Object Operations Benchmark

R. G. G. CATTELL and J. SKEEN
Sun Microsystems

Performance is a major issue in the acceptance of object-oriented and relational database systems aimed at engineering applications such as computer-aided software engineering (CASE) and computer-aided design (CAD). Because traditional database systems benchmarks are inappropriate to measure performance for operations on engineering objects, we designed a new benchmark Object Operations version 1 (OO1) to focus on important characteristics of these applications. OO1 is descended from an earlier benchmark for simple database operations and is based on several years experience with that benchmark. In this paper we describe the OO1 benchmark and results we obtained running it on a variety of database systems. We provide a careful specification of the benchmark, show how it can be implemented on database systems, and present evidence that more than an order of magnitude difference in performance can result from a DBMS implementation quite different from current products: minimizing overhead per database call, offloading database server functionality to workstations, taking advantage of large main memories, and using link-based methods.

Categories and Subject Descriptors: B.2.2 [**Arithmetic and Logic Structures**]: Performance Analysis and Design Aids; D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.8 [**Software Engineering**]: Metrics—*performance measures*; H.2.1 [**Database Management**]: Logical Design—*data models*; H.2.3 [**Database Management**]: Languages—*data description languages (DDL), data manipulation languages (DML), database (persistent) programming languages;* H.2.8 [**Database Management**]: Database Applications; K.6.2 [**Management of Computing and Information Systems**]: Installation Management—*benchmarks, performance and usage measurement*

General Terms: Algorithms, Design, Experimentation, Languages, Measurement, Performance

Additional Key Words and Phrases: Benchmark, CAD, CASE, client-server architecture, engineering database benchmark, hypermodel, object operations benchmark, object-oriented DBMS's, relation of DBMS's, workstations

1. INTRODUCTION

We believe that the additional functionality that the new generation of object-oriented and extended relational DBMSs provide for engineering applications such as CASE and CAD is important, but these new DBMSs will not be widely accepted unless a factor of ten to one hundred times the performance of traditional DBMSs is achieved for common operations that

Authors' address: Sun Microsystems, 2550 Garcia Ave., Mountain View, CA 94043.

engineering applications perform. Engineering applications are of particular interest to us, so we are attempting to focus some attention on performance in this area.

Measuring engineering DBMS performance in a generic way is very difficult, since every application has somewhat different requirements. However, our goal is simply to measure common operations that differ from traditional business applications, operations that cannot be expressed in the abstractions provided by SQL and other high-level DMLs. Engineering applications typically utilize a programming language interspersed with operations on individual persistent data objects. Although database access at this fine-grain level is in some ways a step backward from high-level operations on entire relations in modern DMLs, it is a step forward from current ad hoc implementations and is the best solution available at present.

As an example, consider a CAD application: components and their interconnections on a circuit board might be stored in a database and an optimization algorithm might follow connections between components and rearrange them to reduce wire lengths. In a CASE application, program modules and their interdependencies might be stored in a database, and a system-build algorithm might traverse the dependency graph, examining version numbers to construct a compilation plan. In both cases, hundreds or thousands of objects are accessed per second, perhaps executing the equivalent of a relational join for each object. Simply adding transitive closure to the query language is inadequate, as the steps performed at each junction can be arbitrarily complex. The DBMS must either provide the full functionality of a programming language in the database query language, or it must be possible to efficiently mix the programming and query languages.

2. ENGINEERING DATABASE PERFORMANCE

We do not believe that traditional measures of DBMS system performance, the Wisconsin benchmarks [5] and TPC-A (Transaction Processing Performance Council [10]), adequately measure engineering application performance. The TPC-A benchmark is designed to measure transaction throughput with large numbers of users. While some of the Wisconsin measures are relevant to engineering performance, they generally are measures at too coarse a grain and focus on the intelligence of the query optimizer on complex queries that are rare in our applications. Current query languages are not capable of expressing the operations that our applications require; when this problem is remedied, a benchmark for the more complex queries will be appropriate.

The most accurate measure of performance for engineering applications would be to run an actual application, representing the data in the manner best suited to each potential DBMS. However, we feel it is difficult or impossible to design an application whose performance would be representative of many different engineering applications, and we want a generic measure. Perhaps in the future someone will be successful at this more difficult task [6, 11].

Our generic benchmark measures are the operations we expect to be frequent in engineering applications, based on interviews with CASE and CAD engineers as well as feedback on our previous paper [9]. The benchmark we describe, OO1,[1] substantially improves upon our earlier work, simplifying and focusing the measurement on important object operations. The HyperModel benchmark [1] was also designed to improve on our earlier benchmark; however, that work has gone in the direction of a more comprehensive set of measures rather than simpler ones. We contrast these two benchmarks in Section 5.

The OO1 benchmark measures include *inserting* objects, *looking up* objects, and *traversing connections* between objects. The benchmark is designed to be applied to object-oriented, relational, network or hierarchical database systems, even B-tree packages or custom application-specific database systems. Engineering applications may also require more complex or ad hoc queries, so performance on the OO1 benchmark alone does not make a system acceptable. Also, the OO1 mix of operations is only representative of some engineering applications. However, we believe current systems are orders of magnitude away from acceptable performance levels for engineering applications, so we simply aim to focus attention where high performance is required, just as the TPC-A benchmark focuses on commercial on-line transaction processing.

There is a tremendous gap between the performance provided by in-memory programming language data structures and that provided by disk-based structures in a conventional database management system. Existing database systems typically respond to queries in tenths of a second. Simple lookups using in-memory structures can be performed in microseconds. This factor of 100,000 difference in response time is the result of many factors, not simply disk access. We believe there is a place for a database system that spans the gap between these two systems, performing close to 1,000 operations per second on typical engineering databases.

How can much higher performance be achieved? What do we give up to get this performance? We believe a number of DBMS features used in commercial applications are not necessary in the engineering applications that require high performance. For example, we believe that the level of concurrency is quite different—an engineer may "check out" part of a design and work on it for days, with no requirements for concurrent updates by other engineers, and the data can largely be cached in main memory on the engineer's workstation.

Large improvements in engineering database performance are probably not going to be achieved through minor improvements in data model, physical representation, or query languages. In our experience, the substantial

[1] Note that our benchmark was originally performed in 1988; this paper, in slightly revised forms, has been informally available since then. The current version, containing further updates and more recent results, supersedes the previous ones. More recent information on DBMSs for engineering applications can be found in Cattell's book, *Object Data Management*, Addison-Wesley, 1991.

improvements result from major changes in DBMS architecture: providing efficient remote access to data, caching a large working set of data in main memory, avoiding an "impedance mismatch" between programming and query languages, and providing new access methods with fixed rather than logarithmic access time. Differences in data models (e.g., relational and object oriented) can be dwarfed by architectural differences, as long as the data model does not dictate implementation.

3. BENCHMARK DATABASE

We want a benchmark database independent of the data model provided by the DBMS. The following should therefore be regarded as an abstract definition of the information to be stored, possibly as a single object type with list-valued fields, possibly as two or more record types in a relational system.

We define the database as two logical records:

```
Part: RECORD[id: INT, type: STRING [10], x, y: INT, build. DATE]
Connection. RECORD[from· Part-id, to  Part-id, type· STRING [10], length. INT]
```

We define a database of parts with unique id's 1 through 20,000, and 60,000 connections, with exactly three connections from each part to other (randomly selected) parts. The x, y, and length field values are randomly distributed in the range [0..99999], the type fields have values randomly selected from the strings {"part-type0" . . . "part-type9"}, and the build date is randomly distributed in a 10-year range.

We assume that the database system allows data fields with the scalar types specified above, where INT is a 32-bit integer, DATE is some representation of a date and time, and STRING[N] is a variable-length string of maximum size N. The from and to fields are references to specific parts; they might be implemented by part id foreign keys in a relational DBMS, or by unique identifiers in an object-oriented DBMS. Also, it is permissible to combine part and connection information in a single object if the DBMS permits it. However, the information associated with connections (type and length) must be preserved. Also, it must be possible to traverse connections in either direction.

The random connections between parts are selected to produce some locality of reference. Specifically, 90 percent of the connections are randomly selected among the 1 percent of parts that are "closest," and the remaining connections are made to any randomly selected part. Closeness is defined using the parts with the numerically closest part ids. We use the following algorithm to select part connections, where rand[1..k] is a random integer in the range 1 to k, and N is the number of parts in the database.

```
for part in [1  . N]
  for connection in [1 . . 3]
    if rand[1 .  10] > 1 then
      /* 90% of time create connection to the closest 1% of parts */
  let cpart = part + rand[1 . . N / 100] − 1 − N / 200;
  /* "double up" at the ends so stay in part id range */
  if cpart < N / 200 then let cpart = cpart + N / 200;
  if cpart > N − N / 200 then let cpart = cpart − N / 200;
  connect(part, cpart)
else
  /* 10% of time create connection to any part 1 . . N */
  let cpart = rand[1 . . N]
  connect(part, cpart)
```

Using part id's for locality is quite arbitrary; in Section 6, we justify that this is as good as any other criterion. The choice of 90 percent colocality of reference is also somewhat arbitrary; however, we found that unless locality is as high as 99 percent or as low as 1 percent it does not substantially affect the result (we return to this issue in Section 9).

Our database comprises approximately 2 megabytes of attribute data, plus space overhead for access methods and record storage, which typically doubles this amount. As such, the database is approximately 4 megabytes altogether and is a good representative of an engineering database working set that fits entirely in main memory—for example, the data associated with a CAD drawing shown on the screen, the data in one engineer's portion of a larger engineering design or software project, or a working set from a larger knowledge base used by an expert system.

A DBMS must also scale up to larger databases and working sets. We refer to the database we just described as the *small* database, and include a second database, the *large* database, that is identical except that the record counts are scaled up by a factor of ten. This database then requires approximately 40 megabytes of storage and overhead. Some database systems will not show a substantial performance difference on the larger database, while others, particularly those that use a main memory cache, will run much faster on the *small* database. Other systems, e.g., a persistent programming language, may not even permit the larger database size.

As an experiment, we also scaled up to a *huge* 400 megabyte database to test larger real-world databases. However, we do not require this database in our benchmark because the results seem to differ less dramatically than the first two.

We will see that the most important distinction between the databases is the different *working set* size, not the need for access methods that scale up efficiently with larger data sets on disk. Using the benchmark machine configuration we define in Section 6, the *small* database working set fits in main memory, while the *large* and *huge* databases do not.

We assume that the database itself resides on a server on a network, while the engineering application runs on a workstation. We feel this *remote* database configuration is the most realistic representation of engineering and office databases, and is consistent with the move towards database servers on a network. Performance for remote databases may be significantly impacted by the caching and concurrency control architecture of the DBMS. We also report performance for a *local* database, with the benchmark application running on the same machine where the database is stored, but we do not require the *local* database configuration for purposes of reproducing the benchmark.

4. BENCHMARK MEASURES

We now describe the benchmark. The benchmark measures *response time* and is run by a single user.[2] This is consistent with our model of an engineer "checking out" a segment of data for exclusive use; indeed, the performance we require may not even be achievable with current technology if there is highly concurrent access by multiple users. However, it is important for the DBMS to *allow* multiuser access to the database, and we do require that the data used be locked or versioned in order to support multiple users.

The following three operations are our benchmark measures. We run each measure 10 times, and measure response time for each run to check consistency and caching behavior.

(1) *Lookup.* Generate 1,000 random part id's and fetch the corresponding parts from the database. For each part, call a null procedure written in any host programming language, passing the x, y position and type of the part.

(2) *Traversal.* Find all parts connected to randomly selected part, or to a part connected to it, and so on, up to seven hops (total of 3,280 parts, with possible duplicates). For each part, call a null programming language procedure with the value of the x and y fields and the part type. Also measure time for *reverse* traversal, swapping "from" and "to" directions, to compare the results obtained.

(3) *Insert.* Enter 100 parts and three connections from each to other randomly selected parts. Time must be included to update indices or other access structures used in the execution of (1) and (2). Call a null programming language procedure to obtain the x, y position for each insert. Commit the changes to the disk.

The benchmark measures were selected to be representative of the data operations in many engineering applications. For example, in a CASE application, an operation very similar to the traversal must be performed to determine a compilation plan for a system configuration, an operation like the lookup is needed to find programs associated with a particular system or range of dates, and an operation like the insert is needed to enter new information after a new module is compiled. In an ECAD application, the traversal is needed to optimize a circuit layout, the lookup to find components with particular types, and the insertion to add new components to a circuit board.

[2] Response time is the real (wall clock) time elapsed from the point where a program calls the database system with a particular query, until the results of the query, if any, have been placed into the program's variables.

We feel there is some value to a single "overall" performance time, as with the TPC-A benchmark. We designed the benchmark such that the three measures were executed approximately in proportion to their frequency of occurrence in representative applications (an order of magnitude more reads than writes, and several times more traversal operations than lookups, as a very rough estimate from interviews with CASE and CAD engineers), so that a single overall number can be computed by adding the three results. However, we warn against the abuse of the "overall" number, because the frequency of occurrence of these operations will differ widely between applications: the individual measures are important.

Note that we require that the host application program be called on each simple operation above or that each database operation be called from the host programming language to simulate an engineering application where arbitrary programming operations must be mixed with database operations. For the three operations above, for example, the corresponding procedures might (1) display the part on the screen, (2) compute total wire lengths along different paths, and (3) compute the x, y position for placing the part in a diagram.

Our requirement to call the host application programming language may be controversial: it could be argued that all of the data should be fetched by a single query before returning to the application program or that embedded procedures or procedural constructs in a query language should be used instead of calling the application program itself with the data. However, this requirement of intermixing database access and programming is one of the factors that makes our benchmark interesting. The restriction is necessary to allow an application programmer to take advantage of the computation power available on workstations and to encode operations more complex than our simple benchmark—operations that cannot be encoded in the query language and embedded procedures. For example, in a real application the "traversal" would probably not traverse every node to a fixed depth; the application might perform arbitrary computation based on previous parts visited, global data structures, and perhaps even user input to decide which nodes to traverse next.

Because of our requirement for intermixing database and programming operations at a fine grain, we refer to the OO1 benchmark as *interactive*. We leave to future research the exploration of a *batch* version of OO1, in which all repetitions of an operation, for example the 1,000 lookups or 3,280 traversals, may be executed in a single database call. We further discuss the batch versus interactive issues in Section 7.

We require that the benchmark measures be performed ten times, the first time with no data cached. These repetitions provide a consistency check on the variation on the results and also show the performance (on the second, third, and remaining iterations) as successively more data is in memory. Of course, there will be some cache hits even on the first iteration, depending on database system parameters such as page size, clustering, and caching strategy. As a result, a DBMS can achieve a cache hit ratio for the first iteration as low as 10 percent or as large as 90 percent.

We refer to the results of the first iteration as the *cold* start results and the asymptotic best times (the tenth iteration, when the cache is fully initialized) as the *warm* start results. In the systems we measured, we found the *warm* asymptote was achieved in only two or three iterations, with only small (under 10 percent) random variations after that.

A *different* set of random parts is fetched with each iteration of the measures. Thus, most or all of the database must be cached in order to obtain good performance. The *small* database constitutes a working set that can fit in memory, the *large* database constitutes a working set that does not.

The benchmark implementor may colocate parts in the database such that the forward traversal in measure (2) may go faster than the reverse traversal. We leave it up to the user to decide whether the slower reverse traversal would be acceptable for their application. It is not acceptable to implement *no* access method for reverse traversal (e.g., do linear search to find parts connected to a part); this would take too long for almost any application. For example, in a CASE application it must be possible to find dependent modules as well as modules dependent upon the current module.

Note that the results for the reverse traversal may vary widely, as each part has three parts it is connected "to," but a random number of parts it is connected "from" (three on average). However, the average should be similar to the forward traversal. We found it convenient to count the number of nodes visited on the reverse traversal and normalize the result to obtain comparable figures (i.e., multiply the time by 3280/N, where N is the number of nodes actually visited).

It is not acceptable to take advantage of the fact that the benchmark part id's are numbered sequentially, computing disk address directly from part id—some sort of hash index, B-tree index, and/or record (object) ID based link structure is necessary for a realistic implementation.

Engineering applications typically require some form of concurrency control, but there is currently much debate as to whether they need the atomic transactions of commercial DBMSs or whether some form of versions or locking plus logging of updates is better. We side-step this debate by specifying that the benchmark numbers should be measured and reported with whatever concurrency control mechanisms the DBMS provides. If the DBMS provides more than one choice, the performance with each alternative is important to report. We leave it to the users to decide whether the concurrency control and recovery mechanisms provided, and their performance cost, is adequate for their application.

Although we do not specify which type of concurrency control must be provided, we do insist that the DBMS minimally guarantee physical and logical integrity of the data with multiple readers and writers on a network with the concurrency control mechanism (locking, versions, transactions) provided. Furthermore, the insert measure requires that updates be committed to disk after every 100 parts.

We require that the benchmark measures be performed on both the *small* and *large* database, as described earlier. The space as well as time requirements must be reported for the databases.

5. BENCHMARK JUSTIFICATION

We made many choices in selecting a simple but meaningful benchmark for object operations. In this section, we justify many of these choices, and contrast the OO1 benchmark to our earlier benchmark and the HyperModel benchmark.

Earlier Benchmark. We believe the OO1 benchmark is both simpler and more realistic than that defined in [9]. The database itself is similar, although we incorporated locality of reference and use a database of parts and connections instead of authors and documents (the database contents are a minor point, but it has been a source of confusion about our intended applications).

We dropped two measures we included in our original paper. The *range lookup* measure found the parts in a ten-day range of "build" dates. In our experience the result of this measure was very similar to the lookup measure; in fact, they differed only when more than a few parts were found (in which case the time was dominated by the copying time rather than the index lookup). We also dropped the *sequential scan*, which enumerated all parts in the database to fetch the x, y position for each.

We also dropped the *reference lookup* and *group lookup* measure in the original benchmark. These measures involved following references (e.g., from a connection record to a part) or back-references (e.g., from a part to the connections that reference it). We had hoped to estimate the time to do complex operations, such as the traversal we now propose, by composing individual *reference* and *group lookup* times. In practice it was hard to specify these measurements in such a way that different implementors got consistent numbers for their DBMS without including the set-up time (e.g., to fetch the record from which the reference emanated). So we replaced them with the single *traversal* measure.

We considered including *update* (in addition to *insert*) of part records. We found the results for update and insert to be closely correlated, however, so we included only the *insert*.

Thus, we chose the *lookup*, *traversal*, and *insert* operations as more important than *range lookup*, *sequential scan*, and *update* measures for purposes of a simple, focused benchmark. However, these additional results may be important to obtain for some applications, so we recommend they be included in a complete application study (e.g., [1]). In particular, it is essential that a DBMS include a B-Tree or other access method for *range lookup* because these occur in many applications.

HyperModel Benchmark. The HyperModel Benchmark [1] added to our original benchmark and database rather than simplifying it, thereby resulting in 17 benchmark measures, four types of database entities and four types of relationships. Although the HyperModel benchmark is more complex and therefore harder to implement, we believe it is a more thorough measure of engineering database performance than OO1.

The difference between OO1 and HyperModel might be likened to the difference between TPC-A [10] and the Wisconsin [5] benchmarks—our intent is to focus on overall performance for a few important operations, while the HyperModel benchmark provides many different measures. As such, we believe there is a place for both the HyperModel benchmark and OO1 in comparing systems. It is easy to produce OO1 results on any DBMS for which the HyperModel measures have been implemented because our database and measures are more or less a subset of theirs (mapping "nodes" to parts, and the "MtoN" relationship to connections).

We do not enumerate all of the differences in the HyperModel benchmark, but we note the following:

(1) The HyperModel database includes multiple relationships between objects, including a one-to-many subpart relationship, and a subtype relationship. It also includes a "blob" field that stores a multi-kilobyte value.
(2) The HyperModel benchmark includes the same six measures from our original paper, plus some new measures: a traversal measure like ours (they retain the group and reference measures rather than replacing them), and read/write operations on the "blob" field.

These extensions to the original benchmark are all useful in providing additional data about a DBMS and we think the choice of additions is good. However, we feel that all of these measures are of secondary importance to those we have included, so we have retained only our simpler measures. The "blob" operations are not DBMS bottlenecks in the applications we are familiar with—blob operations are typically limited by disk speed. Thus we include only small STRING and INT fields in our database. The additional relationships do seem to represent a more realistic database, but it is plausible that they all measure the same kinds of operations and access methods in the database; perhaps future experience with the HyperModel benchmark will show whether their traversal times are directly correlated.

In order to make the HyperModel benchmark more useful, we would like to see a few changes in its definition. Some work may already be underway on these changes at the time of this writing.

First, we feel it is important to precisely specify and control the details of implementation so that the benchmark results can be accurately compared. These details include those we cover in the next section: the amount of main memory dedicated to the DBMS and cache, the processor and disk performance, and control of initialization timing and cache.

Second, since the HyperModel benchmark is intended to be more complete in its coverage of engineering databases, it should include some representation of versions, a concept important in many CAD, CASE, and other engineering applications. Versions and the concurrency model should be tested in the context of a multiuser benchmark.

Third, the HyperModel benchmark introduces multiple connection types but defines no benchmark measure or guidelines for how the data should be clustered according to the various connections. Without this, the more complex data structures are not very useful.

Finally, and most importantly, we feel it essential that the HyperModel benchmark be performed using a client/server architecture. A *remote* database is more realistic of engineering and office applications.

Summary. We have responded to a number of suggestions made on our earlier benchmark. The HyperModel work in itself provides a list of suggestions, and we have explored other ideas from the referees and benchmark users. The list includes "blob" fields, inheritance, composite objects with cascaded delete and other operations, versions, multiuser measures, schema evolution, and the "range" queries we removed after our first paper.

We believe that all of these features belong in a complete benchmark, as we discussed in the HyperModel critique above, but that they reduce the utility of a simple benchmark focused on basic performance: they make it harder to implement and reproduce. We have therefore resisted the temptation to expand the benchmark except where we felt it essential.

6. NOTES ON BENCHMARKS

It is essential that a good benchmark be precisely reproducible. In this section, therefore, we clarify more detailed issues on its implementation.

In order to reproduce comparable results, it is necessary to run the benchmarks and DBMS on a similar configuration. Table I shows a number of pertinent particulars of the DBMS architecture and benchmark implementation that should be reported.

For our benchmarks, we used the following configuration on the server machine where the database resides:

Sun 3/280 (Motorola 68020, 4 MIPs) with 16 megabytes memory
SMD-4 disk controller (2.4 MB/sec transfer rate)
2 Hitachi disks, 892 MB each, 15-ms avg. seek time; 1 disk used for database, 1 for system
Sun UNIX 0/S 4.0.3

The client workstation, where we ran the *remote* benchmark programs, was a diskfull Sun 3/260 with 8 MB memory, SMD-4 disk controller, and two 892 MB disks (15-ms avg. seek time). This system ran Sun UNIX O/S 4.0.3. Both the client and server machines were stand alones reserved exclusively for benchmarking during the time of our runs. Swap space on both machines was generous, and was not an issue in any event, as we only ran the benchmark program process.

In all of our benchmark measures, we assume that the DBMS is initialized and that any schema or system information is already cached. The initialization time is included in the database open. The database open call is permitted to swap in as much of the DBMS code as desired on the server or client machines, so that code swap time will not be counted in the actual benchmark measures.

Table I. Reportable Benchmark Information

Area	Types of reportable information
Hardware	CPU type, amount of memory, controller, disk type/size
Software	O/S version, size of cache
DBMS	Transaction properties (whether atomic, level of supported concurrency, level of read consistency, lock modes used, etc.); recovery and logging properties; size of database cache; process architecture (communication protocol, number of processes involved, etc); security features (or lack thereof); network protocols; access methods used
Benchmark	Any discrepancies from the implementation described here; real "wall-clock" running time[3], CPU time and disk utilization time are also useful; size of database

We want to include any disk I/O overhead in the response time measurements for our benchmarks. On the other hand, many database systems *can* validly obtain better performance for engineering applications by caching data between the multiple lookups performed within one transaction or session. We allowed specifically for repetitions in the measures, to allow caching, and we also report the asymptotic *warm* results. However, no data may be cached at the time the benchmark is started for the *cold* results. The database must be closed and reopened before each benchmark measure, to empty the cache. Since the operating system may cache file pages, it is necessary to run a program to clear all database file pages out of memory on both the client and server machines. This is a subtle point that can make a major difference in the results: we suggest running a program or system command that sequentially reads all the pages of a large file, at least 20 megabytes, to fill the system cache with other data.

Thus, the general order of executing a specific benchmark measure for ten loops is:

```
⇒ Exit the previous benchmark measure, if any
⇒ Clear the operating system cache
⇒ Begin execution of the new benchmark measure
  Open the database
  Start timer
    Report cold time for the first benchmark (when cache is being filled)
      Include time for transaction checkpoint for "insert" measure
    Continue execution for remaining 9 loops
      If clear asymptote emerges, report it as WARM time; otherwise report
      average of the 9 runs
    Stop timer
    Close the database
⇒ Exit the benchmark
⇒ For the insert measure, restore the database to its original state
```

We allow the client workstation to cache as much of the database as it can, given 8MB of physical workstation memory. For most DBMS and O/S working sets, this leaves about 5MB for a cache (the cache may be in the O/S

[3] The real-time clock on Sun machines "ticks" at a granularity of 1/60 of a second, which is accurate enough for the measurements we require

or DBMS, or both). We conceive this amount of memory as typical of what is available for application use on contemporary workstations; it will no doubt increase as technology advances and manufacturing techniques improve. However, the amount of memory used by applications and other software will also increase, so approximately 5MB of cache may be realistic for some time to come.

For our *small* database, the entire database typically fits in main storage. For our *large* and *huge* database, there is little advantage to a cache larger than the small space required to hold root pages for indices, system information, data schema, and similar information. However, for the *large* database, a significant portion of the Part table may fit into memory for the lookup measure.

On the server, the memory used by O/S, DBMS, and cache should be limited to 16MB. If it is not possible to limit total memory, we recommend that server cache size be limited to 12MB instead. This was not necessary in our implementation, since we removed memory boards to produce the proper configuration.

There are several issues in the generation of the benchmark database:

Generating IDs. We populate the *small* database with 20,000 parts and 60,000 associated connections; these numbers are 200,000 and 600,000, respectively, for the *large* database. When the parts are loaded, successive records are given successive ID numbers. This allows the benchmark to select a random part, which is guaranteed to exist, by calculating a random integer. We choose indices or other physical structures to achieve the best performance. The physical structure may not be changed between benchmarks (i.e., the same index or link overhead must be included in the database size, update time, and read time).

Connections. There should be exactly three logical "connections" whose "from" field is set to each part, and the "to" fields for these connections should be randomly selected using the algorithm in Section 3. Multiple connections to the same "to" part, or a connection "to" the same part as "from" are acceptable, as long as they arise through normal random assignment. It is not acceptable to use a fixed-size array for the "from" references; the schema must work with any possible random database.

Clustering. We allow the connections to be clustered in physical storage according to the parts they connect. They may also be clustered with the parts themselves, if this gives better performance. We have made it easy for the benchmark implementor to do this colocation, because part connections have colocality based on part id: the best way to group parts is sequentially by part id. In fact, when we designed the benchmark we thought we had made this *too* easy, that we gave no advantage to a DBMS that could automatically colocate by connections. However, we realized that any benchmark implementor could build the equivalent ordering from any underlying colocation mechanism we devise and group the parts accordingly. So while using part id's for colocation may seem odd, it is not unrealistic or unfair, as we see it.

As mentioned earlier, it is not permissible to exploit the consecutive part id's in a way that would not work with a more sparse assignment of id's—for example, using them to index into an array of addresses. However, it is permissible to use physical pointers instead of part ids to represent the connections, as long as the benchmarks are faithfully reproduced (e.g., the parts are actually fetched in the traversal benchmark).

In the insertion benchmark (3), the 100 parts and 300 connections we create are added to the original database, i.e., there will be a total of 20,100 parts in the database after the first execution of the benchmark. The three connections for each of these parts are selected using the same algorithm as the others, i.e., they are 90 percent likely to be to one of the parts with largest part id's, 10 percent likely to be to any existing part. The newly inserted parts and connections should actually be deleted in order to make the insert measure idempotent, i.e., so that there are not 20,200 parts after two executions. However, in our measurements we found the results were not noticeably larger with 21,000 parts, so ten iterations could probably be executed together.

Network load between the workstation and server machines should be minimized, though in our experience this had no measurable effect on the results. This result may initially be surprising, but is consistent with experience at Sun—the Ethernet is rarely close to saturation even with 100 workstations on a single wire.

Be careful with random number generation; some operating systems provide randomizing utilities with nonrandom characteristics in the low-order bits. We implemented the random number generator suggested by Park and Miller [7].

7. NOTES ON DBMSs

We experimented with three different DBMS products. The products vary quite dramatically in their overall design, implementation of concurrency control, and other "traditional" DBMS properties, so we hoped to learn something about architectural performance factors by comparing them. We now review specifics of each product.

INDEX. One system is a B-tree package we had available at Sun. Throughout the balance of the paper, we label this system *INDEX*. We include it to show what can be done under ideal conditions, using a highly optimized access method with no high-level semantics.

We implemented the benchmark database with two record files, one for parts and one for connections. We created a B-tree on the part id's in the parts file. We used this B-tree to perform the lookup operation.

In addition to B-trees, the INDEX package provides efficient lookup operations on record ids; record ids consist of a physical page number plus slot number on a page. For the traversal measures, we used the record ids for parent-child links as in System R [3]. Our connection records contained the type and length attributes, plus four record id attributes: two containing the record ids of the "from" and "to" parts it connects (its "parent" records), and two used to circularly link together all of the connection records for the "from" and "to" parts (the "child" records). The part records contained the id, type, x, y, and build attributes, plus two attributes containing the record ids for the first "from" and "to" connection records for the part.

Thus our INDEX version of the benchmark database consisted of a parts file, a connection file interconnected to the parts file with record ids, and a B-tree file used for the lookup on parts.

To compare the performance of B-trees against parent-child links, we also tried performing the traversal measure using B-trees: we created two more B-trees, one on the "from" field and one on the "to" field in the connections file. Because the parent-child link implementation was faster than the B-tree approach, we report the results from the former in the next section. The B-tree results are shown in the following section. *Note*: the overall database creation time, marked with "*" in our tables, reflects the time to build both the B-trees and links. The insert times reflect only the access method used.

The INDEX package works over a network, and is designed to be particularly fast when the data can be cached in workstation main memory, treating read-only data utilizing a novel network lock service to control access. The INDEX package caches an 8K contiguous page of records on each remote procedure call. The package provides update record and table level locks, but no update logging for concurrency control and recovery. We used the table level locks. Multistatement atomic transactions are not yet supported, nor are native security features. Of course, there is no query optimizer; it is the duty of the applications programmer to compute and program the fastest access paths, as described.

INDEX uses a single-process data access architecture. That is, the INDEX libraries are linked with the application program, and data is read directly from disk to the applications's virtual memory. A separate process is used for the lock manager only, and only one interprocess remote procedure call is required for the table lock used in the benchmark. INDEX uses this architecture even in the *remote* database case, by utilizing Sun's Network File System (NFS) to access database pages. In contrast, most UNIX DBMS products use a client/server architecture, requiring an interprocess call for every data access, copying data from the DBMS buffer to application memory.

The INDEX system, although powerful within its intended application domain, is not strictly comparable to full-function DBMSs. In an application for which a query language or transaction semantics are essential, it is not possible to use the INDEX package. For other applications, gross-level locks and direct use of access methods may be quite adequate. The INDEX results are interesting because they represent the best performance that might conceivably be obtained, in the case where the cost of high-level semantics is made negligible.

OOBMS. A second system we used is a prerelease ("beta") of a commercial object-oriented DBMS. We believe this system to be representative of the current state of the industry: it supports objects, classes, inheritance, persistent storage, multiuser sharing of objects, remote access, and transactions for concurrency and recovery. We call this system *OODBMS*. Since object-oriented products are new to the market, the numbers reported here should not be considered indicative of what will eventually be achieved with an object-oriented system.

We implemented the OODBMS benchmark with an object type Part containing variable-size arrays with the connection information. We created a hash index on the part id, used for the lookup measure. Object references (and reverse references) are stored by the OODBMS as object identifiers that contain a logical page address in the upper-order bits; the OID can normally be mapped to the physical page and offset of the referenced object with no extra disk accesses. The way we implemented the benchmark, OIDs are *not* automatically swizzled into pointers by the OODBMS the first time an object is used, as in some object-oriented DBMS products. However, the mapping from OIDs to memory address is very fast when the object has been cached, requiring only table lookups.

The type declaration for parts and the associated connections looks something like this in the OODBMS:[4]

```
Part. {
  id: INT,
  type· STRING[10],
  x, y· INT,
  build: DATE,
  to. LIST OF {p: Part, type: STRING[10], length. INT}
  from. LIST OF Part}
```

Like the INDEX package, the OODBMS used a single-process data access architecture: the OODBMS is bound directly with the benchmark application. In addition, the OODBMS provides close integration with the application programming language, C++ in this case. Part objects appear to the programmer as conventional C++ objects, except, that they are persistent and support transaction semantics. The disk representation of a C++ object is essentially identical to the in-memory representation, so that objects can be paged into application memory with a minimum of overhead.

We used the transaction mechanism provided by the OODBMS, implemented using shadow pages and journalling. Transaction rollback and roll-forward are supported. The OODBMS system works remotely over a network, fetching pages rather than objects, and caches pages on the workstation. A variable page size was supported; we used 2K pages for the results shown here, but the results did not differ much with 1K or 4K pages. Objects may be grouped in segments, with locking at a segment level. For the benchmark, we put all of the parts and connections in one segment. Page-level locking is being implemented; it will be informative to run the benchmark with

[4] The syntax has been simplified for readability

these finer-grain locks. We predict there will not be substantially slower performance, because the implementation will use a page server in which lock calls can be piggy-backed on the page reads.

RDBMS. The final system we used, named *RDBMS*, is a UNIX-based production release of a commercial relational DBMS. The benchmark measures were implemented in C with SQL calls for the lookup, traversal, and insert operations. The RDBMS, like most current relational DBMS products, is implemented using a client/server architecture, i.e., each database call from the application requires an interprocess call to the DBMS server. Queries were precompiled wherever possible, thus bypassing most of the usual query parsing and optimizations, or at least incurring these costs only on the first execution.

We set up two tables, Part and Connection, indexed on id and from, respectively. A single secondary index was also created, on the to attribute of the Connection table; this permitted reasonable response times on the reverse traversal measure. We used B-trees as the storage structure for all three tables; the RDBMS did not provide parent-child links or another more efficient access method for the traversal measure. We used 100 percent fill factors on both index and leaf pages, and the page size for both indexes and data was 2K.

The RDBMS provided a relatively full set of DBMS features: atomic transactions, full concurrency support, network architecture (TCP/IP for our network benchmarks), and support for journalling and recovery. For the lookup and traversal benchmarks, we programmed the system to not place read locks, and we measured the system with the roll-forward journalling features turned off. We believe these settings make sense for the model of engineering interaction that we envision.

We refer to our implementation of the benchmark on the RDBMS as a "straightforward" implementation, in the sense that we used the most obvious implementation we would expect an application programmer to try. In particular, there are two important limitations of our implementation that may or may not be justified, depending on the application environment and programmers involved:

(1) The RDBMS provides no way to cache data locally as in the OODBMS and INDEX. We considered implementing a cache ourselves, as a layer between the RDBMS and the application. However, in so doing we would effectively have to duplicate much of the functionality of the DBMS: index lookups by part id, connection links between parts, and so on. In a real application, which presumably requires much more than the three operations in the benchmark, the application programmer would have to implement even more—in the worst case duplicating most of the DBMS on top of the application cache: access methods, query language, versions, referential integrity, special types such as blobs, concurrency mechanisms to keep the cache consistent in a multiuser context, and so on. If application programmers are to convert CAD or CASE application to use

a DBMS, they would in our opinion, demand that the DBMS product work on cached data. However, a cache in the application may be acceptable to some users.

(2) DBMS calls are invoked many times for each measure, since the benchmark requires that we mix C and database operations, and the RDBMS uses a client/server architecture. We considered ways to execute many traversal operations as a single database call, or as a smaller number of DBMS calls, instead of 3280 calls. We could perform a breadth-first traversal in just seven database calls, returning the 3^N parts at the Nth stage of the expansion to the application program. However, this might not be very efficient, since the 3^N parts must be passed back again in the next query step, or a temporary table must be used to retain the parts on the server, invoking substantial update overhead (writes are many times more expensive than reads). A relational DBMS with procedural constructs in SQL and the ability to link application procedures with the DBMS could be used to perform the entire traversal as a single DBMS call that performs the required null application procedure calls on the server. On the other hand, this approach violates the intent of the *interactive* benchmark, as defined in Section 4, since much of the application would not execute on the workstation, nor would the server procedure have access to global variables, the screen, or other resources on the workstation.

In any case, we left these more sophisticated implementations, and arguments about whether engineering applications could realistically be decomposed in these ways, to future research. At the least, we believe our straightforward implementation is representative of what a general industry user would expect of a DBMS.

8. RESULTS

Table II shows the most important results, for the *small remote* database (i.e., 20,000 parts accessed over the network). As per our specifications, we performed the benchmark measures *cold*, after running a program to clear out any cached data in the operating system page cache as well as the DBMS.

The table also shows the *warm* case, the asymptotic results we obtained after running the benchmark ten times, as specified.[5] The relative weightings of the *cold* and *warm* results in choosing a database system are application-dependent. Most of the applications we encounter would have behavior between these two extremes.

The INDEX and OODBMS results are clearly fastest; both come close to our goal of ten seconds for executing each *cold* measure. Though the

[5] These asymptotic results were generally reached very quickly: with INDEX and OODBMS, where entire pages are added to the local cache with each data operation, we found that nearly the entire small database is in the cache after the first iteration of the lookup or traversal measures

Table II. Benchmark Results for *Small Remote* Database

Measure	Cache	INDEX	OODBMS	RDBMS
DB size		3.3MB	3.7MB	4.5MB
Load (local)		1100*	267	370
Reverse traverse	cold	23	22	95
Lookup	cold	7.6	20	29
	warm	2.4	1.0	19
Traversal	cold	17	17	94
	warm	8.4	1.2	84
Insert	cold	8.2	3.6	20
	warm	7.5	2.9	20
Total	cold	33	41	143
	warm	18	5	123

Note: Results in seconds unless otherwise noted

results are close, INDEX wins on the *cold* results. The OODBMS wins by a substantial margin on the *warm* results.

The OODBMS results are surprisingly good, considering this is an early version of a product with little performance tuning. The OODBMS uses efficient access methods (links for traversal, B-trees for lookup), minimum concurrency overhead (table locking), no overhead for higher-level semantics, a local cache, and no interprocess communication to make database calls. We speculate that OODBMS does so well on the *warm* results because it uses an efficient representation to find objects in memory (different from the disk representation).

Note that the only measure on which the OODBMS does not beat INDEX on the *cold* results is lookup. There is no obvious explanation for this, but we learned from discussions with the implementors that the OODBMS prerelease had a poor hash index implementation.

Although the lack of more complex concurrency and higher-level semantics in INDEX makes it inapplicable to some applications, the results are interesting in demonstrating the kind of performance that can be obtained where access methods can be applied directly. The overall results are quite good. The load time is much higher than the other systems, but that time (marked with "*") included the overhead to redundantly create B-trees (results, next section), in addition to the parent-child links for the connection traversals. The INDEX access methods used are similar to the OODBMS, as is its ability to cache the database on the workstation.

The RDBMS times are slowest, with an overall total of 143. The results reflect the overhead of accessing the DBMS for each operation as demanded by our straightforward benchmark implementation, approximately 5,000 calls altogether. At 20 ms for a roundtrip network call, this overhead amounts for most of the time. As we noted earlier, these results are not inherent in the relational data model, only in the database server architecture and programming language/database environment boundary overhead.

Both the OODBMS and INDEX come close to our goal of 1,000 read operations per second. Ideally, we would like all three measurements, lookup, traversal, and insert, in the 1–10 second range for good levels of human interaction performance. Generally speaking, 1,000 operations per second is an upper limit for performance on a database that does not fit in memory, even with a 90 percent cache hit rate, since good disk technology requires more than 10 ms to fetch data from the disk. Thus, we cannot hope to perform much better than a 1–10 second range on the *cold* lookup and traversal benchmarks (1,000 lookups, 3,000 traversals) on current technology.

Table III shows the results when we ran against a *large remote* database. Now, none of the products can make much advantage of a local cache.

Note that now there is much less difference between the three systems on either the *cold* or *warm* results. Nearly all of the results are within a factor of two. Furthermore, there is less speedup in the *warm* cases.

We believe that most of the speed decrease in the larger database results are due to the larger *working set*, i.e., that the database can no longer be cached, and not to scaling up of access methods with database size. Indeed, the link access methods used for traversal in INDEX and OODBMS should require only one disk access regardless of database size, unlike the indexes used for the lookup.

The most significant differences between systems on the *large remote* case is on the traversal: the INDEX and OODBMS results are faster than for the RDBMS. Since the locality of reference inherent in the traversal can provide benefits even when the database cannot be fully cached, and INDEX and OODBMS utilize links instead of index lookups to traverse connections, these differences do not surprise us.

As an experiment, we ran the results for the *huge remote* database as well. Because the time to create the database is prohibitive, we did this for the INDEX package only. The results are shown in Table IV.

The overall results are over twice as slow as the *large* database in the *cold* case, and over four times as slow in the *warm* case. Again, we think most of this result is due to inability to cache the larger working set. Note that there is now essentially no difference between *cold* and *warm* times—a fact that supports this hypothesis.

With a 0 percent cache hit rate, 20–25 ms to read data off the disk, 15–20 ms to fetch a page over the network, one page access for the traversal operation, and two page accesses for the lookup (one for the index, one to fetch the data fields), the lookup measure would take 80 seconds and the traversal about 120 seconds. Note that this is very close to the results we actually obtained for the huge database. This suggests that the results might not get too much worse with a "very huge" database, but we do not have the patience to try that!

9. VARIATIONS

In this section we consider some variations of the benchmark configuration, beyond the results required by the benchmark definition, to better understand the DBMSs we compared.

Table III. Benchmark Results for *Large Remote* Database

Measure	Cache	INDEX	OODBMS	RDBMS
DB size		33.1MB	37.0MB	44.6MB
Load (local)		16400*	6000	4500
Reverse traverse	cold	100	84	212
Lookup	cold	47	49	49
	warm	18	43	24
Traversal	cold	56	68	135
	warm	41	59	107
Insert	cold	20	10	24
	warm	18	7.5	24
Total	cold	123	127	208
	warm	77	110	155

Note: Results in seconds

Table IV. Benchmark Results for *Huge Remote* Database

Measure	Cache	INDEX
DB size		330MB
Load		78000 +
Lookup	cold	109
	warm	106
Traversal	cold	143
	warm	142
Insert	cold	53
	warm	53
Total	cold	305
	warm	301

Note: Results in seconds

Table V shows the results for a *local* database, i.e., a database stored on disks attached directly to the user's machine. These results are important for some engineering applications, even though we claim most will require remote data access.

These results surprised us; we expected the results in the *local* case to be much better than they were, especially for the RDBMS.

The RDBMS results were essentially the same as the *remote* case. We expected them to be better because it is no longer necessary to make a remote call to the DBMS for every operation. However, the network call may not be the largest portion of the overhead in an application DBMS call. Even on one machine this RDBMS, as with nearly all other RDBMS products, requires communication between an application process and the DBMS process, and copying data between the two. There is generally significant overhead in such an interprocess call. In addition, the benchmark application is competing for resources on the server machine in the *local* case.

Table V. Benchmark Results for *Local* Databases (Cold)

Measure	Size	INDEX	OODBMS	RDBMS
Lookup	small	5.4	12.9	27
	large	24	32	44
Traversal	small	13	9 8	90
	large	32	37	124
Insert	small	7 4	1.5	22
	large	15	3.6	28
Total	small	26	24	139
	large	71	73	196

Note: Results in seconds

We also guessed that the INDEX and OODBMS results would be much better than for the *remote* database, but they are only 30–40 percent faster. This illustrates that remote access to files is not much more expensive than local access. We performed some rough measurements running an application reading pages locally and remotely through NFS, and found only a 30 percent difference in speed. Of course, this rough measurement does not include time for locking or to flush pages back remotely on the insert results. Note that OODBMS beats INDEX by a small margin in the *small local* case, though it lost in the *small remote* case; this switch is probably a result of minor differences in implementation.

In order to better understand benchmark performance, we did two additional experiments. We used INDEX for these experiments since its overall results were good, it was easy to work with, and we had the source code available so that we could be certain of its internal architecture.

First, we examined the effects of the locality of reference that exists in the connections between parts, to see its effects on the the traversal and other measures. Table VI shows these results.

Note that the result for the traversal (24 in Table VI versus 17 seconds in Table II, for the *cold* case) and insert (36 versus 8 seconds) differ significantly without locality of reference in the *small remote* database. The traversal benefits from the likelihood that connected parts are frequently nearby, with the page-level caching provided by INDEX; the insert benefits because it also must fetch connected parts to create the parent-child links that we use for traversal. Since the results without locality differ so much, the addition of reference locality to our benchmark would seem to be an important one.

Note there is little difference on the lookup numbers; this is as expected, since the lookups are randomly distributed over all the parts. There is also little difference in the case of the *warm* numbers: this is not surprising, since the entire database is cached in memory at this point and locality is irrelevant.

On the *large or huge* database, we would expect there to be similar or larger differences without locality of reference, though we did not test these cases. Note that the *large* and *huge* database results are worse for at least two reasons: the database does not fit in memory and (in some cases) access

Table VI. *Small Remote* Database Without Locality of Reference

Measure DB size	Cache	INDEX 5.0MB
Lookup	cold	7.9
	warm	2.4
Traversal	cold	24
	warm	7.9
Insert	cold	36
	warm	10
Total	cold	68
	warm	20

Note: Results in seconds

structures such as B-trees take longer to traverse. Only the former is substantially affected by locality.

We were curious as to the effect of using B-trees instead of parent-child links to perform the traversal measure. Intuitively, we expect the links to be faster since they can obtain the connection records and part records in a single disk access (or a few memory references, if cached), while the B-trees require at least one more page access and significantly more computation (to compare keys). In Table VII, we show the results we found when we created B-Trees on the "from" and "to" fields of connections and used them for the traversal.

As expected, the traversal is significantly slower: the parent-child links are about twice as fast as B-trees. The insert time does not differ substantially (the time for inserts into connection B-trees is comparable to the time to find connected parts and create links to them) and the lookup times are of course not affected.

This is a significant result, as it shows that the mainstay access method of relational DBMSs, B-trees, is substantially worse than links for applications that require traversal of many objects. With a larger database we would expect the difference to be even larger, since links take a constant time to traverse, while B-tree lookup time grows logarithmically with database size. Thus DBMSs for engineering applications should provide link access methods or some other access method that is efficient for traversal (extendible hash indexes might be adequate).

It should also be noted that B-trees, as used by the relational DBMS, do not preserve an ordering on the entries as do the parent-child links. The part connections in the OODBMS and the parent-child link version of INDEX can be maintained in any desired order; in our benchmark implementation, the connections are traversed in the same order in which they were created. If we added ordering to our OO1 specification, the RDBMS performance would be even slower than in our results—the RDBMS would have to sort the connections during the traversal operation.

In the current version of OO1, we do not require that an ordering be maintained on the connections, nor do we require that the traversal operation

Table VII. *Small Remote* Database Usıng B-Trees instead of Links

Measure DB size	Cache	INDEX 3.9MB
Lookup	cold	7.9
	warm	2 4
Traversal	cold	33
	warm	21
Insert	cold	12
	warm	9 1
Total	cold	53
	warm	33

Note: Results in seconds

be performed in a particular order. It would be interesting, for future work, to measure an "ordered" version of the traversal. There are many applications where such an ordering would be important, for example if the parts were portions of a document with a hierarchy of chapters, sections, and paragraphs, or if the parts were statements and blocks in a program.

10. SOME COMPARISONS

At the time of this writing, other groups have run OO1 on a variety of object-oriented databases, relational databases, and access methods. The results we have seen have been consistent with those reported here. Appendix A shows some results reported on products from Object Design, Objectivity, Ontologic, and Versant on the *small* database. Note that the results on these object-oriented database systems are very close, but a couple of these systems are faster for lookup and traversals in the *warm* case. In fact, one object-oriented DBMS that incorporates swizzling was found to run at essentially the same speed as conventional in-memory C++ data structures.

In comparing DBMSs, it is important to keep in mind that our benchmark is intentionally simple, in order to make it easy to be widely reproduced—and it is only a rough approximation to the requirements of engineering applications. The only truly accurate representation of performance would be to actually execute the application; our goal is merely to focus attention on gross-level issues in the performance of different database architectures. However, OO1 is based on past experience and interviews with CASE and CAD engineers, and bears close resemblance to other engineering benchmarks we have examined [4, 8]. We computed a correlation coefficient of .981 between the latter benchmark and OO1, based on the results of five object-oriented DBMS products.

We would like to emphasize that our results do *not* show that an object-oriented data model is better than a relational data model for engineering applications. We can think of only two limitations of the relational model itself that might be a problem for engineering applications: (1) the relational model provides no "manual" ordering capability, as mentioned earlier, and

(2) using the relational model for the database may necessitate translation overhead between the tabular representation in the database and the data structure representation in the application programming language. Our benchmark provides no evidence to validate or quantify these effects. The order of magnitude difference we see in the OODBMS and RDBMS in our results appear to result from a difference in the DBMS *architecture* rather than *data model*. Furthermore, the addition of other operations (such as ad hoc queries) to OO1 might change the balance of the results, as would a different way of using the relational DBMS.

In this regard, Winslett and Chu [11] report results that might appear contradictory to those presented here. They ported a VLSI CAD system, MAGIC, to run on top of a relational DBMS, UNIFY, and found that MAGIC ran less than 25 percent slower. On OO1, we found that a relational DBMS ran several times slower than other systems. However, the contrast in these results actually support many of the claims we have made; our experiments were different than Winslett and Chu's in several important ways.

One important difference is that Winslett and Chu used a *batch* model of interaction with the DBMS, in which the application fetched all data from the DBMS into its own in-memory structures, operated on the data for some period of time, then dumped the data back to the relational representation. As explained in Section 4, the results reported in this paper are based on an *interactive* version of OO1, operating directly on objects in the database.

In addition, UNIFY differs from the RDBMS we used in two important ways:

(1) UNIFY, unlike most UNIX DBMSs (and a newer UNIFY product from the same company) is bound directly with the application program, thus minimizing overhead to call the DBMS. It also keeps data in the same address space.
(2) UNIFY, unlike most UNIX DBMSs, provides parent-child links.

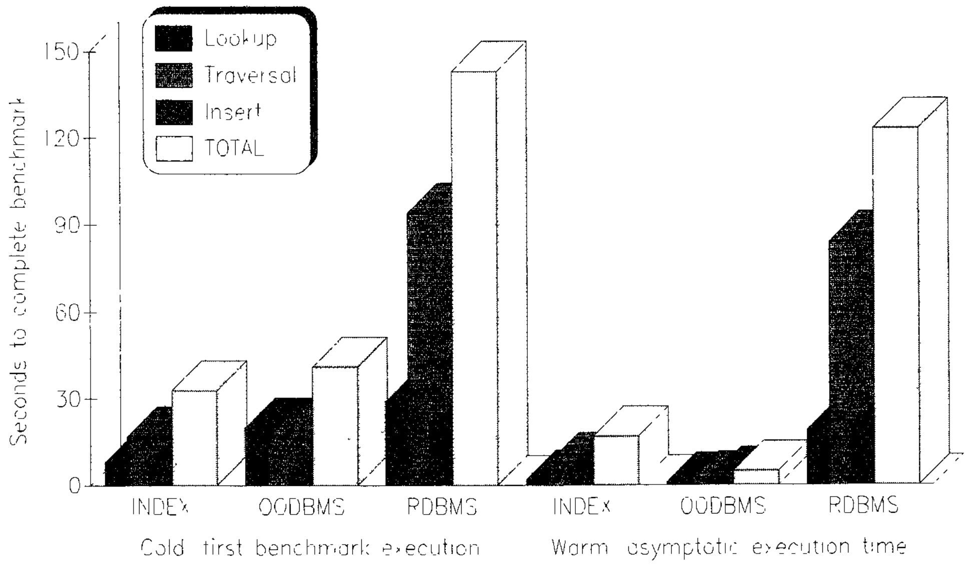
Thus, current relational products should perform better on OO1 by directly binding with an application program, caching large amounts of the database in main memory, and providing new access methods.

11. SUMMARY

We would like to see more emphasis on performance for engineering databases. On the basis of our benchmarking experience and study of engineering applications, we propose that three simple measures representative of such applications are lookup, traversal, and insert operations, performed upon a remotely located database with a relatively small working set.

Our OO1 benchmark differs architecturally from other benchmarks in important ways for engineering applications: (1) the database is remotely located, (2) the programming language must be intermixed with data operations at a fine grain, (3) conventional transactions may be replaced with versions or other concurrency control, and (4) the entire database working set can sometimes fit in main memory. These differences necessitate

Table VIII. Small Remote Database Results with Different Systems



violations of conventional wisdom (e.g., for database servers) in order to obtain acceptable performance.

The main results of the paper, from Table II, are summarized graphically in Table VIII.

Table VIII illustrates that the prerelease of the commercial OODBMS performed well, as did the home-grown INDEX access method, demonstrating that our goals for engineering DBMS performance are not impossibly high. The commercial RDBMS ran more slowly than these systems.

The experimental variations we performed on the benchmark using the in-house INDEX package lend further support to some of these conclusions about database architecture. The results, taken from Section 9, are shown graphically in Tables IX and X.

Notice the deterioration in the results in Table IX as the database size grows. We found that the majority of this deterioration was directly attributable to a dramatic reduction in cache hit rate with the larger databases. The INDEX results with a *small local* database (Table X) are better than for the *small remote* database (Table IX); a similar difference was found with the OODBMS. All these data support our contention that a local cache is important. However, it is also important to minimize the overhead per DBMS call in general; the RDBMS ran slowly even in the *local* case.

Using B-trees instead of parent-child links for the benchmark produces significantly worse results for traversal, with only minor effects on the other measures. This supports the claim that it is important to consider alternatives to B-trees for logical connections that are traversed in a "transitive-closure" type operation.

Without locality of reference, the results on the traversal measure are somewhat worse. This shows that the inclusion of locality of reference in a benchmark is important where applications exhibit such locality.

Table IX. INDEX Results for Remote Access to Small, Large, and Huge Working Sets

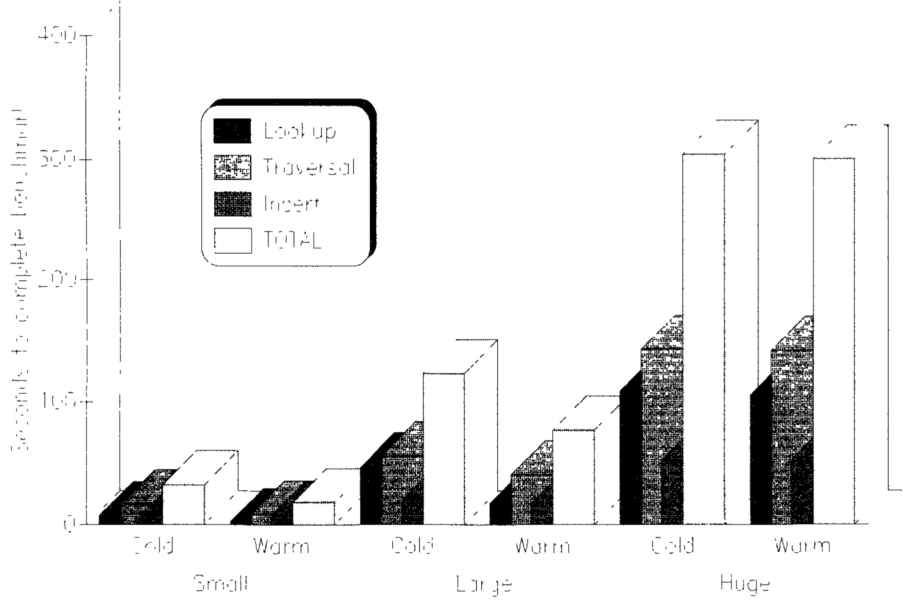
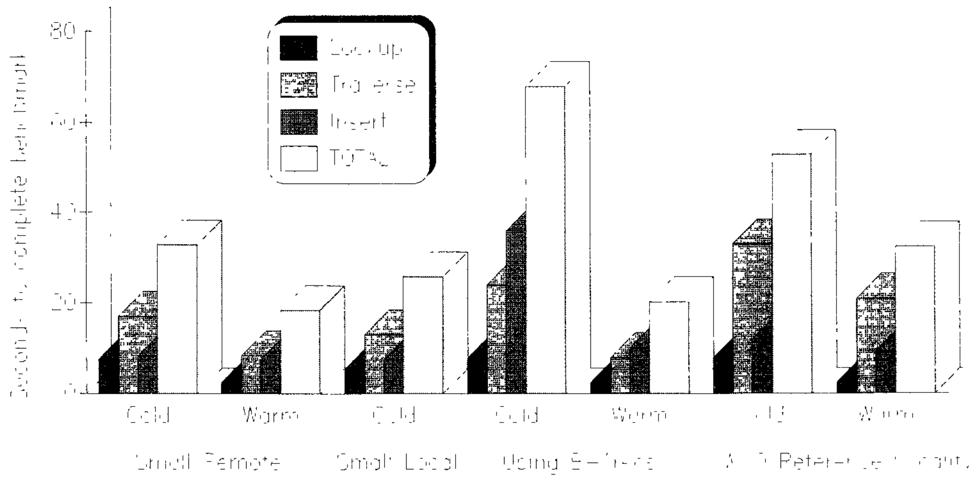


Table X. INDEX Results for Small Database Locally, Remotely, Using B-Trees Instead of Links for Connections, and Without Locality of Reference in Connections



To summarize: Good performance on the lookup, traversal, and insertion operations in OO1 requires effective use and integration of the programming language data space, new access methods, concurrency control allowing long-term check-out, and caching on a local workstation. We contend that database systems will not be used in many engineering applications unless they can perform benchmark measures such as ours in 1–10 seconds with a

90 percent cache hit rate on commonly used workstations and servers. We have shown that such results are possible. We have also shown that products can fall short of these results by as much as an order of magnitude. The essential message in this paper, more important than the results and the observations about database architectures, is to bring attention to the performance needs of engineering applications through the set of measures we define to compare DBMS systems.

APPENDIX A: OTHER RESULTS

The OO1 benchmark was run on Objectivity/DB, Object Design ObjectStore, Ontologic ONTOS, and VERSANT™ in early 1990. The results are shown here without identifying the products, since the purpose of these numbers is to show the performance of the new products relative to relational DBMSs, and not to use as a basis to choose among them. The numbers are not strictly comparable because some were not verified on Sun premises, and slightly different Sun-3 configurations were used. OODB4, RDBMS1, and INDEX are the results from the paper, repeated for comparison. OO1 has been run on half-a-dozen other systems as well, but we did not verify the source code and/or could not obtain permission to reproduce their results here.

OO1 Benchmark Results in Seconds: Small Remote Database

Measure	OODB1	OODB2	OODB3	OODB4	RDBMS1	INDEX
Lookup cold	18	13	22	20	29	7.6
Traverse cold	27	13	18	17	94	17
Insert cold	14	6.7	3.7	3 6	20	8.2
R-traverse cold	34	13	16	22	95	23
L + T + I cold	59	33	44	41	143	33
Lookup warm	0.1	0.03	1.1	1.0	19	2.4
Traverse warm	0.7	0 1	1 2	1 2	84	8.4
Insert warm	3.7	3 1	1.0	2.9	20	7.5
L + T + I warm	4.5	3.2	3.3	5.1	123	18

OO1 Benchmark Results in Seconds: Small Local Database

Measure	OODB1	OODB2	OODB3	OODB4	RDBMS1	INDEX
Database size	5 6MB	3.4MB	7.0MB	3 7MB	4.5MB	3.3MB
Build time	133	50	47	267	370	1100 *
Lookup cold	12	10	10	13	27	5.4
Traverse cold	18	10	8 3	9.8	90	13
Insert cold	9	5.3	1 9	1.5	22	7 4
L + T + I cold	39	25	20	24	139	26

Note: *local warm* results essentially same as *remote warm*.

™ Objectivity/DB, ObjectStore, ONTOS, UNIFY, and VERSANT are registered trademarks.

APPENDIX B: SQL DEFINITIONS

Database Definition

```
create table part (
  id int,
  part_type char(10),
  x int,
  y int,
  build datetime)

create table connection (
  frm int,
  too int,
  conn_type char(10),
  length int)
create unique clustered index ipart on part(id) with fillfactor = 100
create clustered index iconn on connection(frm) with fillfactor = 100
create nonclustered index rconn on connection(too) with fillfactor = 100
```

Stored Procedures

```
CREATE PROCEDURE engl_lookup
    @num int,
  @id int = 0 OUTPUT,
  @x  int = 0 OUTPUT,
  @y  int = 0 OUTPUT,
  @part_type char(10) = "abc" OUTPUT
AS

SELECT id, x, y, part_type
FROM part
WHERE id = @num
CREATE PROCEDURE engl_ftrav_conn
    @partid int,
  @frm int = 0,
  @too int = 0 OUTPUT,
  @length int = 0 OUTPUT,
AS

SELECT too, length
FROM connection
WHERE frm = @partid

CREATE PROCEDURE engl_part_insert
    @id int,
  @part_type char(10),
  @x int,
  @y int,
  @build datetime
AS

INSERT part
(id,
part_type,
x,
y,
build
)
VALUES
(@id,
@part_type,
@x,
@y,
@build
)
```

Example of C Code: Lookup

```
lp_start = time100( );
for (i = 0; i < nrepeats; i + + ) {
  genrandom(nparts, nrecords); /* Generate random numbers */
  starttime = time100( );
for (j = 0; j < nparts; j + + ) {
    dbrpcinit(dbproc, "engl_lookup", (DBSMALLINT)0);
      dbrpcparam(dbproc, NULL, (BYTE)0, SYBINT4, - 1, - 1,
      &rand_numbers[j]);
  dbrpcsend(dbproc);
  dbsqlok(dbproc);
  dbresults(dbproc);
  dbnextrow(dbproc);
  p_id = *((DBINT*)dbdata(dbproc, 1));
  p_x = *((DBINT*)dbdata(dbproc, 2));
  p_y = *((DBINT*)dbdata(dbproc, 3));
  strcpy(part_type, (DBCHAR*)dbdata(dbproc, 4));
  nullproc(p_x, p_y, part_type);
}
endtime = time100( );
printf("Total running time for loop %d is %d.%02d seconds", i + 1,
  (endtime - starttime)/100,
  (endtime - starttime)%100);
  starttime = time100( );
}
dbexit( );
endtime = time100( );
printf("Total running time %d.%02d seconds",
  (endtime - lp_start)/100,
  (endtime - lp_start)%100);
```

REFERENCES

1. ANDERSON, T., BERRE, A., MALLISON, M., PORTER, H , AND SCHNEIDER, B. The HyperModel benchmark, In *Proceedings Conference on Extending Database Technology* (Venice, March 1990), Springer-Verlag Lecture Notes 416
2 ANON ET AL. A measure of transaction processing power. *Datamation. 31*, 7 (April 1, 1985).
3 ASTRAHAN, M , ET AL. System R: Relational approach to database management *ACM Trans. Database Syst. 1*, 1 (1976).
4. BARRETT, M. C + + Test driver for object-oriented databases, Prime Computervision, personal communication, 1990.
5 BRITTON, D., DEWITT, D. J., AND TURBYFILL, C. Benchmarking database systems: A systematic approach. In *Proceedings VLDB Conference* (October, 1983) pp. 000-000. Expanded and revised version available as Wisconsin Computer Science TR 526.
6. MAIER, D. Making database systems fast enough for CAD applications. Tech Rep CS/E-87-016, Oregon Graduate Center, Beaverton, Oregon, 1987

7. PARK, S., MILLER, K. Random number generators: Good ones are hard to find, *Commun. ACM. 31*, 10 (Oct. 1988).
8. SCHWARTZ, J., SAMCO, R. Object-oriented database evaluation, Mentor Graphics, personal communication, 1990.
9. RUBENSTEIN, W. B., KUBICAR, M. S., AND CATTELL, R. G. G. Benchmarking simple database operations. In *Proceedings ACM SIGMOD* (1987).
10. TRANSACTION PROCESSING PERFORMANCE COUNCIL (TPC), "TPC Benchmark A Standard", Shanley Public Relations, 777 N. First St., Suite 600, San Jose, California, November, 1989.
11. WINSLETT, M., AND CHU, S. Using a relational DBMS for CAD data. Tech. Rep. UIUCDCS-R-90-1649, Computer Science, Univ. of Illinois, Urbana, 1990.

Received December 1988; revised January 1991; accepted January 1991