems. Ontos provides three forms of bulk types: sets, lists, and associations. Associations can be either arrays or dictionaries (B-tree or hash indices). Iterators are provided over each of the bulk types, including a nice object-SQL interface. Unfortunately, the system lacks a query optimizer for object-SQL, so we did not use object-SQL to express the benchmark's queries (as performance would not have been acceptable). Support is also provided for nested transactions, an optimistic concurrency control mechanism, notify locks, and databases spanning multiple servers.

The approach to buffering on the client side is different in Ontos from each of the other systems. Instead of maintaining a client buffer pool, persistent objects are kept in virtual memory under the control of a client cache. This approach limits the set of objects a client can access in the scope of a single transaction to the size of swap space of the processor on which the application is running. It also relies on the operating system (or the application programmer, by explicit deallocate object calls) to do a good job of managing physical memory.

For the benchmark, we used the default disk page size of 7.5 Kbytes (this is also the unit of transfer between a client and the server for the group object manager). Unix file systems were used to hold the database and journal files, as Ontos cannot use a raw file system.

5 Results

This section presents the results of OO7 running on three OODBMSs. In order to ensure that all three implementations were "equivalent" and faithful to the specifications of the benchmark, all three implementations were written by the authors of this paper. One interesting result of this exercise was that, despite the lack of a standard OODBMS data model/programming language, we found the features provided by all of these systems to be similar enough that implementations in one system could be ported to another fairly easily.

In addition to doing all three implementations ourselves, we took pains to configure the systems identically when running the benchmark, again in the interest of fairness. We also contacted the companies concerned and used their comments on our implementations to ensure that we were not inadvertantly misusing their systems. We gave all of the companies a March 1 deadline by which time they had to send us bug fixes and application-level comments. The results quoted below represent numbers we achieved on the systems that we had received as of March 1. We should emphasize here that the vendors have not yet had a chance to react to the added feature of varying atomic part fanouts from 3 up to 9; for the final results reported in [CDN93], all participants will be given one last chance to provide us with last minute feedback as well as any final bug fixes or soon-to-be released system enhancements. [3]

In the following, all times are in seconds.

5.1 Traversals

The OO7 traversal operations are implemented as methods of the objects in the database. A traversal navigates procedurally from object to object, invoking the appropriate method on each object as it is visited. Some of the traversals update objects as they are encountered; other traversals simply invoke a "null" method on each object.

We ran each traversal over both the "small" and "medium" single user OO7 Benchmark databases; "large" database results will be reported later (in [CDN93] if all goes well). For the "small" benchmark, each of the read-only traversals (Traversals 1, 6, and 7) were run in two ways: "cold" and "hot." In a cold run of the traversal, the traversal begins with the database cache empty (both the client and server caches, if the system supports both). We took great pains to flush all cache(s) between runs. Because of architectural implementation differences, the actual technique used varied from system to system; however, in all cases the mechanisms were tested thoroughly to confirm their effectiveness. The hot run of the traversal consists of first running a "cold" traversal and then running the exact same query three more times and reporting the average of those three runs. We also tried two ways of running the "cold" and "hot" traversal: as a single transaction, and as two separate transactions.

For the medium single user OO7 Benchmark database, we ran only the "cold" traversals, since with the medium database size, either (1) The traversal touched significantly more data than could be cached, so "cold" and "hot" times were similar, or (2) The traversal touched a small enough subset of the database that the data could be cached, in which case the "hot" time provided no information not already present in the small configuration "hot" time. Similarly, for the update traversals on both the small and medium databases, we ran only cold traversals; running multiple update traversals caused the logging traffic from one transaction to appear in the time for the next, so hot traversals provided no more information beyond that already in the cold. The effect of updating cached data is investigated in the "cu" traversal.

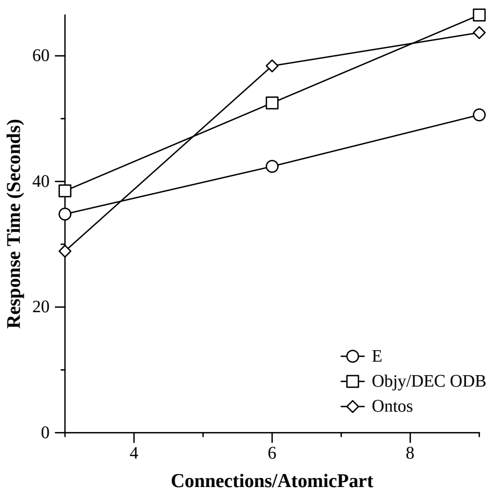

Figure 3: T1, cold traversal, small database.

We present the descriptions and results of the traversals below. Gaps in the numbering of traversals correspond to traversals that we tested, but that we either eliminated from the benchmark (because they contributed no significent new system information) or omitted from this paper due to space constraints; results from the latter traversals can be found in [CDN93].

5.1.1 Traversal #1: Raw traversal speed

Traverse the assembly hierarchy. As each base assembly is visited, visit each of its referenced unshared composite parts. As each composite part is visited, perform a depth first search on its graph of atomic parts. Return a count of the number of atomic parts visited when done.

This traversal is a test of raw pointer traversal speed, and it is similar to the performance metric most frequently cited from the OO1 benchmark. Note that due to the high degree of locality in the benchmark, there should be a non-trivial number of cache hits even in the "cold" case. Figure 3 shows the results of traversal 1 on the small database in the "cold" case.

Initially, Ontos is the fastest, followed by Exodus and then Objectivity. As the fanout is increased, Exodus scales the most gracefully. Perhaps the most interesting feature here is the discontinuity in Ontos between fanout of 3 and fanout of 6. We are unsure of the reason for

[3] To ensure that our results reflect the performance that each tested system is capable of, we chose to allow all vendors to provide pre-released versions of their systems (i.e., versions where known problems in the corresponding product releases have been fixed). We required each vendor to certify that all changes have been accepted internally for their next release and to estimate the date of that release.

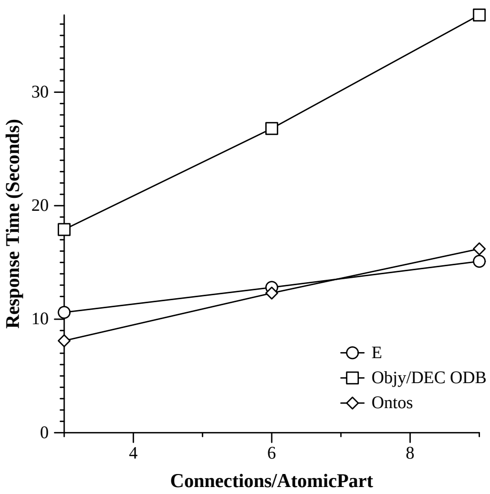

Figure 4: T1, hot traversal, small database.

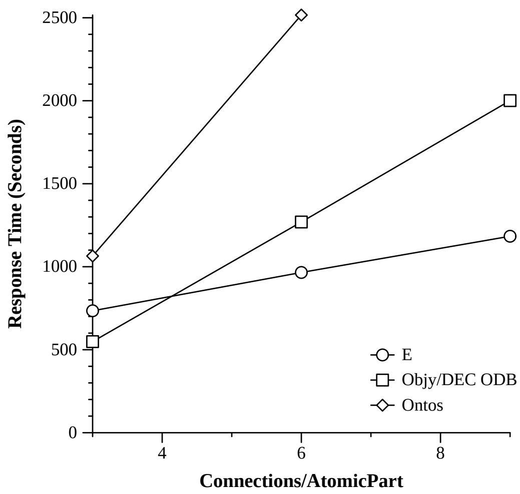

Figure 5: T1, cold traversal, medium database.

this, but it could be due to a discontinuity in how the association that represents the outgoing connections for the atomic parts grows from 3 to 6.

The results from the "hot" traversal on the small database when both the "cold" and the "hot" traversal were run as a single transaction appear in Figure 4. Both Ontos and Exodus employ software swizzling schemes that allow them to have fast "hot" times. Objectivity does no swizzling, and its "hot" time is slower as a result.

The following table compares the performance of the systems on the hot traversal with the cold and hot traversals run as a single transaction ("one") and as multiple transactions ("many"). These traversals were run over the small database with fanout 3.

	Exodus	Objy/DEC ODB	Ontos
one	10.6	17.9	8.1
many	13.8	22.6	21.2

Here we see the benefit of client caching. Exodus can cache data in the client between transactions, so its multiple-transaction hot times are close to those of the single transaction case. Objectivity and Ontos can cache data in the client between transactions if one uses special forms of commit ("CommitAndHold" in Objectivity, "KeepCache" in Ontos) but we did not feel that using these protocols was justified, since in both systems you must retain locks (which the server has no way of breaking) on cached data to keep the data consistent. (Exodus caches data but reaquires locks from the server before it is re-accessed.) Both Ontos and Objectivity do benefit from server caching — in the server buffer pool for Ontos, and in the NFS cache (on the client workstation, which helps some) for Objectivity — in this test. Since this caching effect is duplicated in every operation of the benchmark, when discussing subsequent results for read-only queries we only report on cold and hot times that were run as a single transaction.

Figure 5 shows the cold times of the systems on the medium database. For Ontos, we present only the times for the fanout 3 and fanout 6 databases because a bug prevented us from generating fanout 9. Ontos sent us a simple bug fix for this problem, but it arrived after our March 1 deadline, so in keeping with our stated policies we do not include the fanout 9 numbers in this paper. (The numbers will appear in [CDN93].) We should mention that Objectivity and Ontos both were surprisingly reliable in our experience.

The most interesting performance change for the medium database (versus the small database) is the performance of Ontos. The medium database is significantly larger than client memory, and Ontos copies objects from the database into virtual memory, so its performance degradation is likely a result of paging the client memory image. (We did not experiment with Ontos's explicit virtual memory allocate/deallocate facilities.) Comparing the other systems, Objectivity starts out faster than Exodus, probably because of the efficiency of using NFS reads versus TCP/IP messages [DFMV90], but the time for Exodus increases much more gradually as a function of the fanout.

5.1.2 Traversal #2: Traversal with updates

*Repeat Traversal #1, but update objects during the traversal. There are three types of update patterns in this traversal. In each, a single update to an atomic part consists of swapping its **(x,y)** attributes The three types of updates are: (1) Update one atomic part per composite part. (2) Update every atomic part as it is encountered. (3) Update each atomic part in a composite part four times. When done, return the number of update operations that were actually performed.*

The following table gives the results of T2ABC on the small fanout 6 database.

t2	Exodus	Objy/DEC ODB	Ontos
t2A	43.9	80.5	75.2
t2B	46.3	73.6	71.3
t2C	47.5	75.2	71.0

Comparing the t2 times to the corresponding t1 times (42.4, 53.5, and 58.4, respectively), it is evident that the update overhead in Exodus is significantly lower than that of the other systems, with Objectivity having the largest update overhead due to the cost of NFS writes. This is likely due to the fact that Exodus logs changed portions of objects, not entire pages, thereby generating much less recovery data to be written. In terms of trends, Exodus logs changes at the object level, so the Exodus time increases somewhat from t2A to t2B since the number of updated objects increases; there is little change in going to t2C because only the last change is logged in the case of repeated updates to a cached object. The other two systems both do page-level logging on these operations, so their times are relatively independent of the number of objects updated per page and the number of updates per object. (Note that the set of atomic parts associated with a given composite part is small enough here that they can be placed on a single page.)

For the medium fanout 6 OO7 database, we obtained the following results.

t2	Exodus	Objy/DEC ODB	Ontos
t2A	1000.8	1468.8	2101.2
t2B	1227.9	2199.4	2277.4
t2C	1212.0	2107.4	2243.4

Here, the update time increases when going from t2A to t2B, while no such increase was observed for the small database. This is because in the medium and large databases, the atomic parts within a composite part can span several pages; thus, moving from t2A to t2B leads to more page updates. Moving from t2B to t2C again produces no such effect, as all pages (and of course objects, which is what matters to Exodus) that t2C updates are also updated by t2B. Again, a comparison of the corresponding t2 and t1 times shows that Exodus does relatively well, while Objectivity's update overhead is significant due to the high cost of NFS writes; the performance of Ontos is suffering here due to paging, as discussed earlier.

5.1.3 Traversal #3: Traversal with indexed field updates

Repeat Traversal #2, except that now the update is on the date field, which is indexed. The specific update is to increment the date if it is odd, and decrement the date if it is even.

The goal here is to check the overhead of updating an indexed field. This should be done using the same three variants used in Traversal #2, and again the number of updates should be returned at the end.

For the small OO7 fanout 6 database, we obtained the following results.

t3	Exodus	Objy/DEC ODB	Ontos
t3A	48.9	79.8	79.6
t3B	100.8	140.5	118.7
t3C	244.6	309.9	214.9

In our implementations, only Objectivity used automatic index maintenance. The Exodus and Ontos numbers reflect the overhead of explicit index maintenance coded by hand in those systems. Ontos does provide implicit index maintenance, although we did not use this feature in the tests presented in this paper. The main point to notice here is that the systems are unable to "hide" the multiple index updates within a single log record, since every update (even a repeated update to an object or page) must also update the index. This is why we see an increase from t3B to t3C that we didn't see from t2B to t2C.

For the medium OO7 database, again fanout 6, we obtained the following results.

t3	Exodus	Objy/DEC ODB	Ontos
t3A	1083.9	1389.7	6467.3

5.1.4 Traversals #8 and #9: Operations on Manual.

Traversal #8 scans the manual object, counting the number of occurrences of the character "I." Traversal #9 checks to see if the first and last character in the manual object are the same.

For the medium OO7 database (1M byte manual), we obtained the following cold times. These results were independent of the atomic part fanout.

	Exodus	Objy/DEC ODB	Ontos
t8	12.3	11.5	5.5
t9	0.2	11.0	4.8

Ontos has extremely fast t8 times, perhaps due to the fact that in Ontos persistent character data can be stored and processed just like (non-persistent) C "char *" attributes of objects. Exodus has a fast t9 time because it is able to page large objects, hence t9 reads only the first and last page of the manual, whereas both Objectivity and Ontos must read in the entire manual.

5.1.5 Traversal CU (Cached Update)

Perform traversal T1, followed by T2A, in a single transaction. Report the total time minus the T1 hot time minus the T1 cold time.

The goal of this traversal is to investigate the performance of updates to cached data. The original T1 traversal warms the cache; the T2A traversal updates some of the objects touched by T1. The time to report is defined in such a way as to isolate the time for the updates themselves (and the associated log writes); recall that T2A is like T1 except for the updates to some atomic parts.

For the small fanout 6 database we obtained the following times:

	Exodus	Objy/DEC ODB	Ontos
cu	0.9	20.9	10.4

Exodus does very well, again because it writes log records rather than shadowing or logging updated pages, and many of the log records generated should fit on a single log page. Objectivity suffers from its need to do an NFS write per updated page.

5.1.6 Traversals Omitted

We ran a number of traversals for which, due to space constraints, the results will appear in [CDN93] rather than in this paper.

One of the most interesting omitted traversals was Traversal #6, which is the same as Traversal #1 in the assembly hierarchy, but instead of performing a depth first search on all the atomic parts in each composite part, Traversal #6 merely visits the root atomic part in each composite part. This test coupled with traversal #1 provides interesting insight into the costs and benefits of the full swizzling approach to providing persistent virtual memory. Unfortunately, after ODI withdrew from the benchmark, this test became less interesting.

We also experimented with traversals that changed the size of document objects, traversals that scanned documents instead of traversing atomic part subgraphs, and "reverse-traversals" that go from an atomic part to the root of the assembly hierarchy.

5.2 Queries

The queries are operations that ideally would be expressed as queries in a declarative query language. Not all of the OO7 queries could be expressed entirely declaratively in all of the systems; whenever a query could not be expressed declaratively in a system we implemented it as a "free" procedure that essentially represents a hand coded version of what the query execution engine would do in order to evaluate the query.

5.2.1 Query #1: exact match lookup

Generate 10 random atomic part id's; for each part id generated, lookup the atomic part with that id. Return the number of atomic parts processed when done. Note that this is like the lookup query in the OO1 Benchmark. On the small database, fanout 6, we obtained the following numbers.

q1-one	Exodus	Objy/DEC ODB	Ontos
cold	0.7	8.4	2.4
hot	0.008	0.05	0.005

In Exodus, the query was hand-coded to use a B+ tree index. In Objectivity, the query was hand-coded to use a "hashed container," which essentially gives a key index that can be used for exact-match lookups like query #1. Exodus appears to have the most efficient index lookup implementation, by a significant margin, followed next by Ontos.

On the medium fanout 6 database:

q1	Exodus	Objy/DEC ODB	Ontos
cold	0.8	9.6	9.7

In each case, the systems used the index to avoid scaling the response time with the database; the relative ordering of the systems' performance is consistent with that of the small results.

5.2.2 Queries #2, #3, and #7.

These queries are most interesting when considered together:

- *Query #2: Choose a range for dates that will contain the last 1% of the dates found in the database's atomic parts. Retrieve the atomic parts that satisfy this range predicate.*
- *Query #3: Choose a range for dates that will contain the last 10% of the dates found in the database's atomic parts. Retrieve the atomic parts that satisfy this range predicate.*
- *Query #7: Scan all atomic parts.*

Note that queries #2 and #3 are candidates for a B+ tree lookup. On the medium fanout 6 database we obtained the following "cold" numbers.

	Exodus	Objy/DEC ODB	Ontos
q2	19.1	37.1	39.5
q3	35.0	129.4	63.0
q7	31.8	136.3	52.6

In Exodus, this query was implemented as a hand-coded B+tree lookup. In Objectivity, it was not necessary to hand code this scan — the query was implemented by using an Objectivity iterator with a selection predicate. The Ontos times shown are again for a hand-coded index (B+ tree, in this case) lookup. The index implementation insights from q2 and q3 are fairly consistent with the results of q1; comparing these times to q7, it is clear that for a selectivity of 10%, it is as fast or faster in these systems to scan the entire atomic part set than to use the B+trees.[4]

[4] It was our intent to generate a case where a sequential scan is clearly superior to an unclustered index scan in q3, providing a chance to test the cost-evaluation intelligence of the query optimizer in any OODBMS that supports queries. The size of the q2/q3 ranges will be adjusted accordingly in [CDN93].

5.2.3 Queries Omitted

Due to space constraints, we have omitted most of the OO7 queries from this paper, including a path-join query, a "single-level make" query, and an ad-hoc join query. The results from these queries will appear in [CDN93]. Our original query set also included a "transitive make" query, which we dropped; the results from that query failed to provide any unique insights relative to the other OO7 operations.

5.3 Structural Modification Operations

Again due to space constraints, in this paper we omit the results from an insert operation (insert five new composite parts) and a delete operation (delete five composite parts).

6 Conclusion

The OO7 Benchmark is designed to provide a comprehshensive profile of the performance of a OODBMS. It is more complex than the OO1 Benchmark and more comprehensive than both the OO1 and HyperModel Benchmarks; however, the results of our tests indicate that the added complexity and coverage has provided a significant benefit, as the OO7 test results reported in this paper (and observed in the tests that were not reported for space or legal reasons) exhibit system performance characteristics that could not have been observed in OO1 or HyperModel.

OO7 has been designed from the start to support multiuser operations. While these operations are not yet implemented, the database structure as described in this paper already provides the framework in which to construct a multiuser benchmark. Specifically, the modules and assembly structure, with their "shared" and "private" composite parts, will allow us to precisely vary degrees of sharing and conflict in multiuser workloads. In future work, we will be refining and experimenting with these multiuser workloads to investigate the performance of OODB systems' concurrency and versioning facilities. We also plan to add structural modifications that test the ability of an OODBMS to maintain clustering in the face of updates.

Acknowledgment

Designing OO7 and getting it up and running on all the systems we tested was a huge task that we could not have completed without a lot of help. We would like to thank Jim Gray, Mike Kilian, Ellen Lary, Pat O'Brien, Mark Palmer, and Jim Rye of DEC for initial discussions that led to this project, and for useful feedback as it progressed. Rick Cattell shared his thoughts with us early on about what he would change in a successor to OO1, and gave us some feedback on our design. Rosanne Park and Rick Spickelmier at Objectivity, Gerard Keating and Mark Noyes at Ontos, and Jack Orenstein and Dan Weinreb of ODI were extremely helpful in teaching us about their systems and debugging our efforts. Joseph Burger, Krishna Kunchithapadam, and Bart Miller helped us track down a strange interaction between one of the systems and our environment. Foley, Hoag, and Eliot kept our FAX machine humming and our mailboxes full. Finally, we would like to give a special thanks to three staff members at the University of Wisconsin — Dan Schuh, C. K. Tan, and Mike Zwilling — for their help in getting the testbed up and running.

References

[And90] T. Anderson et al. The HyperModel Benchmark. In *Proc. EDBT Conf.*, March 1990.

[CDN93] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. CS Tech Report, Univ. of Wisconsin-Madison, April 1993.

[CS92] R. Cattell and J. Skeen. Object operations benchmark. *ACM TODS*, 17(1), March 1992.

[DD88] J. Duhl and C. Damon. A performance comparison of object and relational databases using the sun benchmark. In *Proc. ACM OOPSLA Conf.*, Sept. 1988.

[DFMV90] D. J. DeWitt, P. Futtersack, D. Maier, and F. Velez. A study of three alternative workstation-server architectures for object-oreinted database systems. In *Proc. VLDB Conf.*, Aug. 1990.

[EXO92] The EXODUS Group. Using the EXODUS storage manager V2.0.2. Technical Documentation, Jan. 1992.

[FZT+92] M. J. Franklin, M. J. Zwilling, C. K. Tan, M. J. Carey, and D. J. DeWitt. Crash recovery in client-server EXODUS. In *Proc. ACM SIGMOD Conf.*, June 1992.

[Gra81] Jim N. Gray et al. The recovery manager of the System R database manager. *ACM Comp. Surveys*, June 1981.

[RC89] J. E. Richardson and M. J. Carey. Persistence in the E language: Issues and implementation. *Software Practice and Experience*, Dec. 1989.

[RKC87] W. Rubenstein, M. Kubicar, and R. Cattell. Benchmarking simple database operations. In *Proc. ACM SIGMOD Conf.*, May 1987.

[SCD] D. Schuh, M. Carey, and D. DeWitt. Implementing persistent object bases: Principles and practice. In *Proc. 4th Int'l Workshop on Persistent Object Systems.*

[WC92] M. Winslett and S. Chu. Database management systems for ECAD applications: Architecture and performance. In *Proc. NSF Conf. on Design and Manufacturing Systems*, Jan. 1992.

[WD92] S. White and D. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proc. VLDB Conf.*, Aug. 1992.