Benchmarking Simple Database Operations

W B Rubenstein, M S Kubicar, R G G Cattell

Sun Microsystems, Incorporated
Mountain View, California

Abstract

There are two widely-known benchmarks for database management systems the TP1 benchmarks (Anon *et al* [1985]), designed to measure transaction throughput, and the Wisconsin benchmarks (Bitton, Dewitt, & Turbyfil [1984]), designed to measure the performance of a relational query processor In our work with databases on engineering workstations, we found neither of these benchmarks a suitable measure for our applications' needs Instead, our requirements are for *response time* for simple queries We propose benchmark measurements to measure response time, specifically designed for the simple, object-oriented queries that engineering database applications perform We report results from running this benchmark against some database systems we use ourselves, and provide enough detail for others to reproduce the benchmark measurements on other relational, object-oriented, or specialized database systems We discuss a number of factors that make an order of magnitude improvement in benchmark performance caching the entire database in main memory, avoiding query optimization overhead, using physical links for prejoins, and using an alternative to the generally-accepted database "server" architecture on distributed networks

1 Overview

We are interested in database response time the time that elapses from the issuance of a database query until the results are returned In most of our applications, these queries are issued by a program rather than a user, and the programs must issue many simple queries in order to update a window or graphical display, since the intermediate queries often cannot be expressed as a single high-level relational query For these reasons, response times on the order of a small number of milliseconds are required from the database system

We will present a set of benchmarks to measure response time performance from a database system, for simple queries These benchmarks measure, for example, the time to look up an object given its name, or to find connected objects (say, sub-objects), once an object is found We present some results we obtained running these benchmarks on database systems on a Sun workstation

Our benchmarks are not limited to relational database systems, indeed better numbers on many of the benchmark measurements may be obtained from "object-oriented" database systems (Dittrich and Dayal [1986]), network or hierachical systems, ISAM packages, or custom application-specific database systems The benchmark measurements are oriented towards simple operations on individual objects or records in a database system, rather than relational queries involving complex operations over many tables However, engineering applications may also require the more complex operations that relational database systems provide, so performance on our benchmarks alone do not make a system acceptable We would like the simple-query response time performance *and* high-level relational facilities

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 A relational system typically responds 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 a number of factors, only one of which is the disk seeks required by a database system. We believe there is a place for a database system with 10 to 100 times the response time performance of a conventional relational database system, to fill the gap between such systems and specialized or in-memory data structures The thrust of our benchmarks is to identify database systems that fill this gap

Sun Workstation® is a registered trademark of Sun Microsystems, Inc
UNIX™ is a trademark of AT&T Bell Laboratories
UNIFY® is a registered trademark of Unify Corporation
INGRES™ is a trademark of Relational Technology Inc
SQL™ is a trademark of International Business Machines Corporation

2. Response Time

We define *response time* as 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 use a program rather than an end-user for our definition because that is our primary client, and because the response times we need are too short for an end-user to measure We use real time rather than CPU time because that is the measurement critical to the actual applications, the benchmark measurements will therefore differ with CPU and disk properties, as well as with the database system used

Our interest in response time stems from engineering database applications requiring fast, simple database operations These applications include CAD (Computer Aided Design), CAM (Computer Aided Manufacturing), CASE (Computer Aided Software Engineering), and a variety of real-time or network service programs that perform relatively simple queries but require a response in a small number of milliseconds For example, imagine an application drawing a computer circuit layout or building architecture on a graphical display, in which information about the individual components and their interconnectivity is stored in a database These programs may execute thousands of simple queries to complete the drawing Such applications often require response times better than 10 milliseconds in order to provide reasonable response to the user

In addition to supporting specialized engineering database applications, we have constructed generic database tools utilizing a mouse and bitmap graphics to allow a user to browse through a database, or to see a database graphically (Sun [1986a]) These tools, like the engineering applications, require fast response times to provide a reasonable interface to the user

All of these applications and tools require fast response for *simple* operations, not the range of complexity in queries measured by the Wisconsin benchmarks In relational terms, most of our queries are on a single table, or are operations on a single record or small group of records representing one logical object While there is some overlap between our benchmarks and the Wisconsin ones, both benchmarks include measurements missing in the other, as a result of the different emphasis

The TP1 benchmarks involve simple operations, like ours, but are restricted to particular sequences of simple operations, and measure *throughput* rather than response time Additional throughput can be achieved by multiprocessing and pipelining Improved response time cannot Again, there is some overlap between TP1 and our measures Since TP1 includes a composite of our measures, good performance on TP1 generally correlates with all of our measurements, but it does not differentiate among specific strengths

3 Current Solutions

A variety of approaches have been taken to fill the need for fast response time from database systems

(1) Optimizing the performance of relational database systems, by compiling queries, caching information, and otherwise improving the response as viewed by the user

(2) Building database systems that are oriented towards operations on individual objects There is recent popularity for "object-oriented" database systems that provide this orientation, although many earlier network-model and hierarchical-model systems had this capability

(3) Coding "special-purpose" database systems for particular applications, such as a CAD product These special purpose systems achieve performance by storing a database in special-purpose data structures, typically in virtual memory

There are difficulties with each of these approaches

It may not be feasible to achieve adequate performance from a strictly relational database system Typically, the database lookups our engineering applications and tools perform cannot be expressed as a single query in high-level relational query languages, so must be decomposed into many queries Also, very few database systems provide a compilation mechanism that removes *all* of the query parsing, optimization, and validation from the runtime overhead, this overhead can be considerable for simple queries Finally, relational systems typically are not constructed to take advantage of a large main memory, to achieve better performance

We might get the performance we require from "object-oriented" database systems as they become available, but typically we then lose the powerful facilities that come along with the relational system Similarly, some network or hierarchical systems provide reasonable response time, but lack the simplicity and power of the relational model For many engineering applications, we require a language for ad hoc queries, report generation, and forms-based interfaces, as well as a higher-performance object-level interface

Many shops, for example for CAD products, have constructed specialized database systems for their applications, often starting with data structures in a programming language This approach lacks the generality of a full DBMS, not to mention duplicating the development effort for database facilities Typically a specialized system doesn't include good concurrency control mechanisms, a complete set of access methods, or a language for ad hoc queries

A new, fourth alternative is one which *combines* the features of a relational database system with an object-oriented one Such a database system would have two independent programmer's interfaces to the same database An early example in this direction is System R's RSS level, providing programmers an alternative to the high-level SQL language (Astrahan *et al* [1976]), although the RSS level provides only the kernel for an "object-oriented" interface, and does not provide physical data independence

It is more difficult to construct a system with two compatible interfaces at different levels We have experimented with such an approach, however, and will show some preliminary results in a later section

4 Benchmark Database

We want a set of benchmarks independent of the data model provided by a particular database system We achieve this, as much as possible, with the following definitions

A *record* is a set of fields We define a record as an object in an object-oriented system, and as a tuple in a relational system If the database system allows a variable quantity of data to be associated with the fields of a record, not unlike associating a group of records together to represent an object, then this will be reflected in the benchmark performance even though their definition of "object" is not exactly a record

A *record type* is a class in an object-oriented system, or a relation in a relational system It is a group of records with the same field types

A *key* is a field that must be unique over all records of a type This might be a relational primary key or a unique object identifier

We assume that fields of records may have a variety of scalar types, such as integers or strings Some database systems allow fields with list-valued types, and some allow fields with references or lists of references to other objects (records) In a relational system, reference fields are called *foreign keys*, they are fields whose values are the keys of another (referenced) relation List-valued fields are not permitted in first normal relational form, but are represented by putting the values in another relation that references the records in this one

Our benchmark database will consist of three record types

(1) A *person* record type, with three fields a person ID number, a name and birthdate The ID is a 4 byte integer, and is the key for the record type (i e each ID is unique)

The name may contain up to 40 bytes, and the birthdate is a 4 byte integer We don't assume the presence of any special "date" data type There are 20,000 person records in our database, with randomly distributed names and birthdates The ID fields are generated as ascending integers (from 1 to 20,000)

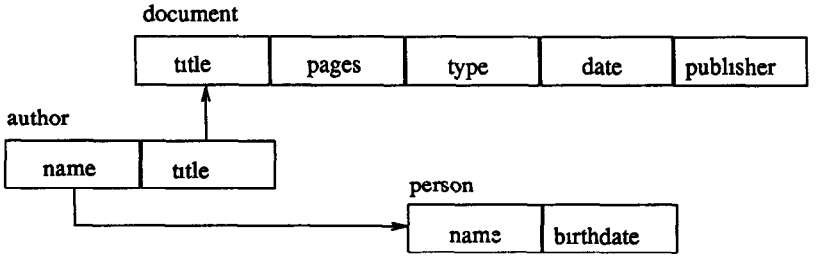
(2) A *document* record type, with six fields a document ID, a title, a page count, a document type, a publication date, a publisher, and a description The document ID, a 4 byte integer, is the key for this record type The title, publisher, and description fields are strings containing up to 80 bytes each The page count, document type and publication date are integers There are 5000 documents in our database, with randomly generated titles, page counts, types, publication dates, publishers, and descriptions The ID fields are generated as ascending integers (from 1 to 5000)

(3) An *author* record type, with two fields a person ID, which references the key of the person table, and a document ID, which references the key of the document table This table therefore connects each person to zero or more documents, and each document to zero or more persons There are 15,000 author records in our database Each document record is associated with three randomly selected person records

We assume that the physical layout of the database is not specialized to our benchmark in any way In particular, the author records are not co-located or clustered with either the document or person records that they reference, unless that would necessarily happen because of a redundant storage mechanism used by the database system We add this constraint because the person and document records in an actual database may be connected to many other objects in the database, and cannot be clustered with more than one of them

Our database comprises approximately 3 megabytes of data (ignoring all overhead introduced by the data manager) As such, it is a good representative of an engineering database that might fit entirely within a large main memory, for example the data associated with an engineering 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. However, our database system must also be able to scale up to larger databases, exploiting access structures that use secondary memory efficiently

We therefore include in the benchmark a second database, which we will call the "large" database, that is identical to the smaller one except that all of the record counts are scaled up by a factor of 10 This database then requires approximately 30 megabytes of storage, plus overhead Typical relational database systems may not have 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 However, it is important to include the large database to eliminate database systems that cannot be scaled up at all We would also like to see numbers reported on a third, "larger" database, scaled up by a factor of 100, however many sites do not have adequate free disk storage to measure such databases



5. Proposed Benchmarks

We propose the following benchmarks for response-time measurements for simple operations

(1) *Name Lookup* This is the simplest database operation, to look for a record with a particular key value This would correspond to looking up the record for a particular part, given its part number A key lookup might be implemented in a database system by a hash or ISAM index

Implementation Fetch the name of the person with a particular randomly-generated ID

(2) *Range Lookup* Finding the records with a particular range of values in a particular set of fields Examples would be finding people with birthdays in a particular range of dates, or finding the documents whose title begins with a particular letter Range lookups are most efficiently implemented by B-Trees and their derivatives

Implementation Fetch the names of people with birthdates in a particular randomly-generated 10-day period within the range we specify for birthdates When the database was loaded, the birthdates were randomly distributed among people in such a way that an average of 10 person records fall within the specified range

(3) *Group Lookup* This is a common operation on databases containing more than one table, to find all of the records in one table that pertain to a particular logical entity in another table Examples would be finding all the author records for a particular document, or all the sub-parts of a particular part In most relational database systems, the most efficient way to implement a group lookup (a join with an entity record) is the same as for the name or range lookups, depending on whether the database implements B-trees and hash indices In other database systems, physical link structures may make this logical operation an independently interesting data point

Implementation Given a random document ID, fetch the author ID's for that document.

(4) *Reference Lookup* Another common operation is the reverse of group lookup, namely finding a record that is referenced by a field of a particular record Examples would be following a chain of connections down a hierarchy of departments and sub-departments, or finding a document record given the author record Again, in relational systems lacking links, this measurement may be identical to the name lookup measurement

Implementation Fetch the name and birthdate of a person referenced by a randomly selected author record

(5) *Record Insert* In some applications, the time to insert new data is critical In general, relational database systems are not very fast for update, and such an application may have to fall back on a hand-written database system, specialized to the application We assume that time to delete a record is similar to insertion time, if not, deletion time should be separately noted

Implementation Store a new author record, including time to update any physical data structures necessary to support efficient retrievals in the above benchmark measurements

(6) *Sequential Scan* This is the operation a database system must fall back on when all other routes fail It is interesting mainly for completeness, because applications requiring high performance must typically define indices making sequential search unnecessary

Implementation Serially fetch records from the document table, fetching the title from each, but not performing any actual pattern match computation on the title

(7) *Database Open* The time to initialize the database system on a particular database This overhead is normally incurred once, when a database application is started The overhead in a database open typically involves opening files and setting up processes to perform database operations

Implementation Perform any operations necessary to open files, database schema information, and other data structures and overhead to execute the above benchmarks, but not time to load the application program itself We assume any database call library is normally loaded with the program

A number of issues arise in measuring these seven benchmark numbers, such as how the initial database is to be generated, how the database system is to initialized when the benchmark is performed, and how "typical" times are to be computed We discuss these issues in the next two sections, providing a more detailed specification Then, in the following section, we summarize the results we obtained running the benchmarks on some of our own database systems

6. Creating the Benchmark Database

Certain issues arise in the creation of the benchmark database which we clarify here

Generating ID's When the person table is loaded, successive records are given successive ID numbers This allows the benchmark to select a random person record, which is guaranteed to exist, by calculating a random integer Other schemes, such as prefetching records from the database in order to determine existing key values, are either overly complex to program, or may skew the timing results by caching the records of interest to the benchmark

Assigning Birthdates In assigning birthdate values, we want to assure that the number of person records selected in benchmark (2) remains constant (at ten records) as the size of the database varies We accomplish this by varying the total range of birthdates from which individual birthdates are randomly assigned For example, in the database with 20,000 people, to maintain an average of 10 people per 10 day range, we select integers from the range 1 to 20,000 These are easily mapped into a day, month, and year, if required by a particular database system, however, this was not done in our benchmarks

7 Notes on Benchmarks

In all of our benchmarks except opening the database, we are assuming that the database system is initialized, and any schema or system information is already cached The initialization time is measured solely by the "database open" measurement

We want to include any disk I/O overhead in the response time measurements for our benchmarks For example, we do not want to average the time to look up the same record many times! On the other hand, many database systems *can* validly obtain better performance for our benchmark measurements by caching data between the multiple lookups performed within one transaction or session So we allow specifically for caching and repetition, as follows

For each benchmark measurement except database open, we perform the specified operation 50 times (on different records each time), and average the resulting times for the number reported It is not permitted to have specifically fetched the same records before, but schema or data may have been cached by earlier operations or by opening the database We repeat the entire set of 6 benchmark measurements 10 times, to simulate an entire session with an engineering application with a mix of different database operations The numbers reported thus average a total of 500 iterations of each benchmark measurement

As an example, for the group lookup, we choose a random title, and fetch the names of its three authors This is repeated with a total of 500 different titles, fetching the names of 1500 authors

We allow the database system to cache as much of the database as it permits in main storage For our small database, the entire database fits in main storage, so that we can measure a database system that exploits in-memory databases For our large database, we do not believe there is a significant advantage to a cache larger than the small space required to hold root pages for indices, system information, data schema, and similar information, since there is little locality of reference in our benchmarks

In all of the benchmark measurements except (5) and (7), we include the time to copy at least one field out of the record retrieved by the operation, because the application program will normally wish to perform some operation on the data retrieved We found that the time to move data out of the record into the user's program is negligible compared to the time for the database system to fetch the record, except in measurement (6), and in that case it is clear that at least one field must be fetched anyway in order to make the sequential scan useful So we feel it is justified, for the sake of simplifying the benchmarks, to lump the copy and fetch times together

We assume that anyone performing our benchmark measurements on a database system will choose indices and other physical structures to achieve the best performance, report what access methods were used, and report on the total space required for the database with the physical structure overhead The physical organization may not be changed between benchmark measurements, of course

8 Our Measurements

We performed the benchmark measurements on Sun Microsystem's two database products, UNIFY (Sun [1986b]) and INGRES (Sun [1987]) We also performed the benchmarks on a specially-modified version of UNIFY, which we will call RAD-UNIFY, that caches as much of the database as possible in memory, and has a simplified locking scheme that allows only one database writer at a time (these restrictions fit many of our engineering applications well)

We used the fastest method available to perform the operations In INGRES, this was the QUEL language embedded in a C program, called EQUEL, in UNIFY, we used a Sun variant of UNIFY's programmer's interface, called ERIC (Extended Record-level Interface Convention), that provides the access method kernel of a lower-level object-oriented interface [1]

The person and document records have hash indexes on their keys The author table has hash indexes on the reference fields in INGRES, in order to avoid a sequential scan to do the group lookups

For name lookup, we used a hash index with both systems Hash index performance was roughly comparable to ISAM performance, for INGRES

For a range lookup, we used a B-tree index with both systems

In the group lookup benchmark, we retrieved 3 author records for a document record, after the document record had already been fetched In INGRES, we used a hash lookup to find these records In UNIFY, we used the physical links ("explicit relationships") that it allows between records These links are also exploited for the UNIFY reference lookup benchmark

In the insertion benchmark, we used the ERIC interface with UNIFY, for consistency with the other measurements, althought much better performance can be obtained for "bulk loads" by using the SQL interface (fewer lock calls are generated, and SQL is streamlined for buffered updates)

Our real-time clock granularity in hardware is only 20ms, but all of our measurements were large enough (for 50 operations) that this was not an issue in getting accurate benchmark times

9. Some Results

Our results for INGRES, UNIFY, and RAD-UNIFY are as follows The benchmarks were run on a Sun-3/160 processor with 8 megabytes of main memory and a local database stored on a disk

---

[1] The ERIC definition in the current product (see Sun [1986a]) provides only access-method procedures, we are experimentally replacing the ERIC interface to include a thin layer that provides an object-oriented "single table query" mechanism (Learmont and Cattell [1987]), as previously done by Cattell [1983]

DBMS	INGRES	INGRES	UNIFY	UNIFY	RAD-UNIFY	RAD-UNIFY
Database	Small	Large	Small	Large	Small	Large
Name Lookup	35ms	45ms	60ms	90ms	9ms	87ms
Range Lookup	393ms*	471ms*	358ms	628ms	76ms	617ms
Group Lookup	116ms	156ms	85ms	108ms*	24ms	28ms*
Reference Lookup	165ms	227ms	50ms	78ms	6ms	70ms
Record Insert	56ms	73ms	230ms	251ms*	43ms	71ms*
Sequential Scan	2ms	2ms	11ms	9ms	3ms	5ms
Database Open	1300ms	1800ms	580ms	580ms	580ms	580ms
Actual Size	6 3MB	60MB	3 8MB	43MB	3 8MB	43MB

Notes Range lookup time is for the entire set of ten record retrievals Group lookup time is for the entire set of three record retrievals Results are derived from INGRES 5 0 and UNIFY 3 2 We allowed a 4MB in-memory cache for RAD-UNIFY All benchmarks were performed on local databases INGRES and UNIFY perform worse, relatively speaking, for remote access over network, see Section 11 Numbers marked with "*" are estimates We could not correctly make these measures because of bugs we discovered in INGRES and UNIFY indexing mechanisms, and made calculations so as to reflect correct operation Exact measurements will be available when the bugs are fixed

It should be noted that the numbers in the table are averages RAD-UNIFY performance improved for approximately the first 100 iterations of the operations, as pages of the database were cached in memory, after that, times for the first 5 measurements dropped to approximately half the times shown in the table INGRES and UNIFY reached asymptotic performance in a few iterations, with little improvement on the times shown

10 General Comments

There are a number of inferences we have already drawn about database system architecture for fast response time, from our experience with these benchmarks and database systems Most database systems have *not* been optimized for the kinds of performance we require, indeed they are at least an order of magnitude away from our requirements, and the trend is often towards *worse* response time in the interest of improving other factors

Impressive performance is obtained by caching much of a database in memory and by assuming little or no contention for database writes, as in RAD-UNIFY Note that large portions of the database may effectively reside in one application's virtual memory and/or be locked out from other users for long periods of time (in our benchmarks, the entire database was cached after about 100 iterations of the benchmark measurements) This architecture allows only one database writer at a time, and readers may not cache data in the presence of writers unless they accept "old" data However, there are still a wide range of uses for a database system such as RAD-UNIFY, since its "single-user" mode can be enabled dynamically

We performed earlier versions our benchmark measurements on previous releases of INGRES and UNIFY from the outside vendors INGRES has shown a substantial performance improvement, up to a factor of 4 on some of our measurements, in the 5 0 release from Relational Technology, making it quite competitive with UNIFY

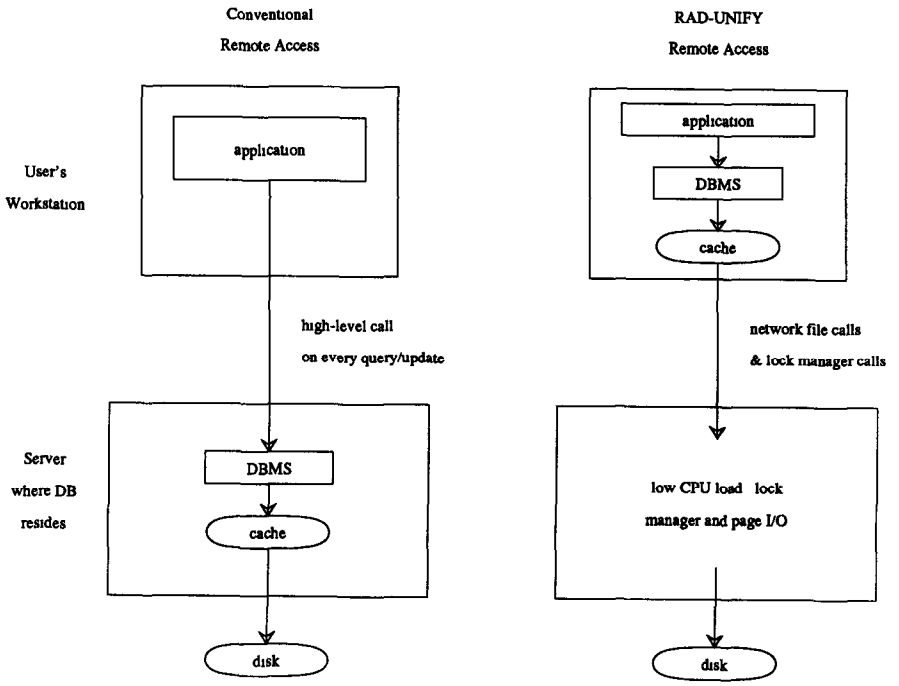
In fact, the performance-optimized INGRES 5 0 now dominates UNIFY on all but the group lookup and reference lookup benchmarks We believe that UNIFY doesn't do as badly on these because it has physical links that can be used for the logical references from the author records to the person and document records The links are similar to the parent-child connections in System R, are bi-directional (can be traversed in the reverse direction), and are invisible to the user of UNIFY's query language, SQL (they are automatically updated, and can even be reconstructed from the foreign keys if destroyed) If INGRES had physical links, it might perform a factor of two or three better, judging from its current advantages across the board The name lookup, group lookup, and reference lookup benchmarks are all simply hash lookups in the INGRES benchmarks, the three measurements are only independently interesting when links are implemented by the DBMS

We performed some of our benchmark measurements on INGRES without using "repeat" queries (repeat queries compile queries after their first execution) As expected, we find that compilation of queries provides substantially better performance as much as 5 times improvement for simple queries In using UNIFY, we effectively compiled queries ourselves by using the ERIC access method procedures, this is easy for simple queries, but obviously unreasonable for more complex programs

11 Remote Databases

All three database systems we tested allowed a database to be located remotely on a network of workstations, i e the database front-end and back-end may be on different machines Generally, remote access cost approximately 30ms extra per record transferred (and per operation invoked) in UNIFY, and somewhat more for INGRES

The desire to access remote databases with fast response time effects the architecture of a distributed database management system The performance exhibited by INGRES and UNIFY could not be obtained using a UNIX pipe to send data over a network, we would require remote procedure calls The performance exhibited by RAD-UNIFY could not even be obtained by remote procedure calls, the entire database system must reside on the front end processor, as must the data cache Only a lock manager and the page-level file system resides at the shared site The architecture for most distributed database systems only support a front end / back end split at the "query" level, so cannot give this performance



12 Conclusions

In summary, most of our interest for engineering database applications is in response time performance Few existing systems do well for our applications, so we designed a set of benchmarks to measure response time We performed these benchmark measurements on our own database systems, and found wide ranges of performance

Space does not permit a good discussion of database system architecture factors effecting the benchmark results in this paper, but we believe the following points are most important

(1) Not surprisingly, a factor of 5-10 in speed can be obtained either by compiling relational queries or by having the programmer access the database at a single-table level, so that optimization is trivial Parsing and optimizing an SQL query may take most of a second, even on a Sun-3 processor

(2) Caching data in main memory can produce almost another factor of 5-10 in speed However, efficient use of main memory caching demands constraints on the remote architecture and also on the number of levels of software between application calls and the data

(3) In a conventional distributed database architecture, network overhead would decrease our best performance numbers by a factor of 5-10 We require an architecture such as RAD-UNIFY's, in which the entire database system resides on the workstation except for disk access (through a network file system) and the concurrency control (remote procedure calls to a lock manager) This suggests that a transaction-based network file system would be very valuable (such as Alpine, built by Brown, Kolling, & Taft [1985])

(4) Specific additional optimizations may together give another factor of 5-10 in speed Improvements result from the maintenance of physical links to speed the group and reference lookups, bypassing file system overhead (considerable at this level of performance), and optimizing the structure of the database system to minimize the number of levels of software involved in the processing of the query This problem must be faced by architectures building object-oriented operations on *top* of a relational system, as in POSTGRES (Stonebraker [1986])

We hope to perform our benchmarks on other database systems, to further substantiate or refute our claims about database architecture We expect that other relational database systems using SQL will have performance similar to INGRES an UNIFY We think that another order of magnitude in improvement is to be had by more work in the spirit of RAD-UNIFY, since we obtained substantial performance improvements with so little work starting with a conventional database system not optimized for main memory speeds Without such work on performance, neither object-oriented nor relational database systems will be suitable for many engineering applications

References

Astrahan, M M et al, "System R A Relational Approach to Database Management", *ACM Transactions on Database Systems*, Vol 1, No 2, June 1976

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

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

Brown, M R , Kolling, K N , Taft, E A , "The Alpine File System", *ACM TOCS* 3, 4, November 1985

Cattell, R G G , *Design and Implementation of a Relationship-Entity-Datum Data Model,* Xerox PARC technical report CSL-83-4, April 1983

Dittrich, K , and Dayal, U , Eds , *International Workshop on Object-Oriented Database Systems*, September 1986

Learmont, T R , and Cattell, R G G , "An Object-Oriented Interface to a Relational Database", submitted to a follow-up publication to the International Workshop on Object-Oriented Database Systems, to appear 1987

Stonebraker, M , "Object Management in POSTGRES Using Procedures", *International Workshop on Object-Oriented Database Systems*, September 1986

Sun Microsystems, Inc *SunSimplify 1 0 Manuals* Order Number SunSimplify-09, Sun Microsystems, Mountain View, California, 1986a

Sun Microsystems, Inc *SunUNIFY 2 0 Manuals* Order Number SunUNIFY-09, Sun Microsystems, Mountain View, California, 1986b

Sun Microsystems, Inc *SunINGRES 5 0 Manuals* Order Number SunINGRES-09, Sun Microsystems, Mountain View, California, 1987