The Engineering Database Benchmark

R. G. G. Cattell

1. Introduction

Performance is a major issue in the acceptance of object-oriented and extended relational database systems aimed at engineering applications such as Computer-Aided Software Engineering (CASE) and Computer-Aided Design (CAD). Because traditional database system benchmarks (Bitton, DeWitt, & Turbyfill [BITT84], Anon *et. al.* [ANON85], TPC [TPC89]) do not measure the performance of features essential to engineering applications, we designed an engineering database benchmark. The benchmark, named EDB (for Engineering Database Benchmark), has been run on a dozen database products and prototypes. This chapter describes EDB and the results obtained running it on a relational and on an object-oriented DBMS. This chapter derives from a paper by Cattell and Skeen [CATT91]. EDB is a redesign of an earlier, more primitive benchmark described in Rubenstein, Kubicar, and Cattell [RUBE1987].

The EDB benchmark is important because traditional DBMSs are a factor of ten times to one hundred times too slow for many engineering applications when compared to DBMSs specifically designed for engineering applications. Functionality of new DBMSs aimed at engineering applications is not useful when the performance shortfall is large. Engineering applications are of particular interest to us, so we sought to bring focus on these performance issues. This chapter not only provides a careful specification and demonstration of the EDB benchmark; it shows an order of magnitude difference in performance can result by reducing overhead in database calls, implementing new distributed database architectures, taking advantage of large main memories, and using link-based access methods.

Measuring engineering DBMS performance in a generic way is very difficult, since every application has somewhat different requirements. However, most engineering applications are quite similar at the data access level. The operations they perform on data are not easily expressed in the abstractions provided by SQL and other high-level DMLs. Instead, engineering applications typically utilize a programming language interspersed with operations on individual persistent data objects. Although database access at this level is in some ways a step backward from modern DMLs, it is a step forward from current ad hoc implementations and is the best solution available.

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 language statements.

2. Engineering Database Performance

The EDB benchmark differs in a number of ways from the TPC-A and Wisconsin benchmarks described earlier in this book. TPC-A is designed to measure transaction throughput with large numbers of users, while EDB focuses on an engineer with negligible contention with other users. Some of the Wisconsin measures are relevant to engineering performance, but they generally are measures at too coarse a grain, and focus on the intelligence of the query optimizer on complex queries. Such set-oriented operations are rare in current engineering applications.

The most accurate measure of performance for engineering applications would be to run an actual application, representing data in the manner best suited to each potential DBMS. However, 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 (Maier [MAIE87]).

The generic benchmark measures in EDB are operations expected to be most frequent in engineering applications, based on interviews with CASE and CAD engineers and feedback on the earlier Sun benchmark (Rubenstein et.al. [RUBE87]). EDB substantially improves upon the earlier work, simplifying and focusing the measurement on these engineering database operations. An effort by Anderson et. al. [ANDE89] has also been made to improve on the earlier benchmark, however that work has gone in the direction of a more comprehensive set of measures rather than simpler ones. These benchmarks are contrasted in Section 5.

The EDB measures include inserting objects, looking up objects, and following connections among objects. Engineering applications may also require more complex or ad hoc queries, so

performance on EDB alone does not make a system acceptable. However, current systems are orders of magnitude away from acceptable performance levels for engineering applications, so we aim to focus attention on the area where the highest performance is required, just as the TPC-A benchmark focuses on commercial on-line transaction processing.

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. It is designed to be scaleable, to be representative of small database working sets that can be cached in main memory and large ones that require efficient access methods. To make it portable, the benchmark is defined in terms of ANSI C and ANSI SQL. Any equivalent implementation is allowed (see the Appendix).

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 random read queries in tenths of a second. In contrast, simple lookups using in-memory data structures can be performed in microseconds. This factor of 100,000:1 difference in response time is the result of many factors, not simply disk access. There is a place for a database system that fills the gap between these two systems, performing close to 1000 simple random read queries per second on typical engineering databases. This requirement comes from a desire to access complex objects consisting of thousands of components within a few seconds.

How can much higher performance be achieved? Primarily, by exploiting characteristics of engineering applications, which are quite different than business applications. For example, an engineer may "check out" part of a design and work on it for hours, 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 accomplished through minor improvements in data model, physical representation, or query languages. The substantial improvements result from major changes in DBMS architecture: caching a large working set of data in main memory, minimizing overhead in transfering data between programming and query languages, implementing data access and concurrency on a network at a page level to reduce overhead, 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. The Benchmark Database

The EDB benchmark database is 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.

The database is defined as two logical records:

```
create table part (  id         integer       not null primary key,
                     type       char(10)      not null,
                     x          integer       not null,
                     y          integer       not null,
                     build      datetime      not null
                     );
create table connection (
                     from       integer       foreign key references (part.id),
                     to         integer       foreign key references (part.id),
                     length     integer       not null,
                     type       char(10)      not null,
                     primary key(from,to, length)
                     );
```

A database N of parts will have dense unique part numbers (*part.id*) in the range [1..N]. Such a database will be 3•N 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.

It is assumed that the database system allows data fields with the scalar types specified above, where `integer` is a 32-bit integer, `date` is some representation of a date and time, and var`char(n)` is a variable-length string of maximum size `n`. The `from` and `to` fields of the connection relation 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% of the connections are randomly selected among the 1% 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. The following algorithm selects 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.

```
halfPercent := N/200;
for part in [1..N]
  begin
  for connection in [1..3]
    begin
    if rand[1..10]>1 then   /* 90% of connection to closest 1% */
      begin
      cpart := part - halfPercent + rand[1 .. 2*halfPercent + 1];
      /* "double up" at the ends to stay in part-id range */
      if cpart < halfPercent then cpart := cpart + halfPercent;
      if cpart > N - halfPercent then cpart := cpart - halfPercent;
      end;
    else            /* 10% of time connect to any part [1..N] */
      cpart := rand[1..N];
    end;
    connect(part, cpart, length, type);         /* add connection */
  end;
```

Using part-ids for locality is quite arbitrary; we justify why this is as good as any other criterion in Section 6.

A database of 20,000 parts comprises approximately 2 megabytes of attribute data, plus space overhead for access methods and record storage, which typically double this amount. As such, the typical database used in the experiments reported here 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 benchmark must also scale up to larger databases and working sets, exploiting access structures that use secondary memory efficiently. A *large* database is therefore included in the benchmark, identical to the smaller one except that all of 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, since they do not utilize large main memory caches and their access methods scale up well with database size. Other systems, e.g. a persistent programming language, may not even permit databases of this size.

Results for a *huge* 400 megabyte database are also examined to test performance on larger databases. However, the huge database is not part of the EDB benchmark requirements, since the results were found to differ less dramatically than the first two.

Unlike most other benchmarks, the EDB specification requires that remote database access be implemented on a network. The results for both local and remote access to data must be reported, i.e. for a configuration in which the data is on disks attached to the same machine as the benchmark application, or on another machine. Since most engineering applications run on a workstation with the data on a server, remote access is actually the more important configuration. The experimental results reported in section 8 show that performance differs significantly from a benchmark with only local data: remote access is substantially impacted by the caching and concurrency control architecture of the DBMS.

4. Benchmark Measures

The benchmark measures *response time* and is run by a single user.[1] This is consistent with the model of an engineer "checking out" a segment of data for exclusive use; indeed, adequate performance 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* multi-user access to the database. EDB requires that the data used be locked or versioned in order to support multiple users.

The following three operations are the EDB benchmark measures. Each measure is run ten times, measuring response time for each run to check consistency and caching behavior.

Lookup. Generate 1000 random part ids 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.

Traversal. Find all parts connected to a randomly selected part, or to a part connected to it, and so on, up to 7 hops (total of 3280 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. Perform the traversal depth-first. Also measure time for *reverse* traversal, swapping "from" and "to" directions, to compare the results obtained.

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

[1] 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.

There is some value to a single "overall" performance time, as with the TPC-A benchmark. The benchmark is designed so that the three measures are 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, it is important not to 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 the specification requires that the host programming language 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 the total wire lengths, and (3) compute the x,y position for placing the part in a diagram.

The requirement to call the host application programming language may be controversial, but it is an important feature of the EDB benchmark. Using current relational DBMSs, frequent DBMS calls from an application program can only be reduced by copying all the desired data from the database in a single super-query, or by embedding a portion of the application program (that cannot be expressed in the query language) as a procedure executed within the query on the database server. Copying all the data at once is not practical for most engineering applications because of the limited expressiveness of the query language and the overhead of copying data between the environments. Embedding application procedures in queries is not practical because the procedures cannot interact with the rest of the application program, and must perform computation on the server instead of utilizing the distributed workstations. In a real application, a "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.

The benchmark measures are designed to be representative of the data operations in common 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, and so on.

The benchmark measures must be performed at least 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, the average cache hit rate for the first iteration can be as low as 10% or as large as 90%.

The results of the first iteration are called the *cold* start results, and the asymptotic best times (the tenth iteration, when the cache is fully initialized) are called the *warm* start results. In the systems we measured, the warm asymptote was achieved in only 2 or 3 iterations, with only small (under 10%) 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 cluster parts in the database so that the forward traversal in measure (2) may go faster than the reverse traversal. It is 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 3 parts it is connected "to", but a random number of parts it is connected "from" (3 on average). However, the average should be similar to the forward traversal. It is convenient to count the number of nodes visited on the reverse traversal and then to normalize the result to obtain comparable figures. Since 3280 parts are visited in the forward traversal, the reverse traversal results can be normalized by multiplying 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-ids 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 require the atomic transactions of commercial DBMSs, or whether some form of versions or locking plus logging of updates is better. The EDB benchmark requires that (1) the DBMS provides serializability consistency under concurrent access, e.g. by locking data read and written, (2) the DBMS supports some form of recovery for the insert measure, backing out updates if a failure occurs before all 100 parts and associated connections are inserted, and (3) these concurrency and recovery constraints are satisfied with multiple readers and writers on a network of machines. A transaction mechanism satisfies these constraints, but different implementations (e.g. based on new versions of data) are also acceptable. If the DBMS provides more than one choice, it is important to know the performance of each alternative .

The benchmark measures must be performed on both the small and large database, as described earlier. Both the space and time requirements must be reported.

The data must be located on a machine different than the user. For example it must be on a remote database server on a network. Results may also be reported for a local database, as is done in Section 9; but, we believe the remote results are a more realistic model of new engineering and office applications.

5. Benchmark Justification

This section justifies many of the choices made in the design of EDB, and contrasts EDB with our earlier benchmark and with the HyperModel benchmark.

5.1. Earlier Benchmark

The EDB benchmark is both simpler and more realistic than that defined in Rubenstein, Cattell, and Kubicar [RUBE86]. The database itself is similar, although EDB incorporates locality of reference and uses 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 intended applications).

EDB drops two measures included in our original paper. The *range lookup* measure found the parts in a ten-day range of *part.build* dates. In our experience the result of this measure is very similar to the EDB *lookup* measure, as B-tree and hash performance do not differ greatly; 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). The *sequential scan* was also dropped; it enumerated all parts in the database to fetch the x,y position for each.

The *reference* and *group* lookup measure in the original benchmark were replaced with the EDB *Traversal* measure. 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* now in EDB, by composing individual *Reference* and *Group* lookup times. In practice, it was hard to specify these measurements in 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, they were replaced with the *Traversal* measure.

We considered including *update* (in addition to *Insert*) of part records, but found the results for update and insert to be closely correlated. So only *Insert* is present in EDB.

Thus, we chose the *Lookup*, *Traversal*, and *Insert* operations as sufficient for a simple, focused benchmark. This eliminated *range lookup*, *sequential scan*, and *update* measures from the set. However, these additional measures may be important or significant for some applications or systems, so we recommend they be included in a complete application study (e.g., Anderson et. al. [ANDE89]). In particular, it is essential that a DBMS include a B-Tree or other access method that supports range lookup, because these operations occur in many engineering applications.

5.2. HyperModel Benchmark

The HyperModel Benchmark (Anderson et. al. [ANDE89]) extended the early benchmark and database defined in Rubenstein, Cattell, and Kubicar [RUBE86], rather than simplifying it. These extensions resulted in 17 benchmark measures, four database entities types (tables like the EDB part table), and four relationships (tables like the EDB connection table). As of 1990, the HyperModel has not been widely implemented. It has been used to measure the local-only behavior of two DBMSs. This is probably because HyperModel is more complex and harder to implement. Also, the specification is not as complete as EDB's. If work continues on the HyperModel benchmark, however, it could be a more thorough measure of engineering database performance than EDB.

The difference between EDB and HyperModel might be likened to the difference between TPC-A (TPC [TPC89]) and the Wisconsin benchmarks (Bitton, DeWitt, and Turbyfill [BITT84]) -- the intent of EDB is to focus on overall performance for the few most important operations, while the HyperModel benchmark provides many different measures. As such, there is a place for both the HyperModel benchmark and EDB in comparing systems. It is easy to port the EDB benchmark to any DBMS for which the HyperModel benchmark has been implemented, since the database

and measures are more or less a subset of HyperModel (mapping "nodes" to “parts”, and the "MtoN" relationship to “connections”).

We will not enumerate all of the differences in the HyperModel benchmark, but note that:

(1) The HyperModel database includes multiple relationships between objects, including a 1-to-many sub-part relationship, and a sub-type relationship. It also includes a "blob" field that stores a multi-kilobyte value.

(2) The HyperModel benchmark includes the six performance tests proposed by Rubenstein, Cattell, and Kubicar [RUBE86], plus some new measures: a traversal measure similar to the EDB *Traversal*, and read/write operations on the "blob" field.

The extensions to the original benchmark provide additional information about a DBMS, and the choice of additions is good. However, we feel that these measures of secondary importance to those included in EDB. The "blob" operations are not bottlenecks in the applications we are familiar with -- blob operations are typically limited by disk speed. The additional relationships do seem to represent a more realistic database, but they appear to 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.

Note that the HyperModel benchmark does not measure versions, distributed databases, operations on compound groups of objects, multi-engineer concurrency, or other substantially new DBMS functionality not already covered by EDB. If the HyperModel benchmark continues to be evolve, such measures will be important additions to a comprehensive benchmark characterizing DBMS functionality and performance for engineering applications.

5.3. Summary of Engineering Database Benchmark Rationale

To summarize, EDB was designed based on earlier benchmark experience and feedback. The HyperModel work provided a list of suggestions as did vendors, referees, and benchmark users. The full list includes "blob" fields, aggregation and generalization relationships, versions, multi-user measures, and the "range" queries that we removed after our first paper. All these features belong in any more comprehensive benchmark; but they reduce the utility of a simple benchmark focussed 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. Running EDB and Reporting Results

It is essential that a good benchmark be precisely reproducible. This section, therefore specifies more detailed issues on the implementation of EDB.

In order to reproduce comparable results, it is necessary to run the benchmarks and DBMS on a similar configuration. Table 1 shows a number of pertinent particulars of the DBMS architecture and benchmark implementation that should be reported.

Table 1: Reportable Benchmark Information

Area:	Types of Reportable Information
Hardware:	CPU type, amount of memory, controller, disk type/size/bandwidth
Software:	OS version, size of cache
DBMS:	Transaction properties (atomic?, degree of concurrency, degree of isolation, lock modes used, etc.); recovery & 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; "wall clock" running time; CPU time and disk utilization time are also useful; size of database. Optionally: local performance.

The benchmarks reported here used a database server machine consisting of a Sun 3/280 with 16 megabytes memory (Motorola 68020, 25 MHz)[2], it had a SMD-4 disk controller (2.4 MB/sec transfer rate) connecting two Hitachi disks, 892 MB each, 15ms avg seek time; one disk used for database, and one for system. The server ran the Sun UNIX O/S version 4.0.3 and was connected to the client via a standard 10 Mb/s Ethernet.

The client workstation, where the remote benchmark programs ran, was a 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 reserved exclusively for benchmarking during the runs. Swap space (virtual memory space) on both machines was generous, and paging activity was not an issue during the benchmark.

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

All the benchmark measures except database-open, 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 and client machines, so that code swap time will not be counted in the actual benchmark measures, it is only counted once in the "open" time.

Disk I/O overhead should be included in the EDB benchmark response time measurements. However, many database systems *can* validly obtain better performance for engineering applications by caching data between the multiple lookups performed within one transaction or session. Repetitions in the measurements are specifically provided to allow reporting asymptotic warm results as well as the cold results on the first run. No data may be cached at the time the benchmark is started for the cold results. The database must be closed and re-opened before each cold 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 between each cold run. This is a subtle point that can make a major difference in the results; one solution is to run 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.

Despite all of these precautions, the cold and warm measures can still be hard to define consistently. If the DBMS is capable of storing data on disk as an image of virtual memory, so that it can run faster when the database is mapped into the same memory space, we would consider this a "cool" start. If the DBMS can run much faster fetching exactly the same objects in each iteration of the benchmark rather than random ones, we would consider these "hot" results. While such numbers may be interesting, for simplicity's sake they are not included here.

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

- Clear the operating system cache
- Open the database
- Start timer
  - Perform cold run & report cold time for the first benchmark loop
    - Include time for transaction checkpoint for "insert" measure
  - Continue execution for remaining 9 loops & report warm times for successive runs
    - If stable asymptote emerges, report it as warm time; otherwise report average of the 9 runs
- Stop timer
- Restore the database to its original state (reverse all inserts).
- Close the database

For the small database, the entire database typically fits in main storage. The large and huge database, are defined so that there is little advantage to a client cache larger than the small space

required to hold root pages for indices, system information, data schema, and similar information since 90% of the data will not fit in cache.

The client workstation is permitted to cache as much of the small database as it can. On the other hand the large database should be at least ten times larger than the client cache and at least three times larger than the server cache.

For results reported here, 8 MB client machines had 5 MB for a cache split between the O/S or DBMS. This amount of memory is typical of what is available for cache use on 1990 workstations. It will no doubt increase as technology advances and manufacturing techniques improve, however, the amount of memory used by applications and system software will also increase, so approximately 5 MB of cache may be realistic for some time to come.

For results reported here, the 16 MB server cache size was limited to 12 MB so that is was less than one third the size of the large database. Memory boards were removed to produce the proper configuration. If the server machine had 64 MB of memory of which the system software left 60 MB free for cache, then the large database should be at least 180 MB (about 800,000 parts).

There are several issues with generation of the benchmark database:

Generating ID's: The small database is populated 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 program to select a random part, which is guaranteed to exist, by calculating a random integer. Indices and other physical structures were chosen to achieve the best performance for each system. 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 OK, 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: Connections may be clustered in physical storage according to the part they connect. They may also be clustered with parts themselves, if this gives better performance. It is easy for the benchmark implementor to do this clustering, because part connections are clustered on part-id: the best way to group parts is sequentially by part-id. In fact, when we designed

the benchmark it seemed that this was *too easy*; that it gave no advantage to a DBMS that could automatically cluster by connections. However, benchmark implementors could build the equivalent ordering from *any* underlying clustering statistics inherent in the benchmark database, and group the parts accordingly. So, while using part-ids for clustering 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 id's 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, the new 100 parts and 300 connections 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 3 connections for each of these parts are selected using the same algorithm as the others, i.e., they are 90% likely to be to one of the parts with largest part-ids, 10% 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 are not 20,200 parts after two executions. However, it was found that 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 no measurable effect on the results was seen when this load was not controlled. This may 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. The EDB benchmarks reported here used the random number generator suggested by Park and Miller [PARK88].

7. Porting EDB to Three DBMSs

The EDB benchmark has now been implemented on a number of DBMSs. This section examines the results on three products that vary quite dramatically in their overall design, remote data access, concurrency control implementation, and traditional DBMS properties. They show some of the performance implications of different architectures.

7.1. OODBMS

The first system is a pre-release ("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, transactions, and remote access. It is called "OODBMS" here. 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.

The EDB benchmark was implemented on the OODBMS with an object type *Part* containing variable-size arrays with the connection information. Object references (and reverse references) are stored by the OODBMS as object identifiers that contain a logical page address in the upper-order bits, which can normally be mapped to the physical page and offset of the referenced object with no extra disk accesses. The particular implementation we chose, did *not* allow OIDs to be automatically swizzled[3] into pointers by this OODBMS the first time an object is used. However, the mapping from OIDs to memory address is very fast when the object has been cached, requiring only table lookups.

The OODBMS's transaction mechanism supports transaction rollback and rollforward. The OODBMS works remotely over a network, fetching pages rather than objects, and caches pages on the workstation. Objects may be grouped in multi-page segments, with locking at a segment level. The benchmark design placed all of the parts and connections in one segment. Page-level locking is being implemented, it will be informative to run the benchmark with these finer-grain locks. It will probably not be substantially slower, because the implementation will use a page server in which lock requests can be "piggy-backed" on the page reads.

7.2. RDBMS

The second system, here 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. Note that the DBMS calls are invoked many times for each measure, since it mixes C and database operations. Queries were precompiled wherever possible, thus bypassing most of the usual query parsing and optimization costs; or at least incurring these costs only on the first execution.

[3] *Swizzeled* is a OODBism which means *converted*, *compiled*, or *bound*.

Two tables were used, Part and Connection, indexed on *part.id* and *connection.from*, respectively. A single secondary index was also created, on the *connection.to* attribute; this permitted reasonable response times on the reverse traversal measure. B-trees were used as the storage structure for all three tables, with 100% fill factors on both index and leaf pages.

The RDBMS provides 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 which we envision.

The RDBMS, like all current relational DBMS products, is implemented using a client-server architecture, i.e. every database call from the application must go over a network to the DBMS server where the data is stored. This is very inefficient for an engineering application or benchmark such as ours -- there is no way to cache data locally as in the OODBMS and INDEX. However, a stored procedure was also defined for each of the three benchmark measures, so the client/server interaction consisted of a single customized call per operation.

We considered implementing a cache ourselves, as a layer between the RDBMS and the application. However, doing so would effectively duplicate much of the functionality of the DBMS, as an "in-memory" DBMS: index lookups by part id, connection links between parts, and so on. In addition, the cache would not work properly with the RDBMS, because the RDBMS transactions do not keep the data stored in the application cache locked. We feel our utilization of the RDBMS representative of what an application implementor would expect of a DBMS.

7.3. INDEX

The third system is a B-tree package we had available at Sun. Throughout the balance of the paper, this system is called "INDEX". INDEX is included to show what can be done under ideal conditions, using a highly optimized access method with no high-level semantics.

The benchmark database was implemented as two files, one for parts and one for connections. A B-tree was created on the part ids in the parts file. This B-tree was used 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, the record ids for parent-child links as in System R (Astrahan et. al. [ASTR76]) were used. The 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 part-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, the *Traversal* measure was also performed using B-trees. Two more B-trees were created, one on the *connections.from* field and one on the *connections.to* field. Because the parent-child link implementation was faster than the B-tree approach, the results from this pure B-tree approach are not reported in the next section. The B-tree-plus-link results are shown in the following section. Note: the overall database creation time, marked with "*" in our tables, reflects the time for both the B-trees and links. The insert times reflect only the link 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 specially utilizing a novel network lock service to control access. The INDEX package caches one 8K contiguous chunk 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. The measurements reported here used table granularity locks. Multi-statement atomic transactions are not supported, nor are native security features. Of course, there is no query optimizer; it is the duty of the applications programmer to design and program the fastest access paths, as just described. A single-process architecture is used (plus a remote lock-and-access server process in the networked case). The INDEX access libraries are linked directly into the benchmark program.

The INDEX system, although extremely powerful within its intended application domain, is not strictly comparable to full-function, transaction-oriented DBMSs. The present implementation does not satisfy our recovery requirement and it performs file-level locks. However, these results are included for comparison because (1) gross-level concurrency and recovery such as this, with the addition of versions, might be adequate for many CASE and CAD applications, and (2) the INDEX results represent the best performance that might reasonably be obtained in the case where the cost of high-level semantics and transaction management can be made negligible.

8. EDB Measurements on Three DBMSs

Table 2 shows the EDB measurements for the most important scenario, 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 data is graphically displayed in Figure 9.

Table 2. Small Remote Database
benchmark elapsed times in seconds

Measure	Cache	INDEX	OODBMS	RDBMS
DB size		3.3MB	3.7MB	4.5MB
Load Time (local)		1100*[4]	267	370
Reverse traverse	cold	23	6.3	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	19.	5.	123

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 best; both come close to the goal of ten seconds for executing each cold measure. Though the results are close, INDEX wins on the cold results. The OODBMS wins by a substantial margin on the warm results.

[4] As explained in the description of the INDEX database design, this time includes the time to create two indices which are not actually used in the B-tree plus link design.

[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, nearly the entire small database was in the cache after the first iteration of the lookup or traversal measures.

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 inter-process 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. In consulting with OODBMS implementors we discovered that the OODBMS pre-release had a poor hash index implementation, now being re-implemented. This is probably the cause of the discrepancy.

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 a "*") included the overhead to redundantly create B-trees 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 worst, with an overall total of 143. The results reflect the overhead of accessing the DBMS for each operation in the benchmark application, approximately 5000 calls altogether. At 20 ms for a roundtrip network call, this overhead amounts for most of the time. As noted earlier, the RDBMS architecture is at least as important as the relational model in producing this poor performance.

Both the OODBMS and INDEX come close to the goal of 1000 simple queries per second (random reads per second). Ideally, all three measurements, *Lookup*, *Traversal*, and *Insert*, should be within the ten second limit for acceptable levels of human interaction. Today, 1000 random operations per second is an upper limit for performance on a database that does not fit in memory, even with a 90% cache hit rate, since the best disks generally require at least 10 ms for a seek. Thus, unless parallel disc servers are used, one cannot hope to perform much better than a ten second range on the cold lookup and traversal benchmarks (1000 lookups, 4000 traversals) with current technology. Fortunately, this satisfies many engineering application needs.

Table 3 shows the EDB results for a large remote database. Now none of the products can take much advantage of a local cache. Again, this data is graphically displayed in Figure 9.

Table 3. Large Remote Database benchmark elapsed times in seconds				
Measure	**Cache**	**INDEX**	**OODBMS**	**RDBMS**
DB size		33.1MB	37MB	44.6MB
Load Time (local)		16400s*	6000s	4500s
Reverse traverse	cold	100	84	212
Lookup	cold warm	47 18	49 43	49 24
Traversal	cold warm	56 41	68 59	135 107
Insert	cold warm	20 18	10 7.5	24 24
Total	cold warm	123 77	127 110	208 155

Note that there is much less difference between the three systems now, 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 no longer can be cached. The access methods seem to scale well with database size. Indeed, the link access method 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 *Traversal*, INDEX and OODBMS do better than the RDBMS. These differences do not surprise us because: (1) The locality of reference inherent in *Traversal* can provides benefits even when the database is not fully cached, and (2) INDEX and OODBMS use links instead of index lookups to traverse connections.

As an experiment, EDB was run for the *huge* remote database. Because the create time to create for the database is so large, only the INDEX package was measured. The results are shown in Table 4.

Table 4. Huge Remote Database benchmark elapsed times in seconds				
Measure	**Cache**	**INDEX**	**OODBMS**	**RDBMS**
DB size		330MB	--	--
Load (local)		78000+	--	--
Reverse traverse	cold	109	--	--
Lookup	cold	109	--	--
	warm	106	--	--
Traversal	cold	143	--	--
	warm	142	--	--
Insert	cold	53	--	--
	warm	53	--	--
Total	cold	305	--	--
	warm	301	--	--

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

With a 0% 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 this is very close the the results we actually obtained for the huge database. This suggests the results might not get much worse with a "very huge" database.

9. Variations

This section considers some variations of the benchmark configuration, beyond the results required by the benchmark definition, to better understand the differences among the DBMSs.

Table 5 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 many will require remote data access.

Table 5. Cold Local Database benchmark elapsed times in seconds				
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

These results surprised us. We expected the results in the local case to be much better, 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 DBMS process, and copying data between the two. There is significant overhead in such an interprocess call. In addition, the benchmark application is competing for resources on the server machine in the local case.

It might seem that the INDEX and OODBMS results should be much better than for the remote database, but they are only 30-40% faster. Many people are surprised to hear that remote access to files is not much more expensive than local access. To quantify this, some rough measurements were performed running an application reading pages locally and remotely through NFS (Sun's Network File Server). The measurements showed remote access to be only a 30% slower. Of course, this rough measurement does not include time for locking, or to flush altered pages back to the server. 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.

As another experiment, consider the effect of the locality of reference that exists in the connections among parts. Table 6 shows these results.

Table 6. Small Remote Database Without Locality of Reference (elapsed times in seconds)		
Measure	**Cache**	**INDEX**
Lookup	cold	7.9
	warm	2.4
Traversal	cold	24
	warm	7.9
Insert	cold	36
	warm	10
Total	cold	68
	warm	20

Note that the results for the cold *Traversal* (24 seconds in Table 6 vs 17 seconds in Table 2) and insert (36 seconds vs 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 used for *Traversal*. The performance differences caused by locality show that the addition of reference locality to EDB is important.

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, one would expect there to be similar or larger differences without locality of reference, though these cases were not tested. 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 structures such as B-trees take longer to traverse. Only the former is substantially affected by locality.

We were curious about the effect of using B-trees instead of parent-child links to perform the traversal measure. Intuitively, one would 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). Table 7 shows the measurements of EDB when B-Trees are created on the *connection.from* and *connection.to* fields, and used for the traversal.

Table 7. Small Remote Database (3.9 MB) **Using B-Trees Instead of Links** (elapsed times in seconds)		
Measure	**Cache**	**INDEX**
Lookup	cold warm	7.9 2.4
Traversal	cold warm	33 21
Insert	cold warm	12 9.1
Total	cold warm	53 32

As expected, (compared to Table 2), *Traversal* is much 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 effected.

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 represented by EDB. With a larger database the difference will 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. Extensible hash indexes might do well.

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. For the OODBMS, and parent-child link version of INDEX, the connections are traversed in exactly the same order they were created. If an application requires an ordering on the connections, for example if the parts were portions of a document with a hierarchy of chapters, sections, and paragraphs, then the RDBMS versus OODBMS performance difference would be even greater, as the RDBMS would have to sort the connections during each query. It would be interesting, for further work, to measure this "ordered" version of the traversal.

10. Summary

The Engineering Database Benchmark was originated to focus more emphasis on performance for engineering database applications. Its design is based on benchmarking experience and on the

study of engineering applications. It consists of a scaleable database design with clustering, and of three simple operations representative of such applications (*Lookup, Traversal, and Insert*). The operations are performed by a client accessing a remote database. The client and server are tested in two cases: a small database which can be cached, and a large database which cannot be cached.

EDB differs from other benchmarks in several important ways: (1) the database is remotely located, (2) the application logic 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 violations of conventional wisdom, to obtain acceptable performance.

It should be emphasized that the results do *not* show that an object-oriented data model is better than a relational data model for engineering applications, although there may be some differences in data model performance for engineering applications as a results of expressability (e.g. if lists must be represented by sorting relations on a cardinal attribute). The order of magnitude difference in the OODBMS and RDBMS results appear to result from a difference in the DBMS *architecture*.

All of the following changes to a conventional relational implementation would result in significant speedup on the EDB benchmark:

(1) It is essential to integrate the DBMS with a persistent programming language, or otherwise reduce the overhead in accessing data from the application environment. Portals may afford some improvement **[?? ref to portals Rowe??]**, but they still require data to be translated between the DBMS representation and programming language representation on each database call.

(2) Caching data on a workstation is important, because of the locality of reference present in the benchmark. The concurrency control architecture of the DBMS must support long-term checkout in order for this caching to work.

(3) Remote access, local caching, and concurrency control should be performed at a granularity of pages or through meta-queries that prefetch data that will be operated on by a series of application queries or operations.

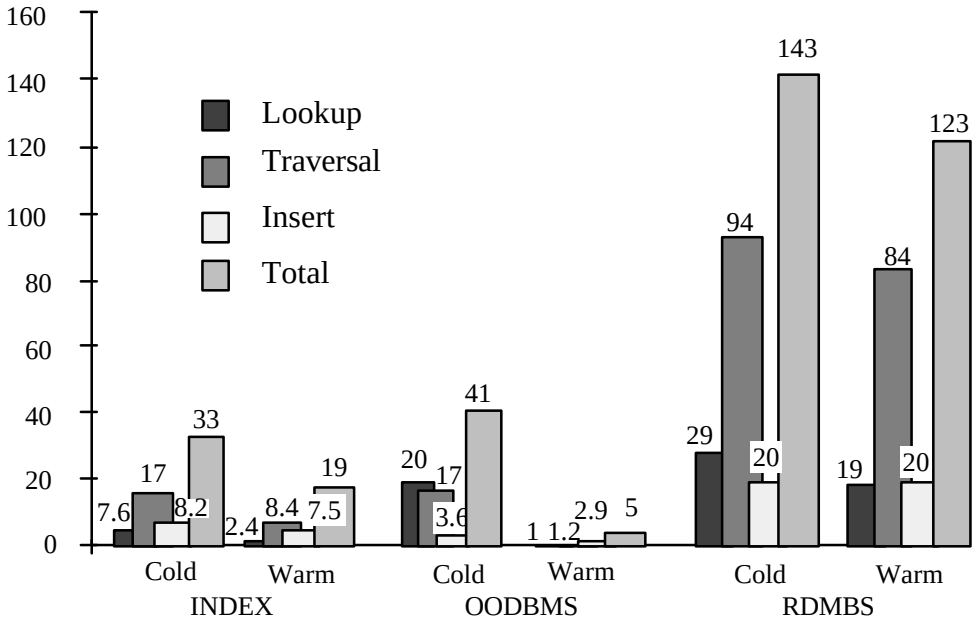
(4) Parent-child links are useful for efficient traversal operations.

In theory a relational DBMS could be built with these features, however it would be a major re-implementation effort. Very little work has been done in this direction.

It should also be emphasized that this benchmark is intentionally simple to make it easy to port and perform. It is only a rough approximation of the requirements of engineering applications, based on experience and interviews with CASE and CAD engineers. The only truly accurate representation of performance is actually executing the application. Our goal is mainly to focus attention on gross-level issues -- as shown in Section 9, there is an order of magnitude difference in performance among DBMS architectures.

To summarize the benchmark results, a pre-release of a commercial OODBMS performed well, as did a home-grown INDEX access method. These systems demonstrate that our goals for engineering DBMS performance are not impossibly high. A commercial RDBMS did poorly in comparison to these systems, but we believe a goodly portion of this difference is due to the architecture, not the data model, of the RDBMS. Figure 9 shows the results graphically (based on Table 2).

Figure 9. Small remote database results with different DBMSs.



The benchmark was also executed under various configurations with an in-house system here called INDEX. Since INDEX had good overall results, and we were familiar with its internal architecture, it was chosen for these comparisons. The results are shown graphically in Figures 10 and 11.

Figure 10. INDEX results for remote access to small, large and huge databases.

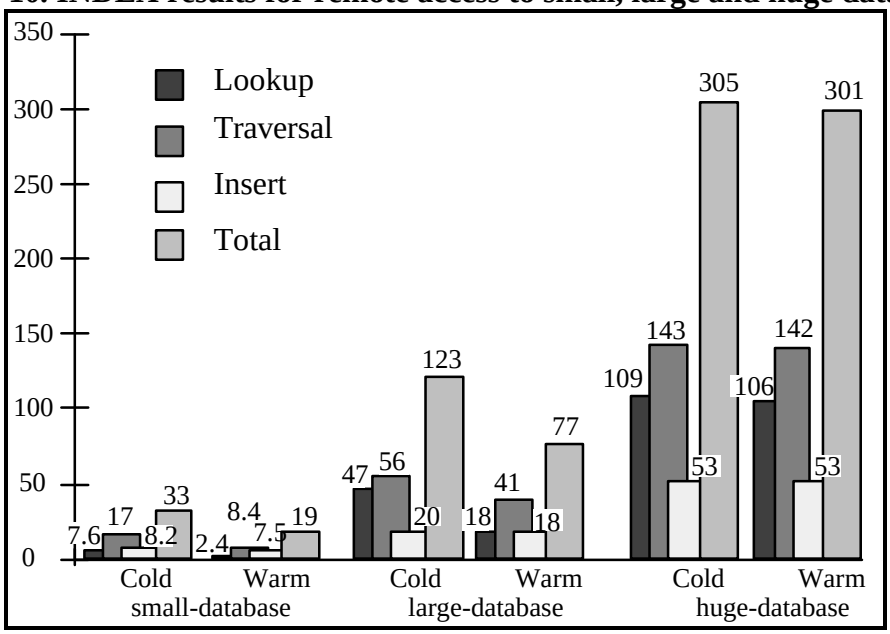
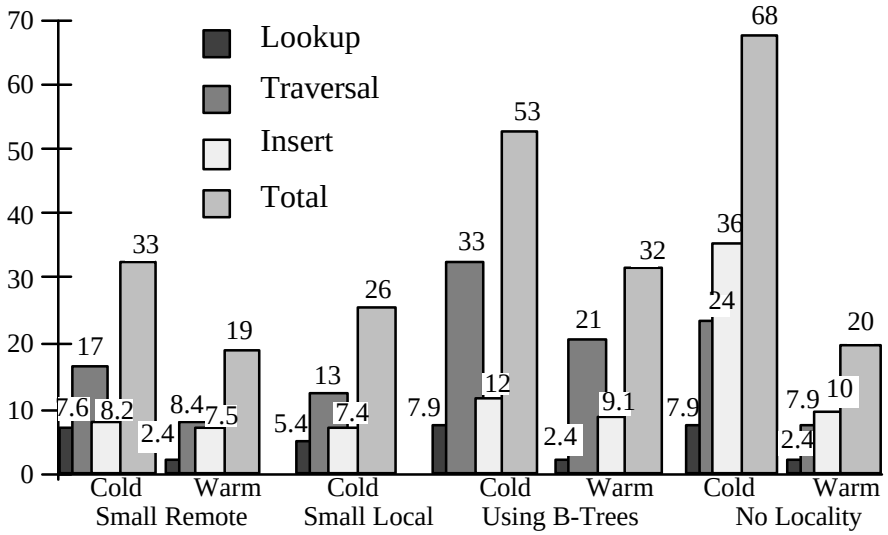


Figure 10 shows that the elapsed times grow with database size: each time the database grew by a factor of ten, the elapsed times rose by a factor of three or four. In addition, once the database grew past the cache size, there was little difference between the cold and warm results. This is especially clear for the huge database measurements in Figure 10. It is a consequence of less locality of reference and because the number of disk accesses for index lookups and other operations increase.

As shown in Figure 11, when the database is local the results are better for INDEX; this was also true with the OODBMS. 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 performed poorly even in the local case.

Figure 11. INDEX results for the small database accessed locally, remotely, or via B-trees instead of links for connections, and without locality of reference in connections.



Without locality of reference, the results on the traversal measure are somewhat worse. This shows that it is important to include locality of reference in a benchmark measure, assuming the applications the benchmark is to represent exhibits such locality.

Finally, as shown by Figure 11, using B-trees instead of parent-child links for the benchmark produces significantly worse results than for traversal, with only minor effects on the other measures. This suggests that it is important to consider alternatives to B-trees for logical connections that are traversed in a "transitive-closure" type operation.

To summarize, the primary purpose of the EDB benchmark was to bring attention to the performance needs of engineering applications, by carefully defining a set of measures that can be used compare DBMSs. However, we have also learned a lot about DBMS performance for these applications through using the benchmark with different systems and configurations.

Database systems will not be used in "hard-core" engineering applications unless they perform close to 1000 random operations in one second with a 90% cache hit rate on commonly used workstations and servers. We have shown that such results are possible. We also have seen that products can fall short of these results, by as much as an order of magnitude. Good engineering database performance require effective use and integration of the programming language data space, concurrency control allowing long-term check-out, and caching on a local workstation.

11. References

Astrahan, M., et. al., System R: "Relational Approach to Database Management," *ACM Transaction on Database Systems,* 1, 1, 1976.

Anderson, T., Berre, A., Moira, M., Porter, H., Schneider, B., The Tektronix HyperModel Benchmark, Tektronix technical report, August, 1989.

Anon et. al.: "A Measure of Transaction Processing Power," *Datamation*, Vol. 31, No. 7, April 1, 1985.

Bitton, D., DeWitt, D. J., Turbyfill, C., "Benchmarking Database Systems: A Systematic Approach," *Proceedings VLDB Conference*, October, 1983. [Expanded and revised version available as Wisconsin Computer Science TR #526.]

Cattell, R. G. G., Skeen, J., "Engineering Database Benchmark," to appear, ACM Transactions on Databases, 1991.

Maier, David: Making Database Systems Fast Enough for CAD Applications, technical report CS/E-87-016, Oregon Graduate Center, Beaverton, Oregon, 1987.

Park, S., Miller, K.: "Random Number Generators: Good Ones are Hard to Find," CACM 31, 10, October 1988.

Rubenstein, W. B., Kubicar, M. S., Cattell, R. G. G.: "Benchmarks for Database Response Time," *Proceedings ACM SIGMOD 1987*.

Transaction Processing Performance Council (TPC), "TPC Benchmark A Standard", November, 1989. Contact Omri Serlin, ITOM International Co., PO Box 1450, Los Altos, California.

Appendix: SQL Definitions

??I translated the Sybase ddl and db-lib calls to ANSI sql. Jim ???

Database Definition

```
      create table part ( id        int         not null,
                          part_type   char(10)    not null,
                          x           int         not null,
                          y           int         not null,
                          build       datetime    not null,
                          primary key (id) )

      create table connection (
                          from        int         not null,
                          to          int         not null,
                          length      int         not null,
                          conn_type   char(10)    not null,
                          primary key (from,to,conn_type) )

      create index rconn on connection(too);
```

***Example of C Code:* Lookup**

```
     /** obvious sql and c declares missing **/
    lp_start  = time100();
    for (i = 0; i < nrepeats; i++)
      {  starttime = time100();
         for (j = 0; j < nrepeats; j++)
            {  p_id = ((int) randf()* nparts) + 1;
            exec sql    select x, y,part_type
                           into :x, :y, :part_type
                           from part
                           where id = :p_id;
            nullProc(x,y,type);
            }
         endtime = time100();
         printf (
         "\tTotal running time for loop %d is %10.2f seconds\n",
                                    i+1,  (endtime - starttime) / 100);
         }
    endtime = time100();
    printf ("\n\nTotal running time %10.2f seconds\n\n",
                                    (endtime - lp_start) / 100);
```

?? This is my attempt insert. I think it would be good to do insert and traverse in standard SQL ??? (yes it will be ugly..

Example of C Code: ***Insert***

```
    lp_start  = time100();
     p_id = nparts;
    for (i = 0; i < nrepeats; i++)
      {  starttime = time100();
         for (j = 0; j < nrepeats; j++)
            {  p_id = ((int) randf()* nparts) + 1;
            exec sql    insert into part
                           values (:p_id,
                                   : part_type,
                                   :x, :y,
                                   CURRENT);
               connect (p_id);  /* code as per section. 3. */
            }
         endtime = time100();
         printf (
         "\tTotal running time for loop %d is %10.2f seconds\n",
                                   i+1,  (endtime - starttime) / 100);
         }
    endtime = time100();
    printf ("\n\nTotal running time %10.2f seconds\n\n",
                                   (endtime - lp_start) / 100);
```