The OO7 Benchmark*

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

Version of April 12, 1993

Abstract

The OO7 Benchmark represents a comprehensive test of OODBMS performance. In this report we describe the benchmark and present performance results from its implementation in three OODB systems. It is our hope that the OO7 Benchmark will provide useful insight for end-users evaluating the performance of OODB 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 (OODBMS) 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 OODBMSs 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;

*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.

- 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. Since at this point there is not any consensus on what constitutes the canonical OODBMS application, and in fact there is growing evidence that there is no such "canonical OODBMS" application, we think it would be a mistake at this point to try to build a single number benchmark.

In this paper, we describe the OO7 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)[1]. 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, 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 results from the paper or else face possible legal action. (See the README file in the OO7 directory of `ftp.cs.wisc.edu` for more details.) It is unfortunate that they chose to withdraw, as ODI's approach to persistence provided some interesting contrasts with the other systems.

The timings presented in this paper present interesting information about the performance of the systems that we tested, most of which was unavailable before this comparison. In addition to providing insight into the performance of existing OODBMSs, we hope that in the future OO7 will provide a rich source of test workloads for the OODBMS research community. We are already seeing early signs that OO7 will be useful in this context, as it has managed to uncover correctness and/or performance problems in at least one version of every system that we have run the benchmark on so far. In fact, at least one of the companies that participated in the benchmark now plans to use the OO7 Benchmark as part of the performance and correctness tests in the system validation suite for future releases of their product.

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.

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

2 Related Work

2.1 The OO1 Benchmark

The OO1 Benchmark [CS92][2], commonly referred to as the Sun Benchmark, was really the first "standard" benchmark that attempted to predict DBMS performance for engineering design applications. Its intent was to measure performance for navigation and simple updates, and as such it was primarily intended to capture the cost of database interactions and the presence and the effectiveness of client caching. The specification of OO1 was based on its designers' experience with an initial "simple database operations" benchmark [RKC87] that they had implemented and used at Sun. Some operations in the initial benchmark were found to be highly correlated with others; e.g., small range queries added little insight beyond that obtained from exact-match lookups, so they were eliminated in OO1. Others were combined into more general operations in OO1; e.g., several relationship operations in the initial benchmark were subsumed by a more general traversal operation in OO1. Because of its early visibility and its simplicity, OO1 has became a de facto standard for OODB benchmarking. As such, it has been run on many of the current OODB products [CS92].

The database that the OO1 Benchmark is based upon consists of part objects and connections between them. Each part has five data fields: a part id, a type, an (x,y) coordinate pair, and a build date. Each part has exactly three out-going ("to") connections to other parts plus a variable number of incoming ("from") connections, and each connection has a type and a length. To provide a notion of locality in the object graph, OO1 parts are logically ordered by part id, and the "to" connections for each part are chosen so that each connection has a 90% chance of referencing a "nearby" part. The OO1 definition of a "nearby" part is one within $\pm 1\%$ of the part id space. There are two database sizes against which all OO1 Benchmark operations are run — one with 20,000 parts, to model applications whose working sets fit in memory, and one ten times larger, with 200,000 parts, to model those applications whose data sets exceed physical memory. (There is also an optional "huge" database with 2,000,000 parts.)

The OO1 Benchmark itself consists of three operations. The first is a part "lookup" operation, which looks up 1,000 random parts by their part ids. The second is an object graph "traversal" operation, which accesses 3,280 connected parts by selecting a random part and then performing a seven-level depth-first traversal (with multiple visits allowed) of the parts reachable from there.[3] The third OO1 Benchmark operation is an *insert* operation that adds 100 new parts to the database. The benchmark specification calls for two elapsed time measures, "cold" and "warm", for each operation. Each operation is repeated ten times; the "cold" time is the time required for the first iteration, reflecting the elapsed time starting with an empty cache, while the "warm" time measures the case where the cache has been fully initialized.

[2] Object Operations, version 1.

[3] A reverse traversal is also part of the benchmark, but it is not emphasized in terms of evaluating OODB system performance.

2.2 The HyperModel Benchmark

Another benchmark directly related to OO7 is the HyperModel Benchmark developed at Tektronix [And90]. Starting with the initial Sun simple database operations benchmark, the developers of HyperModel set out to develop a more comprehensive engineering DBMS performance test suite based a hypertext application model. Compared to OO1, Hypermodel includes both a richer schema and a wider range of operations.

The HyperModel database, like OO1, is a graph of interconnected nodes. Unlike OO1, which has just one relationship between nodes (i.e., the M:N connections), nodes in HyperModel participate in several relationships. The HyperModel nodes participate in one hierarchical (1:N) relationship and two M:N relationships. And, unlike OO1, where there is just one type of node, HyperModel databases include three types of nodes. Viewed from the 1:N relationship, which forms a tree with a five-way fanout, a HyperModel database includes $k-1$ levels of non-leaf nodes plus a level of leaf nodes. A non-leaf node holds several integer values, while leaf nodes contain either a small (10-100 byte) text string or a bitmap of moderate size (100×100 up to 400×400). The database is parameterized by the number of levels, which is set to 4, 5, and then 6 for running benchmark operations, and there is a 125:1 ratio of text leaf nodes to bitmap leaf nodes. One of the two M:N relationships is created by randomly interconnecting nodes from adjacent levels of the 1:N hierarchy; the other, intended to model hypertext links, randomly interconnects nodes throughout the hierarchy.

The HyperModel Benchmark consists of several different groups of operations, including exact-match lookups (by integer attribute value and OID), range queries (1% and 10%), group lookups (which each follow a relationship from a random node to its directly related nodes), reference lookups (which do the inverse), a sequential scan, a set of "closure" operations, and some editing operations. HyperModel's closure operations were added in response to the fact that, unlike OO1, the initial Sun Benchmark had no traversals, which are felt to be important in engineering applications. Each of the HyperModel closure operations performs a reachability traversal, starting at a randomly chosen node at level three of the 1:N hierarchy, via one of the three relationships. Several gather lists of the OIDs encountered, several compute aggregate values (a sum or count), one applies a predicate to avoid traversals from a small subset of the objects, and one modifies a field of the visited objects. The two editing operations update either the text string field or the bitmap field of the leaf nodes, respectively. Like OO1, Hypermodel requests both cold and warm times for each operation.

2.3 Other OODB Benchmarking Work

There are several other OODB studies that are related to our work on OO7. Engineers at Ontologic used the initial Sun Benchmark to study the performance of Vbase, their first OODB product offering [DD88]. Their reflections on the work provided a useful summary of some of that benchmark's shortcomings, including its lack of an opportunity for semantic object clustering (discussed further below), its very simple schema, and its lack of complex operations such as traversals. Also in this area, researchers at Altair designed a complex object benchmark (ACOB) for use in studying alternative client/server process architectures [DFMV90]. Unlike previous OODB benchmarks, a notion of complex objects was included in the design; however, only a small number of traversal and update operations were involved, as these were sufficient to expose the tradeoffs among the software architectures of interest. Finally, Winslett and Chu recently studied OODB (and relational DB) system performance by porting an actual VLSI layout editor (the MAGIC editor from UC Berkeley) onto several systems [WC92]. However, since only the file I/O portions of MAGIC were modified, this work focused on the systems' save/restore performance rather than their performance when applications are operating on database objects.

2.4 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? Basically, as mentioned briefly in the Introduction, none 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. OO1 is an excellent test of OODB performance on simple navigational and update tasks, as intended, but there are a number of application and system characteristics that it does not measure. For example, while complex objects are important in many OODB applications, OO1 has no real notion of complex objects. OO1 models inter-object reference locality via its use of "nearby" objects during database generation, but this has two problems. First, $\pm 1\%$ of a large database is not very local; more seriously, while a given object tends to reference its neighbors, this is quite different than semantically based complex objects (where a group of objects are used as one aggregate object, providing a natural unit of clustering). OO1 also does not examine the density of traversals (i.e., the fraction of objects accessed per page), traverals that involve updates, object queries, and various other potentially important application and/or OODB system features.

HyperModel attempted to correct a number of the Sun (and indirectly, OO1) Benchmark's shortcomings. Its designers clearly succeeded in developing a more complex benchmark, particularly with respect to the set of operations tested. What is less clear is how much of the added complexity is actually worthwhile in terms of insights provided by the benchmark. Despite a more complex schema, there is still no semantic notion of complex objects in the HyperModel design. While HyperModel includes different read-only traversal operations, they seem to have been designed mainly to be different from one another, as opposed to probing the design/performance space of OODB systems. (In fact, the performance results in [And90] were presented as an appendix, with no attempt being made to draw insights or conclusions from the results.) No tests are included in HyperModel for object queries, updates to indexed vs. non-indexed object attributes, repeated object updates, the impact of transaction boundaries, or various other performance-related OODB features that OO7 has been designed to test. Finally, HyperModel is difficult to implement consistently from its published specifications.[4]

As will be described in the next section, we tried to be methodical and broad, without going completely overboard in the area of complexity, when designing OO7. To guide the design process, we began by surveying the various features (both functional and performance-oriented) found in current OODB products. We then made a list of those features and techniques that seemed likely to have a significant impact on relative performance, and we proceeded to develop a series of tests to cover these features and to distinguish among the techniques use by different vendors to implement them. While doing this, we were also influenced significantly by the philosophy underlying the Wisconsin Benchmark for relational database systems [BDT83]; we wanted OO7 to be a relatively broad stress test for OODB systems, both in terms of function and system performance.

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`. The informal description of the benchmark given here should suffice for understanding the basic results; anyone planning to implement the benchmark should obtain a copy of one of the available implementations.

In any benchmark, there are many opportunities for "cheating" by implementing hacks that follow the letter of the benchmark specification without following the intentions of the benchmark designers. Many of these opportunities are only discovered when companies begin "tuning" the implementations of the benchmark, at which point benchmark designers typically take steps to disallow the "cheating" in the benchmark specification. At this point we have not tried to write a bullet-proof specification that prevents cheating. We expect that in the future, when we gain experience from watching others implement the benchmark, the specification will evolve as necessary.

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.

Appendix A gives a DDL description of the OO7 Benchmark database and a corresponding ER diagram. The interested reader may wish to consult Appendix A while reading the following description of the database.

[4] To be fair, completely specifying a complex benchmark is a very difficult task. As mentioned elsewhere, our approach to overcoming this difficulty mainly consists of making several reference implementations of OO7 readily available.

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.

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 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 5.2 through 5.4.) The `buildDate` values in atomic parts are randomly chosen in the range `MinAtomicDate` to `MaxAtomicDate`, which is currently 1000 to 1999. 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.

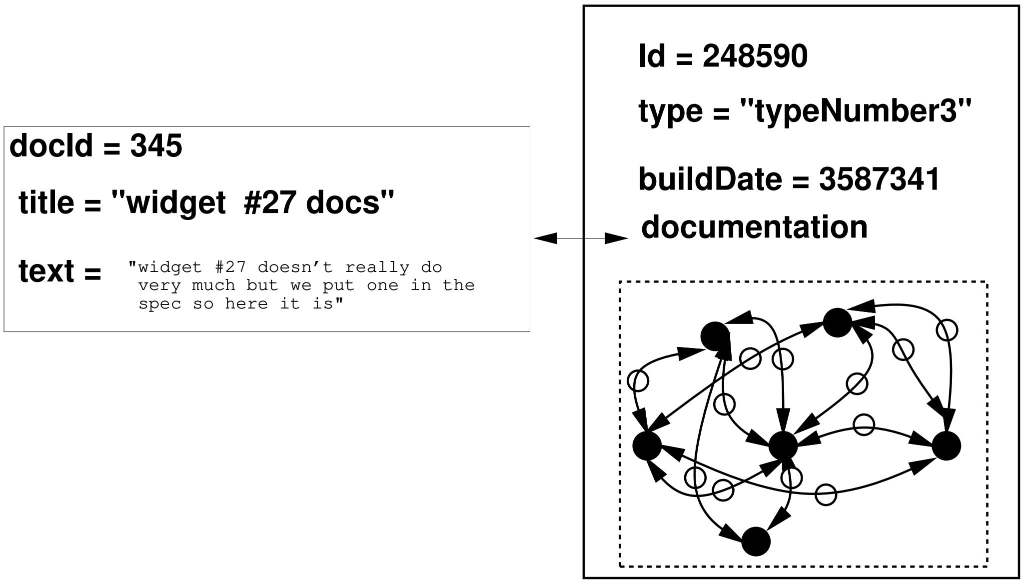

Figure 1: A Composite Part and its associated Document object.

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

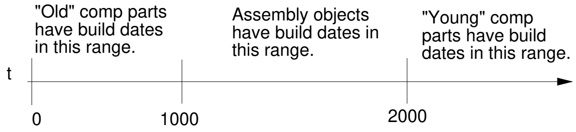

Figure 2: Build dates for benchmark objects.

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. (The sharing is on a module basis. If a composite part is referenced as a private composite part of a base assembly from module i, then it is not referenced as a private composite part by a base assembly of any module other than module i.) 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.

The relationship between the build dates in composite parts and base assemblies is important in one of the queries of the benchmark (Query 5, which finds all base assemblies that reference a composite part with a more recent `buildDate` than the build date in the base assembly). We control this relationship as follows: All base assemblies have `buildDate` chosen randomly in the range `MinAssmDate` to `MaxAssmDate` (currently 1000 to 1999). Composite parts are divided into two categories. "Young" composite parts have build dates chosen randomly in the range `MinYoungCompDate` and `MaxYoungCompDate` (currently 2000 to 2999), whereas "old" composite parts have build dates chosen randomly in the range `MinOldCompDate` to `MaxOldCompDate` (currently 0 to 999). The percentage of "young" versus "old" composite parts is controlled by the paramenter `YoungCompFrac`, with the interpretation that 1 out of `YoungCompFrac` composite parts are "young," while all other composite parts are "old." Currently `YoungCompFrac` is set to 10. Figure 2 gives a timeline showing the relationship between the build dates for old composite parts, assembly objects, and young 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 3 depicts the full structure of the single user OO7 Benchmark Database. Note that the picture is somewhat misleading in terms of both shape and scale; the actual assembly fanout used is 3, and there are only $(3^7 - 1)/2 = 1093$ 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.

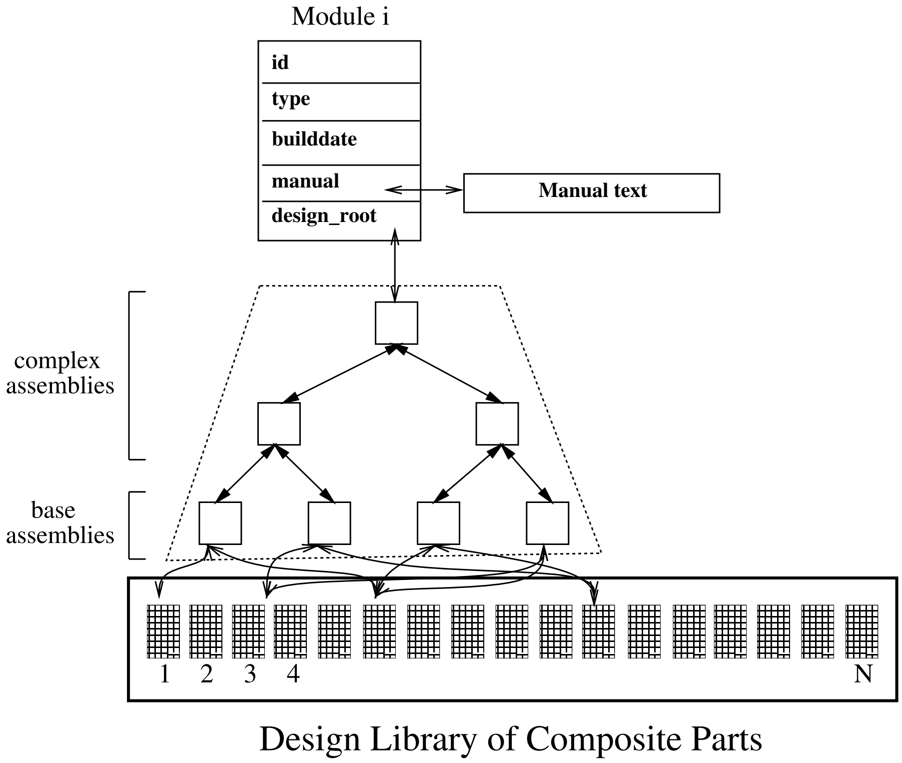

Figure 3: Structure of a module.

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. If the server cannot satisfy the request from its buffer pool, a disk I/O is initiated by invoking a disk process to perform the actual I/O operation. After the disk process has read in the page, the server process returns it to the requesting client process and keeps a copy in its own buffer pool.

The ESM also provides concurrency control and recovery services. Locking is provided at the page and file levels (all the normal modes) with a special non-2PL protocol for index pages. Recovery is based on logging the changed portions of objects [FZT+92]. Pages in the client buffer pool are cached across transaction boundaries, but locks must be reacquired from the server when cached pages are first used in subsequent transactions.

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.

The current version of E is based on the GNU g++ 2.3.1 compiler. E uses the ESM for storing persistent objects. Operations on persistent objects are compiled into instructions for a virtual machine that are interpreted at runtime [SCD]; the current version of this interpreter is EPVM 3.0. EPVM 3.0 stores memory-resident persistent objects in the buffer pool of the ESM client process; pointers between such memory-resident objects are swizzled (and tested) in software as they are traversed, as was done in EPVM 2.0 [WD92]. When the ESM decides to replace a page, all pointers are unswizzled; if the page is dirty, log records for its updated objects are generated and sent to the server along with the page.

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.

Recovery is implemented via shadows [Gra81]. During the course of a transaction, updates are written to a shadow database. At commit time, these updates are applied carefully to the actual database with a journal being used to recover in the event that the commit fails. If the transaction aborts, the shadow database is simply deleted.

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 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.

Recovery is via REDO logging. During the course of a transaction, all updates are buffered in virtual memory. When the transaction commits, the updates are then written to one or more journal files on the server. Once the journal files have been successfully flushed to disk, the updates are applied to the actual database.

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.

When an Ontos transaction commits, the application is given the choice of keeping its cache intact or flushing the objects from virtual memory. For the experiments that tested the ability of a system to cache objects across multiple transactions we did not use the "keep cache" option because this mode does not insure that the cached data is consistent with respect to transactions running elsewhere. A solution that we did not have time to investigate would have been to use notify locks.

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.[5]

The results presented here are for the "small" and "medium" single user OO7 Benchmark databases. "Large" database results will be reported at a later date if all goes well.

In the following, all times are in seconds.

5.1 Database Sizes

In the following, the size of the databases in each of the systems is useful for intepreting the results. For the small databases, we measured the following sizes:

Fanout	Exodus	Objy/DEC ODB	Ontos
3	11.5M	5.7M	4.2M
6	13.4M	7.7M	4.4M
9	15.9M	10.1M	4.9M

The Exodus DB sizes are large because of the size of OID's that Exodus uses (12 bytes). The OO7 database contains a lot of pointers; each of these pointers is stored as an OID. Ontos has extremely small database sizes, while Objectivity is between the two.

For the medium databases we measured the following sizes:

Fanout	Exodus	Objy/DEC ODB	Ontos
3	103.8M	54.6M	51.7M
6	125.6M	74.9M	122.3M
9	152.6M	98.9M	

[5]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.

The same trends hold as for the small database sizes except that Ontos now grows dramatically moving from fanout 3 to 6. We are unsure of why this occured. 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. We should mention that Objectivity and Ontos both were quite robust in our experience.

5.2 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" OO7 Benchmark databases. For the "small" benchmark, each of the read-only traversals (Traversals 1, 6-9) 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.

We considered requiring a third class of traversal, "warm," in which the cache is "warmed" by first running a traversal that is similar but not identical to the "warm" traversal. However, since warm performance is just a function of cold and hot performance ("cold" when the warm traversal misses in the cache, "hot" otherwise) we decided that "cold" and "hot" succinctly provided the most important information. The two approaches produced different results since the systems tested differ in their ability to cache objects across transactions. This effect is examined for traversal one in Subsubsection 5.2.1; in the other traversals, in the interest of space we include only the times for running the two traversals as a single transaction.

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.

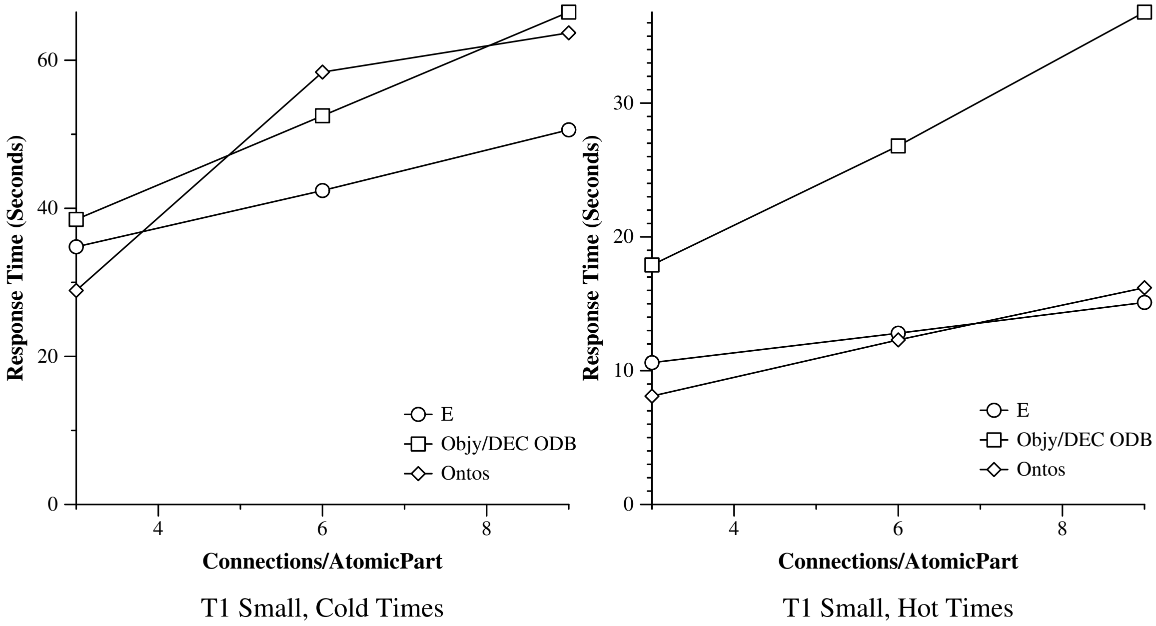

Figure 4: T1 traversal, small database.

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 eliminated from the benchmark because they contributed no significent new system information. The traversals are not presented in numeric order, as we felt the order used here would make the exposition clearer.

5.2.1 Traversal T1: 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. The left graph in Figure 4 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 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 the right graph in Figure 4. Both Ontos and Exodus employ software swizzling schemes that allow them to have fast "hot" times. Objectivity does less effective 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 and Objectivity can cache data in the client between transactions, so its multiple-transaction hot times are close to those of the single transaction case. Ontos can cache data in the client between transactions if one uses special forms of commit ("KeepCache") but we did not feel that using these protocols was justified, since KeepCache requires that the client retain locks (which the server has no way of breaking) on cached data to keep the data consistent. Exodus and Objectivity cache data but reaquire locks from the server before it is re-accessed. Ontos also provides a similar facility using "notify locks", but we did not experiment with this facility in our OO7 implementation. In our implementation, Ontos does benefit from caching in the server buffer pool. 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. 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 due to paging of the client memory image. (We did not experiment with Ontos's explicit virtual memory allocate/deallocate facilities.) This degradation due to paging will also be seen again later, in some of the the other medium database operations, for Ontos. 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.

5.2.2 Traversal T6: Sparse 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, visit the root atomic part. Return a count of the number of atomic parts visited when done.

Note that this is like T1 except that instead of doing a DFS on all of the atomic parts in the composite part, T6 just visits one (the root part). This test coupled with Traversal T1 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. The left graph in Figure 6 gives the performance we measured for the small database cold times; the right graph in Figure 6 gives the corresponding hot times. Finally, Figure 7 gives the T6 performance on the medium database.

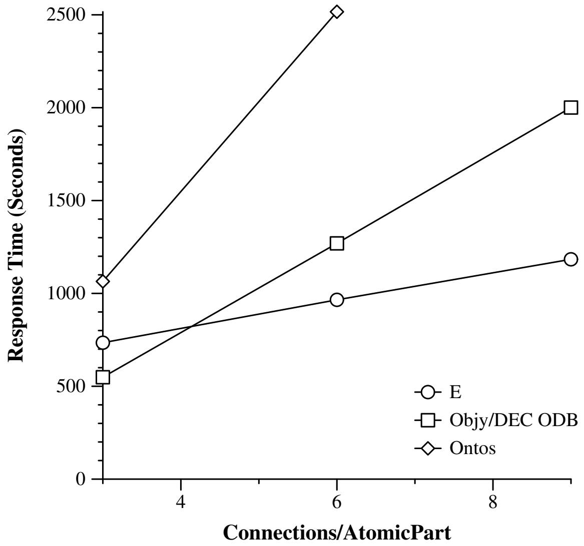

Figure 5: T1, cold traversal, medium database.

It is interesting to contrast Ontos' performance here as compared to T1. In T1, Ontos and Exodus had comparable execution times on the small, cold T1 traversal. Here, however, Ontos is significantly faster than Exodus. This is due to the way that the Ontos group storage manager lays out data on disk. In Ontos, when using the group storage manager, a single disk page may only hold objects of the same type, so (unlike in Exodus and Objectivity) atomic parts and connections are never stored on the same page. Since T6 accesses no connection objects, the Ontos data layout is clearly advantageous for T6.

5.2.3 Traversal T2: Traversal with updates

Repeat Traversal T1, 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:*

- A *Update one atomic part per composite part.*
- B *Update every atomic part as it is encountered.*
- C *Update each atomic part in a composite part four times.*

When done, return the number of update operations that were actually performed.

Figure 8 gives the results of T2A, B, and C on the small database.

In order to understand these results, we need to examine how each of the systems does commit processing. Objectivity uses a combination of shadows and logging as its recovery mechanism. When an update transaction commits, all dirty pages must be written to disk using NFS writes. Exodus and Ontos use logging instead. Ontos does full page logging, but sends the pages to the server via TCP/IP instead of using NFS to write them directly to disk. This allows the

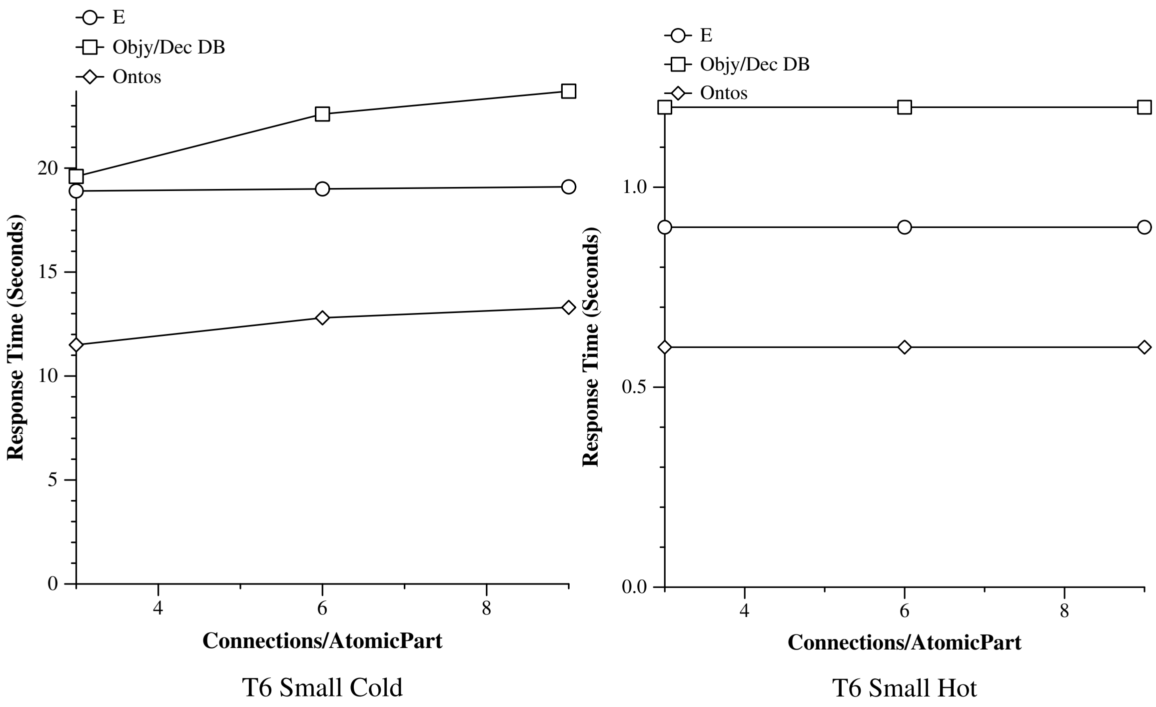

Figure 6: T6 traversal, small database.

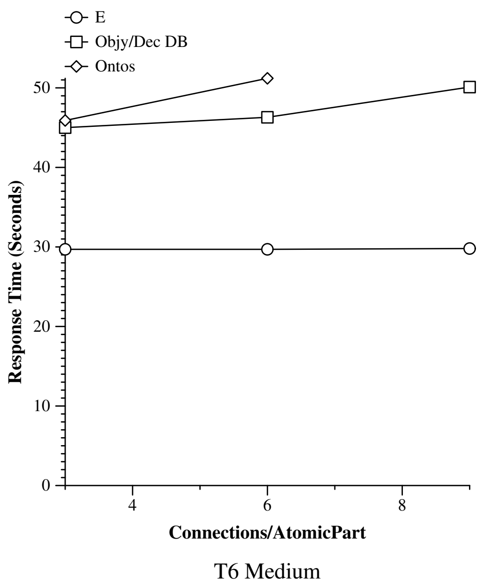

Figure 7: T6, cold traversal, medium database.

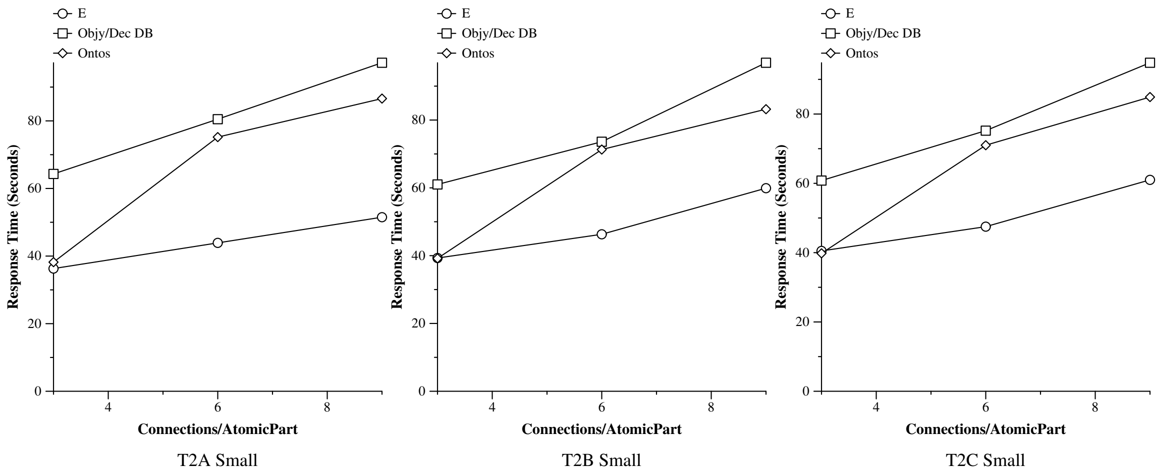

Figure 8: T2 traversal, small database.

server to overlap communications with I/O. It is interesting to compare the times of Ontos and Objectivity, since both use page-based recovery mechanisms. One way to gain insight into their relative performances is to compare the performance of the systems on T1 and on T2B, since both traversals touch exactly the same number of objects. While Objectivity and Ontos have about the same performance on T1, on T2B Ontos is always faster.

Exodus benefits in two ways on this traversal. First, like Ontos, Exodus uses a server process to which clients send dirty pages (rather than having to write them to disk). Second, unlike Ontos, Exodus uses object-level logging, which in this case cuts down on the amount of I/O that must be done at commit time (since the objects in question are significantly smaller than a page). Finally, the Exodus server "steals" I/O time by continuously flushing dirty pages to disk in the background when there is no current client I/O request. In terms of trends, 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.)

In some sense this comparison using T2ABC does not completely characterize the update story for these systems. Both Exodus and Ontos leave dirty data pages in the server's buffer pool at commit time; these pages eventually must be written to disk. Objectivity, on the other hand, forces all dirty pages back to disk. If subsequent transactions tend to rereference pages in the server's buffer pool, the Exodus/Ontos approach is clearly superior. On the other hand, if the pages are not referenced, subsequent transactions will incur the cost of writing these dirty data pages to disk in order to make room for their own pages. Using Exodus for the T2 queries we measured the difference between the time required to simply commit the transaction and the time required to commit the transaction and flush the server's buffer pool to disk to be 6 seconds (out of a total of

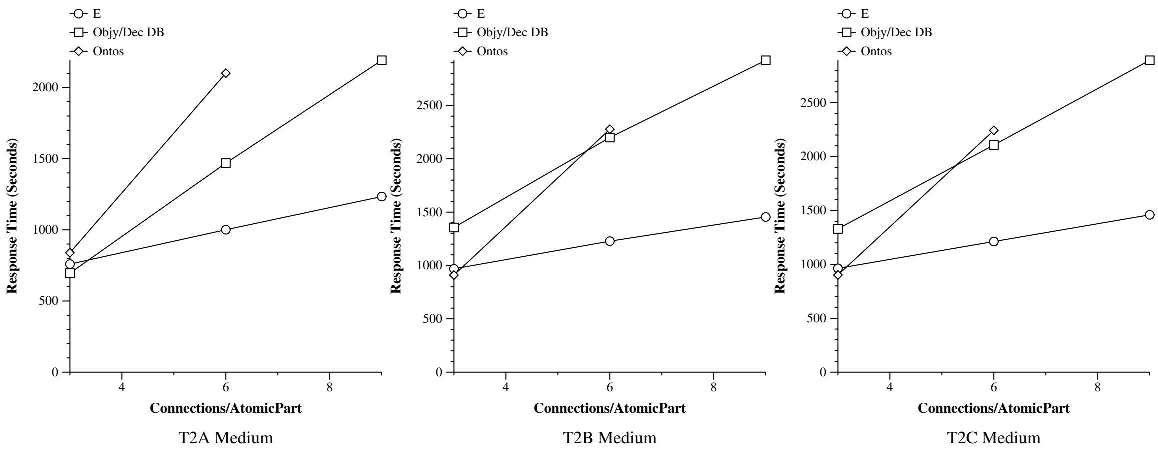

Figure 9: T2 traversal, medium database.

over 900 seconds). This low overhead is probably partially due to the fact that when running the T2 traversal the server was able to flush some dirty pages to disk in the background, concurrently with the traversal running on the client. We are considering adding a new update traversal to test this aspect of system performance (keeping dirty pages in the server cache after commit).

Figure 9 gives the results of T2A, B, and C on the medium database. Here, the update time increases when going from T2A to T2B, while little if any 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.

5.2.4 Traversal T3: Traversal with indexed field updates

Repeat Traversal T2, 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 T2, and again the number of updates should be returned at the end.

Figure 10 gives the results for T3A, B, and C on the small database.

In our implementations, only Objectivity used automatic index maintenance. (Automatic index maintenance means that the index is updated transparently as a result of a data member update.) 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

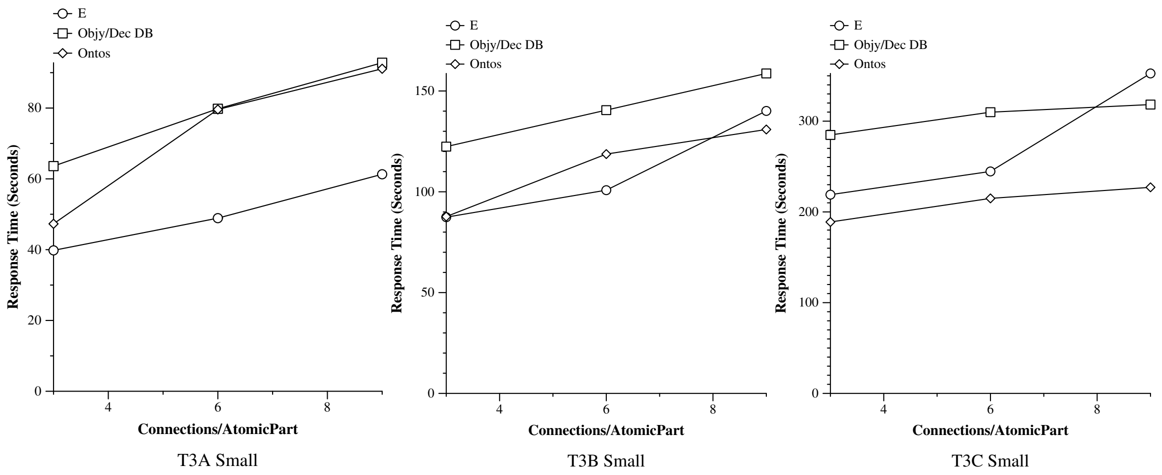

Figure 10: T3 traversal, small database.

are unable to "hide" the multiple 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.

Figure 11 gives the results for T3A on the medium database.

5.2.5 Traversals T8 and T9: Operations on Manual.

Traversal T8 scans the manual object, counting the number of occurrences of the character "I." Traversal T9 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. Also, the small OO7 database times for these traversals (100K manual) did not provide any insight not already provided by the medium OO7 database times.

	Exodus	Objy/DEC ODB	Ontos
T8	12.3	11.5	5.5
T9	0.2	11.0	4.8

Ontos has the fastest T8 times. However, none of the systems is doing particularly well at scanning large objects, as the Exodus and Objectivity times correspond to about 80 KB/sec, and the Ontos time corresponds to about 180 KB/sec. Clearly, there is room for improvement in all three systems based on these data rates. 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.

5.2.6 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.

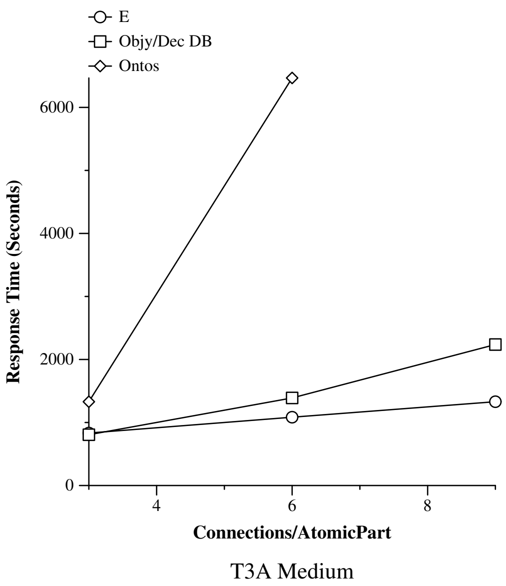

Figure 11: T3A, medium database.

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. Figure 12 gives the times we obtained on the small databases.

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 clearly suffers due to the high commit processing cost implied by its use of a server-less architecture that depends upon NFS writes.

5.2.7 Traversals Omitted

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. These tests were deleted from the final benchmark because, after experimentation, we discovered that they did not provide additional insight beyond that provided by the other traversals.

5.3 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

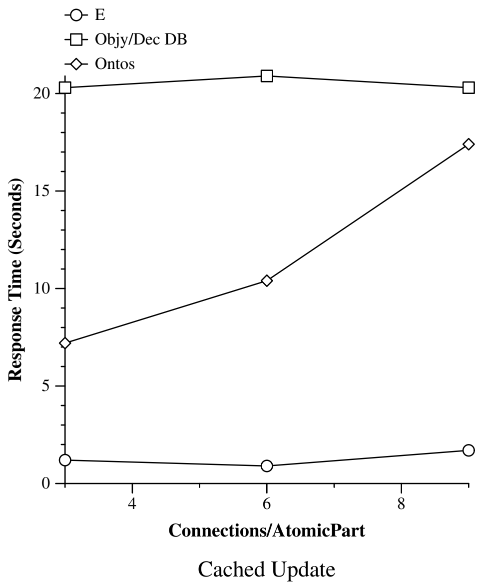

Figure 12: Cached update, small databases.

would do in order to evaluate the query.

Like the traversals, the queries should be run both cold and hot. In each case, OO7 also tests the coupling of the query facility with the application programming language by requiring a "do nothing" function to be called with field values from each qualifying object. All of the queries are read-only. Again, gaps in the numbering correspond to queries that we tested but eliminated from this paper due to space constraints or a lack of useful additional insights.

5.3.1 Query Q1: 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. The left graph in Figure 13 gives the results we measured on the small database. The slow Objectivity cold times are in part due to the fact that Objectivity reads in all of the schema information for the database when the first transaction in an application is begun.[6] The time for these reads are included in the cold query time.

The right graph in Figure 13 gives the results we measured on the small database for the corresponding "hot" times.

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. (An Objectivity container is roughly at the same granularity as a file.)

[6] Clearly, it should be a simple modification for Objectivity to instead read the schema information in when the database is opened.

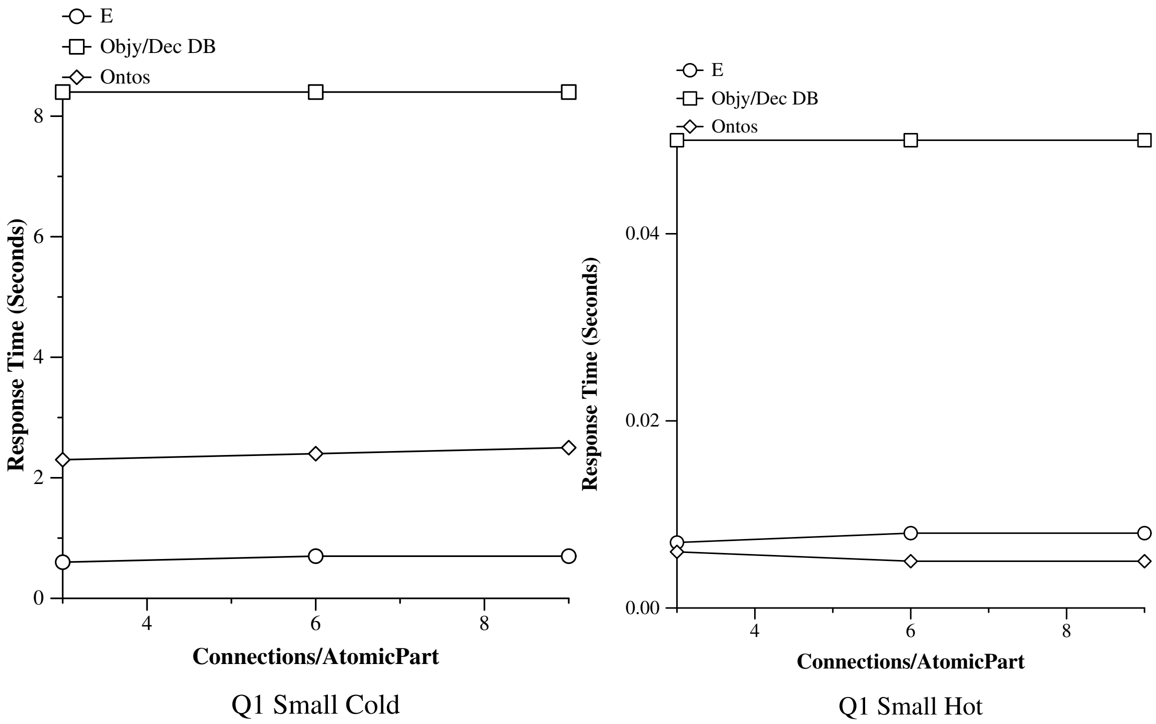

Figure 13: Q1, small database.

Exodus appears to have the most efficient index lookup implementation, by a significant margin, followed next by Ontos.

Finally, Figure 14 gives the results we measured on the medium database for the cold queries.

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.

5.3.2 Queries Q2, Q3, and Q7.

These queries are most interesting when considered together:

- *Query Q2: 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 Q3: 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 Q7: Scan all atomic parts.*

Note that queries Q2 and Q3 are candidates for a B+ tree lookup.

On the medium fanout 6 database we obtained the following "cold" numbers. (Similar results were obtained for fanouts 3 and 9, so we omit them here.)

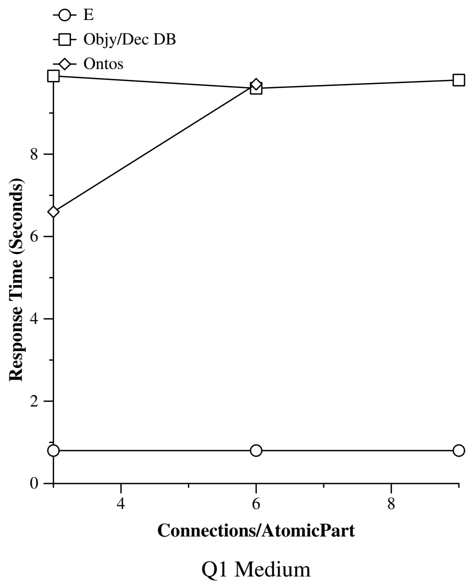

Figure 14: Q1, cold query, medium database.

Fanout 6	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, Q2 and Q3 were implemented as hand-coded B+ tree lookups. In Objectivity, it was not necessary to hand code these scans — the queries were implemented by using an Objectivity iterator with a selection predicate (a hand-assembled predicate, though, not a textual one). The Ontos times shown are again for hand-coded index (B+ tree, in this case) lookups. 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.[7]

5.3.3 Query Q4: path lookup

Generate 100 random document titles. For each title generated, find all base assemblies that use the composite part corresponding to the document. Also, count the total number of base assemblies that qualify.

Note that if the system supports path indices, this query can be run without faulting in the documents or their corresponding composite parts. If the system does not support path indices, then all objects along the selected paths must be brought in. Unfortunately, ODI was the only

[7]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 the future.

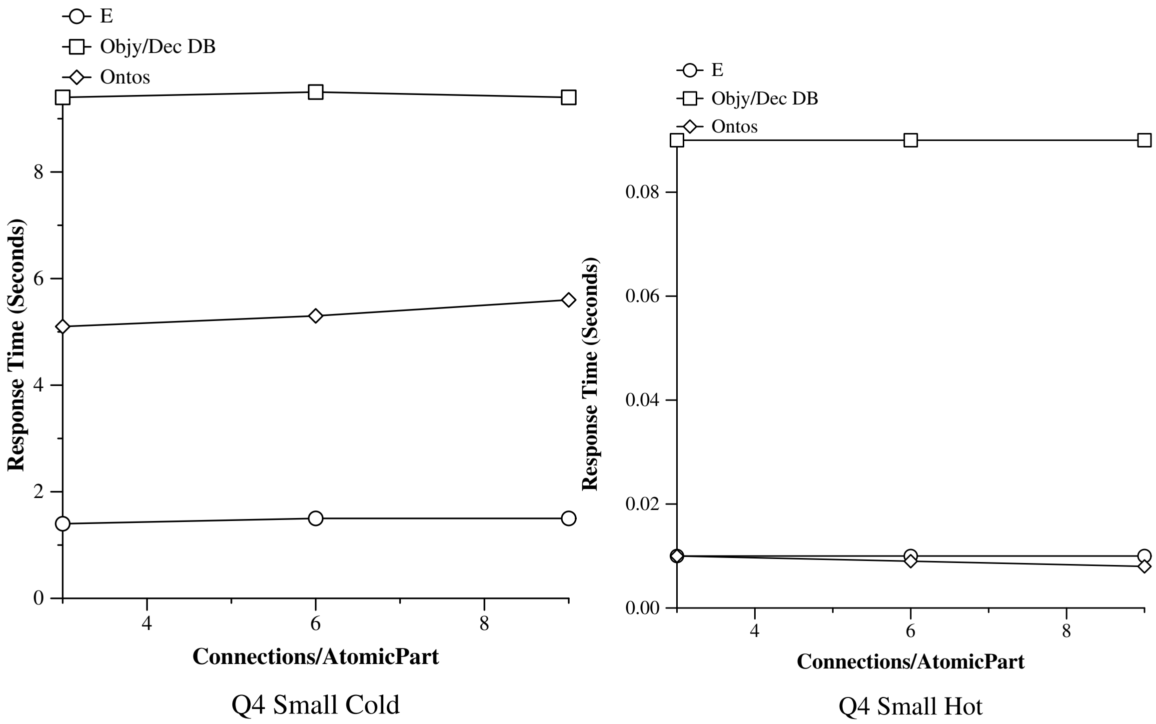

Figure 15: Q4, small database.

system we tested that supports path indices, so this is another operation that became less interesting after ODI withdrew from the benchmarking effort.

The left graph in Figure 15 shows the small database cold times, while the right graph in Figure 15 gives the small database hot times. Finally, Figure 16 gives the medium cold times.

5.3.4 Query Q5: single-level make

Find all base assemblies that use a composite part with a build date later than the build date of the base assembly. Also, report the number of qualifying base assemblies found.

This query mimics the processing that the Unix "make" command must do to determine which base assemblies are rendered out of date after modifying some of the composite parts.

The left graph in Figure 17 shows the small database cold times, while the right graph in Figure 17 gives the small database hot times. The most interesting aspect of this graph is the way the performance of E/Exodus deteriorates moving from fanout 3 to 6 to 9. This is due to paging within the client buffer pool. Q5 is essentially a pairwise, unclustered, pointer join between base assemblies and composite parts, so pages containing composite parts are likely to be reaccessed several times during the execution of the query. At a fanout of 3, the Exodus database is 11.5M, while the client buffer pool is 12M, so paging is not a problem. However, at fanout 6, the Exodus database is 13.4M, so we begin to see degradation due to paging; note that this effect is visible in both the cold and hot times. At a fanout of 9, the Exodus database is 15.9M, and the effect becomes

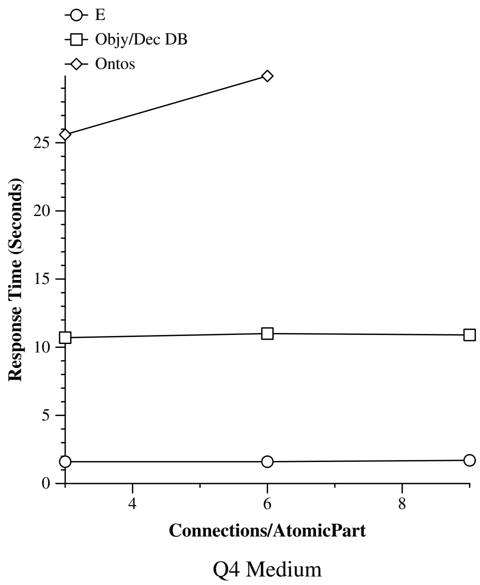

Figure 16: Q4, cold query, medium database.

even more pronounced. For the other two database systems, since the small databases are less than 12M, each page can be accessed once and then found in the client cache on subsequent references; as a result, no such paging occurs for those systems. These results highlight the importance of having reasonable pointer sizes, indicating an area where improvement is needed in Exodus.

Figure 18 gives the medium cold times. Here, none of the databases fits into the client buffer pool, so the performance of Exodus is more uniform when moving from fanout 3 to fanout 6 to fanout 9.

5.3.5 Query Q8: ad-hoc join

Find all pairs of documents and atomic parts where the document id in the atomic part matches the id of the document. Also, return a count of the number of such pairs encountered.

This is intended to test an OODBMS's value-based join processing ability. In the systems we tested, only Ontos's Object-SQL could express this query in a declarative form. Unfortunately, as mentioned earlier in this report, Object-SQL does not have a query optimizer so the performance was unacceptably slow on this test. For this reason, in all systems the result presented is the time for a hand-coded indexed nested loops algorithm.

The left graph in Figure 19 shows the small database cold times, while the right graph in Figure 19 gives the small database hot times. As in Q5, the small database performance results for E/Exodus show that the client buffer pool begins paging at fanout 6, as Q8 involves a pattern of page references that is qualitatively similar to that of Q5. Finally, Figure 20 gives the medium cold times.

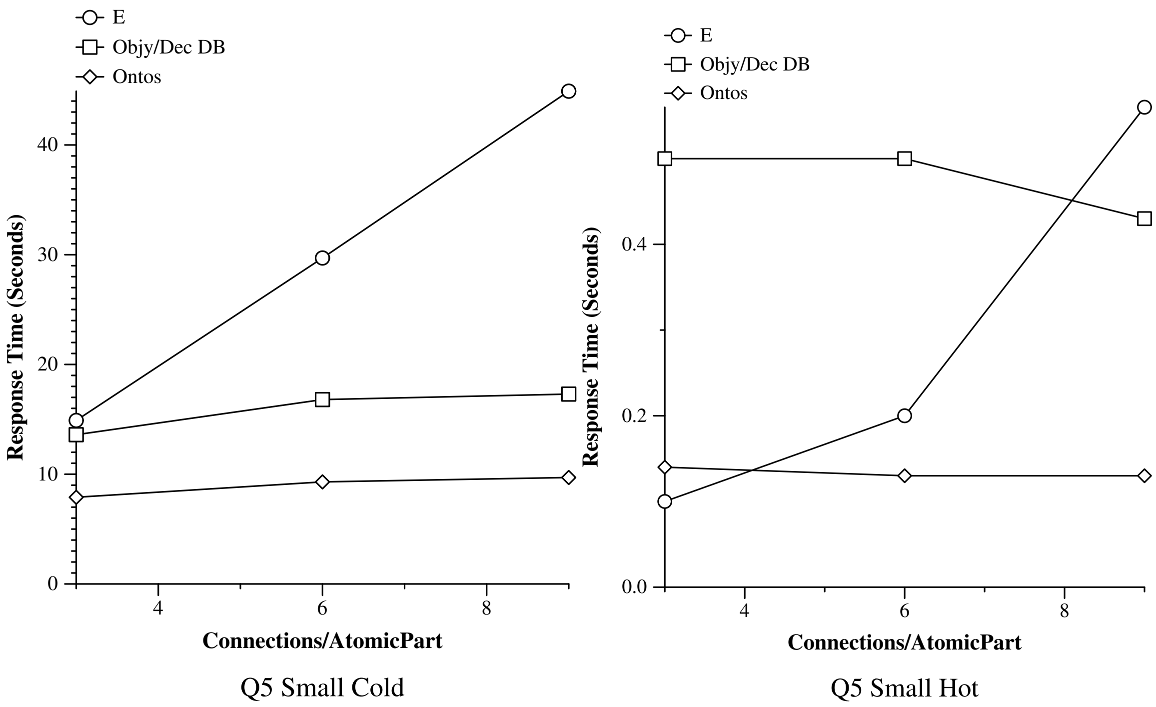

Figure 17: Q5, small database.

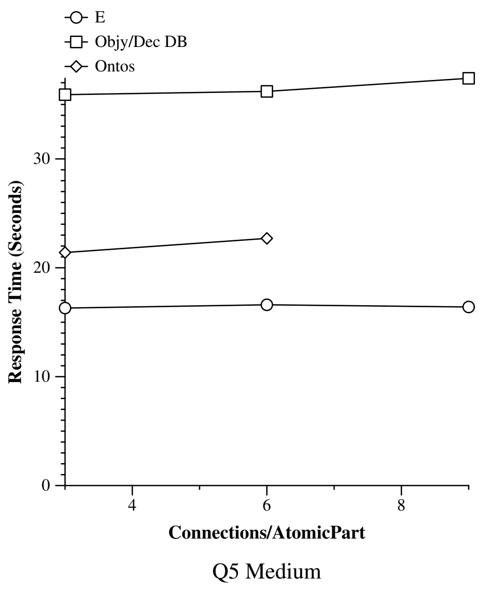

Figure 18: Q5, cold query, medium database.

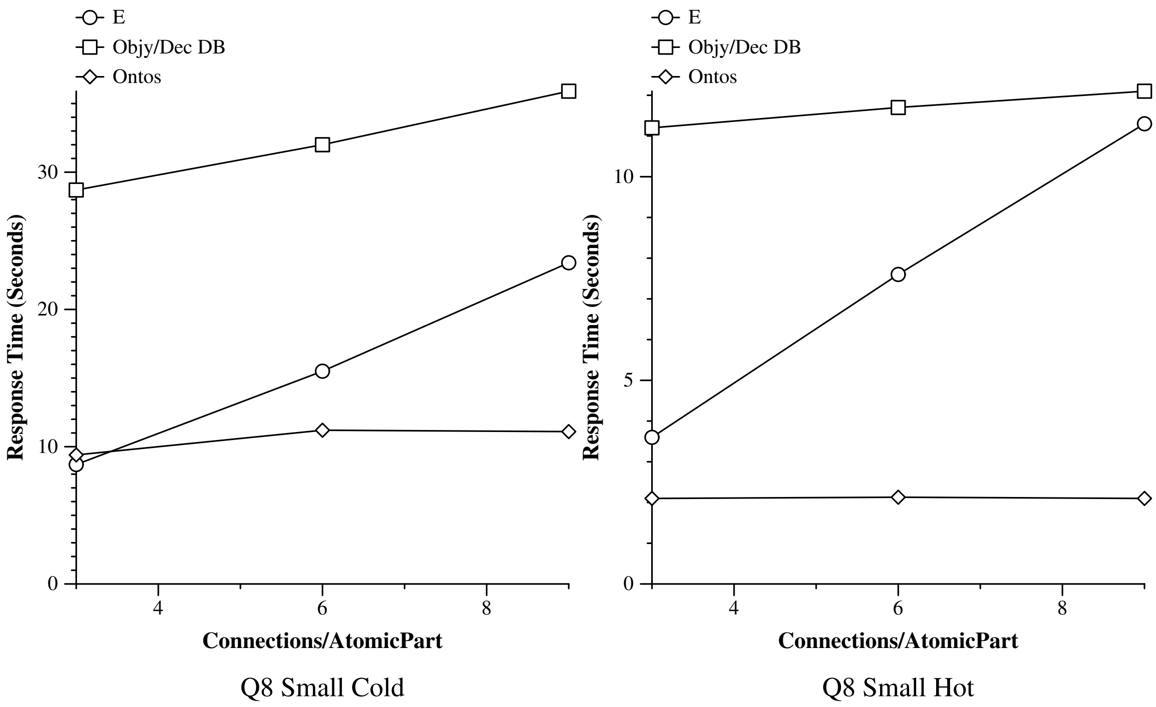

Figure 19: Q8, small database.

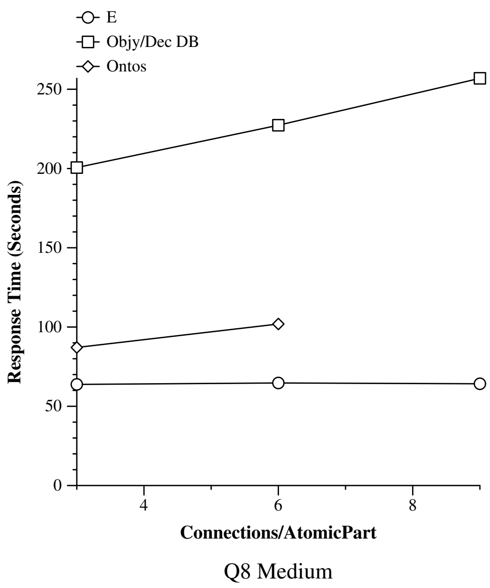

Figure 20: Q8, cold query, medium database.

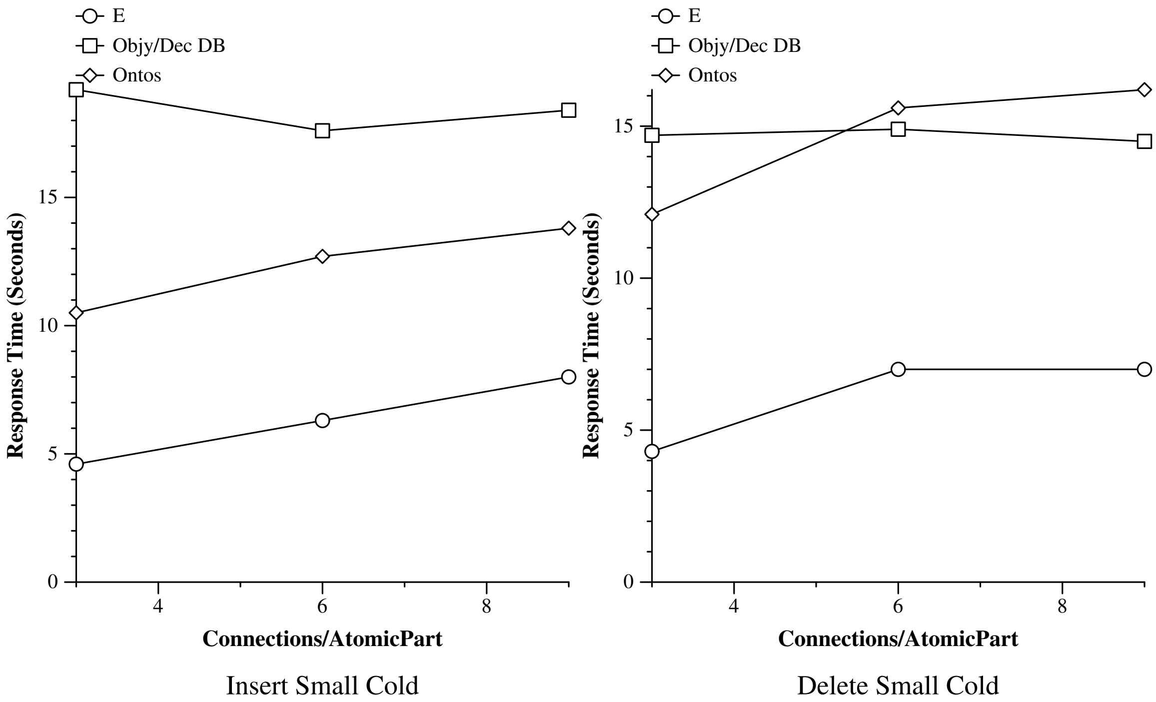

Figure 21: Insert and Delete, small database.

5.4 Structural Modification Operations

In this report we present the results from an insert operation (insert five new composite parts) and a delete operation (delete five composite parts). We also timed the "build" operation, but since the database build programs were very different among the different systems (e.g., in E/Exodus and Ontos we used multiple transactions, in Objectivity the build was a single large transaction) we felt that a comparison of the build times was not meaningful.

5.4.1 Structural Modification: Insert

Create five new composite parts, which includes creating a number of new atomic parts (100 in the small configuration, 1000 in the large, and five new document objects) and insert them into the database by installing references to these composite parts into 10 randomly chosen base assembly objects.

The left graph in Figure 21 shows the results of the insert operation in the small databases, while the left graph in Figure 22 gives the results for the medium databases.

5.4.2 Structural Modification 2: Delete

Delete the five newly created composite parts (and all of their associated atomic parts and document objects).

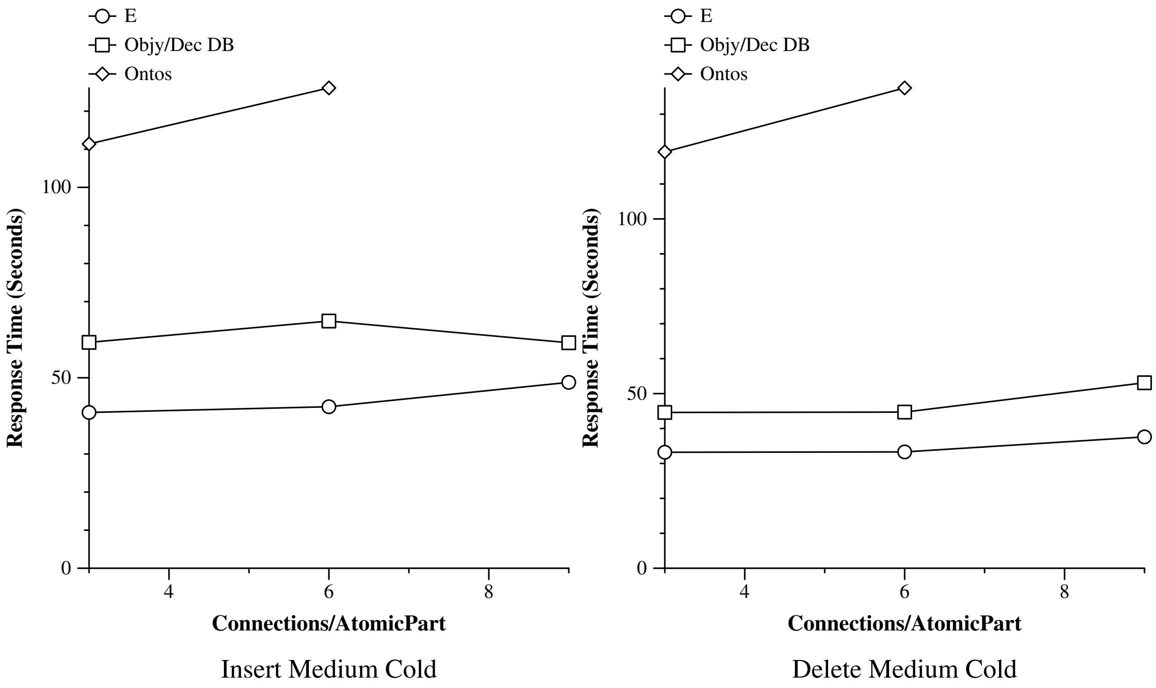

Figure 22: Insert and Delete, medium database.

The right graph in Figure 21 shows the results of the delete operation in the small databases, while the right graph in Figure 22 gives the results for the medium databases.

6 Conclusion

The OO7 Benchmark is designed to provide a compreshensive profile of the performance of an 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. The law firm of Foley, Hoag, and Eliot kept our FAX machine humming and our mailboxes full. Finally, we would like to give a special thanks to four staff members at the University of Wisconsin — Joseph Burger, 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 *Proceedings of the EDBT Conference*, Venice, Italy, March 1990.

[BDT83] Dina Bitton, David J. DeWitt, and Carolyn Turbyfill. Benchmarking database systems: A systematic approach. In *Proceedings of the Ninth International Conference on Very large data bases*, pages 8–19, 1983.

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

[DD88] J. Duhl and C. Damon. A performance comparison of object and relational databases using the sun benchmark. In *Proceedings of the ACM OOPSLA Conference*, San Diego, California, September 1988.

[DFMV90] David J. DeWitt, Philippe Futtersack, David Maier, and Fernando Velez. A study of three alternative workstation-server architectures for object-oreinted database systems. In *Proceedings of the VLDB Conference*, Brisbane, Australia, August 1990.

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

[FZT+92] Michael J. Franklin, Michael J. Zwilling, C. K. Tan, Michael J. Carey, and David J. DeWitt. Crash recovery in client-server EXODUS. In *Proceedings of the ACM-SIGMOD Conference on the Management of Data*, pages 165–174, June 1992.

[Gra81] Jim N. Gray et al. The recovery manager of the System R database manager. *ACM Computing Surveys*, 13(2):223 – 242, June 1981.

[RC89] Joel E. Richardson and Michael J. Carey. Persistence in the E language: Issues and implementation. *Software Practice and Experience*, 19, December 1989.

[RKC87] W. Rubenstein, M. Kubicar, and R. Cattell. Benchmarking simple database operations. In *Proceedings of the ACM SIGMOD Conference*, San Francisco, California, May 1987.

[SCD] D. Schuh, M. Carey, and D. DeWitt. Implementing persistent object bases: Principles and practice. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*.

[WC92] M. Winslett and S. Chu. Database management systems for ECAD applications: Architecture and performance. In *Proceedings of the NSF Conference on Design and Manufacturing Systems*, Atlanta, Georgia, January 1992.

[WD92] S. White and D. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proceedings of the VLDB Conference*, Vancouver, British Columbia, August 1992.

A OO7 Benchmark Database Schema

The following is a description of the OO7 schema in a notation that is a hybrid of the C++ based data definition languages (DDLs) of several of the systems that we tested. Inter-object references are denoted using C++ pointer notation, e.g., the `partOf` field of each atomic part object is a reference to the composite part object that the atomic part is a part of, and its type is indicated as `CompositePart*`. The expressions `Set(T*)` and `Bag(T*)` denote sets and bags (multisets) of references to objects of type `T`. For example, the `to` field of an atomic part object is a set containing references to the connection objects that hold data about (outgoing) connections between the atomic part and other atomic parts. In addition, inverse relationships are denoted in the schema using `<->`, pronounced "inverse of". The fact that an atomic part has an inverse relationship with the connection objects that it is connected to is captured in the DDL description of the atomic part class via: `Set(Connection*) to <-> Connection::from`. This says that the field `to` field of an atomic part is inversely related to the `from` field of the connections that it references. Finally, it should be noted that the OO7 schema includes instances of 1:1, 1:N, and M:N relationships; these are modeled as inverse relationships between a pair of reference fields, between a reference field and a set of references, and between two bags of references, respectively.

In addition to type information, the DDL description that follows also notes the top-level collections (expressed here as type extents) that are needed for the benchmark operations. These are indicated via the `with extent` annotation at the end of some of the class definitions. Also listed in the `with extent` clauses are those attributes (data members) of each class that should be indexed in order to provide acceptable performance on the various benchmark operations. It should be noted that it is not mandatory for the indicated type extents to be implemented as such;

however, some way must be provided to materialize the relevant collections (e.g., the set of all atomic parts) to support the benchmark operations. Similarly, the indices suggested here are not mandatory, but omitting one or more of them is likely to lead to inferior OO7 performance results. Finally, we also avoid dictating any one particular data clustering strategy. Rather, implementors are free to cluster the objects of the OO7 database however they like in order to achieve good overall performance – subject to the constraint that the same database instance (and thus the same clustering strategy) must be used for the entire series of OO7 operations.

```
//----------------------------------------------------------------------------
// DesignObj is the root of the class hierarchy for most OO7 objects
//----------------------------------------------------------------------------

class DesignObj {
    int                 id;
    char                type[10];
    int                 buildDate;
};

//----------------------------------------------------------------------------
// AtomicPart objects are the primitives for building up designs
//----------------------------------------------------------------------------

class AtomicPart: DesignObj {
    int                 x, y;
    int                 docId;
    Set(Connection*)    to <-> Connection::from;
    Set(Connection*)    from <-> Connection::to;
    CompositePart*      partOf <-> CompositePart::parts;
} with extent (id indexed, buildDate indexed);

//----------------------------------------------------------------------------
// Connection objects are used to wire AtomicParts together
//----------------------------------------------------------------------------

class Connection {
    char                type[10];
    int                 length;
    AtomicPart*         from <-> AtomicPart::to;
    AtomicPart*         to <-> AtomicPart::from;
};
```

```
//-----------------------------------------------------------------------------
// CompositeParts are parts constructed from AtomicParts
//-----------------------------------------------------------------------------

class CompositePart: DesignObj {
    Document*          documentation <-> Document::part;
    Bag(BaseAssembly*) usedInPriv <-> BaseAssembly::componentsPriv;
    Bag(BaseAssembly*) usedInShar <-> BaseAssembly::componentsShar;
    Set(AtomicPart*)   parts <-> AtomicPart::partOf;
    AtomicPart*        rootPart;
} with extent (id indexed);

//-----------------------------------------------------------------------------
// Document objects are used to describe CompositePart objects
//-----------------------------------------------------------------------------

class Document {
    char               title[40];
    int                id;
    String             text;
    CompositePart*     part <-> CompositePart::documentation;
} with extent (title indexed, id indexed);


//-----------------------------------------------------------------------------
// Manual objects are used to describe a whole Module.
//-----------------------------------------------------------------------------

class Manual {
    char               title[40];
    int                id;
    String             text;
    int                textLen;
    Module*            mod <-> Module::man;
};
```

```
//-------------------------------------------------------------------------
// Assembly objects are used to build up hierarchical designs
//-------------------------------------------------------------------------

class Assembly: DesignObj {
    ComplexAssembly*    superAssembly <-> ComplexAssembly::subAssemblies;
    Module*             module <-> Module::assemblies;
};

class ComplexAssembly: Assembly {
    Set(Assembly*)      subAssemblies <-> Assembly::superAssembly;
}

class BaseAssembly: Assembly {
    Bag(CompositePart*) componentsPriv <-> CompositePart::usedInPriv;
    Bag(CompositePart*) componentsShar <-> CompositePart::usedInShar;
} with extent (id indexed);

//-------------------------------------------------------------------------
// Modules are the designs resulting from Assembly composition
//-------------------------------------------------------------------------

class Module: DesignObj {
    Manual*             man <-> Manual::mod;
    Set(Assembly*)      assemblies <-> Assembly::module;
    ComplexAssembly*    designRoot;
} with extent (id indexed);
```

Finally, Figure 23 gives an extended Entity-Relationship diagram for the schema of the OO7 database for those readers who prefer to think in terms of E-R modeling concepts.

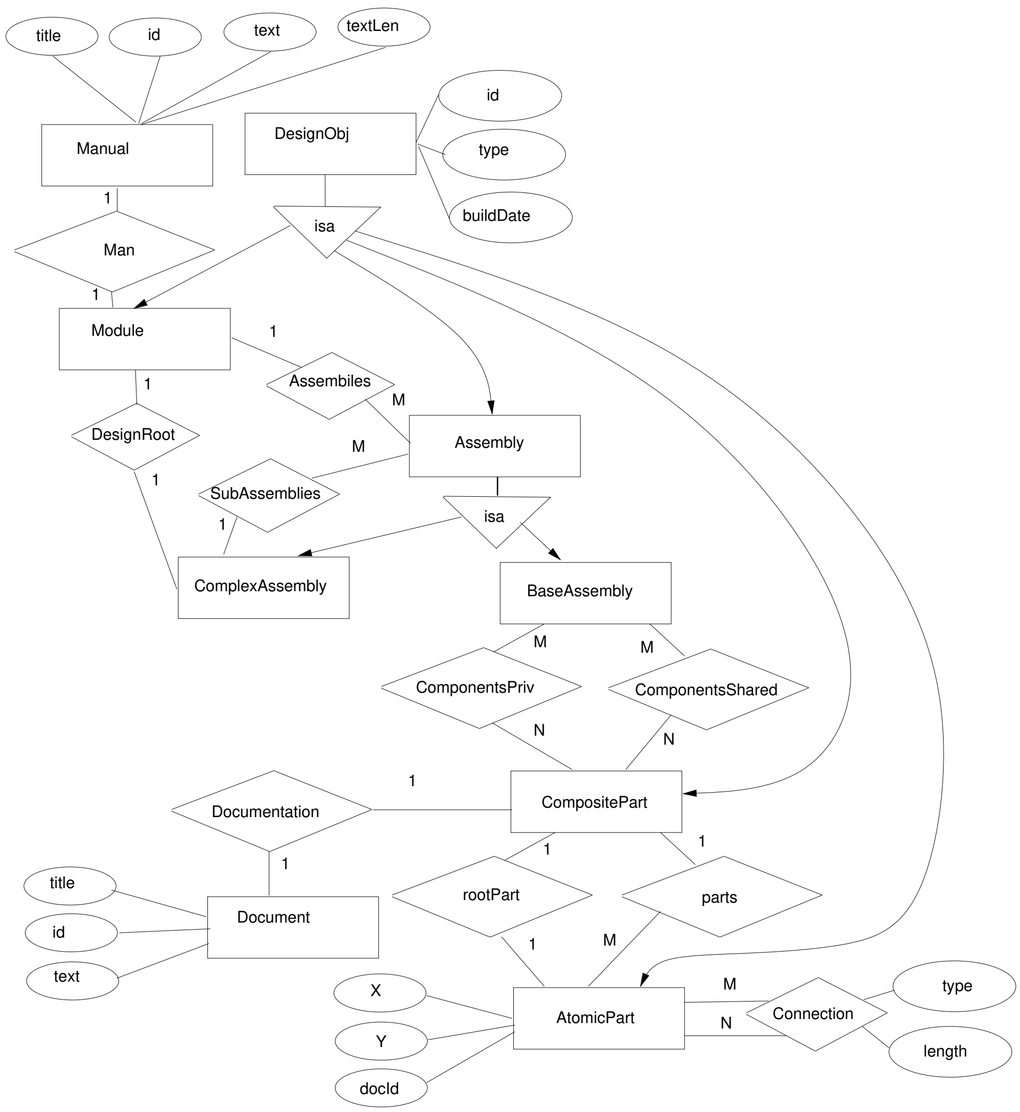

Figure 23: Entity-relationship diagram for the OO7 benchmark.