Measured performance of a special-purpose database engine for distributed system software

Petri Kutvonen and Lea Kutvonen*

Department of Computer Science, University of Helsinki
P.O. Box 26 (Teollisuuskatu 23), University of Helsinki, Finland
Petri.Kutvonen@cs.Helsinki.FI, Lea.Kutvonen@cs.Helsinki.FI

Abstract

This paper reports the measured performance of a special-purpose database engine that was developed to meet the requirements of modern, open and distributed, systems. The database engine is memory based, relational and supports properties that enable it to be used as an object based system. As the goal of this database engine is to support general functions needed in distributed system, we justify the feasibility of our database engine realization by a numerical example involving a telephone switching system. The example shows, that the measured database engine performance is of the correct magnitude for the needs of a real distributed system.

1 Introduction

The fast evolution and growth of open distributed computing environments is supported by new architecture models. One of the well accepted and upcoming models is the Open Distributed Processing Reference Model (ODP-RM) [2].

The ODP-RM specifies an abstract computing engine, an infrastructure, that identifies the fundamental services required in each subsystem that is willing to be part of the open

*This work was partially supported by TEKES, the Finnish Technology Development Center.

global computing system. The key services of this system software are a binding function, a type repository [2] and a trading function [3]. The binding function creates channels between objects across the global system. The model supports late binding of objects: a channel is created at the time of service request. The trading function discovers the most suitable service provider from the network at the time of channel creation. Trading requires the help of the type repository. The type repository matches the service requester's view of the service to the corresponding view at the service provider's interface.

These fundamental services heavily trust on fast, distributed repositories. Both the trading information and the type information set special requirements for the underlying storage or database system.

The requirements can be classified as follows:

- The access and update of database items must be very fast.
- The databases must be able to handle transient data. Data values need not to be static, but can be evaluated by demand at the time of database access. The evaluation of virtual values becomes part of the search process within the database, and it should allow reasonable amount of concurrency but still remain in given resource boundaries. The mechanism for accessing transient data can also be used for predicting new values, based on the history of previous values.
- The access time and the cost of searches from the database should be minimized by caching the transient data values after evaluation. Yet, the caching mechanisms should be sensitive to the class of data and the cost of its evaluation.
- The database engine should be able to give answers to queries even when some of the data is missing, or the evaluation of transient data is unsuccessful (within given time limits). Incomplete results shall be handled separately within the actual services themselves (the trading function, the type repository function), because of the different semantics.
- There is a need to update the database schemas during the usage of the database without disturbing the running searches. In a dynamically evolving distributed system, new service types and new policies for cooperation behaviour are introduced constantly in different parts of the global system.
- Most importantly, the system functions require access to a distributed collection of information. Each subsystem has autonomy in the support and the usage of service types and service providers. Therefore, each subsystem should have a private database. The subsystems can independently select the database technology. They can also independently decide about the database schemas. However, when cooperation with other subsystems is required, part of the database schemas must be negotiated by the system administrators. The requirement of autonomy does not necessarily lead

to a distributed database problem: the system function standards will eventually describe, how agents at each subsystem should act as clients for each other [2, 3].

We have developed a database engine, Debbie, to meet these requirements. In this paper, we describe some measurements on the performance of the Debbie database engine. Measurements include,

- the response time as a function of the database size,
- the response time as a function of the number of concurrent clients,
- the response time as a function of the number external evaluations of transient data values performed during the query, and
- effects of caching the transient data values to the response time.

The test cases run in a distributed environment with UNIX computers connected through Ethernet. This environment provides a lower bound for the performance of the system software functions. Higher speed can be achieved, for example, by replacing the Ethernet with an ATM network.

We also discuss the feasibility of the requirements set on the database performance by the basic system function repositories. The discussion is based on an numerical example with a TINA [6] style environment where the trading function is used in switching telephone calls [7].

2 The database engine

Debbie is a simple database management system. It has been designed for applications where the flexibility and the performance of database operations is more important than the stability of storage. This does not imply that Debbie were unsafe, but it implies that no heavy log mechanism is included. Debbie trusts on the recovery mechanisms offered by the underlying system environment. Debbie is not designed to handle really large databases. Instead, we can view databases managed by Debbie as an intermediate form of memory structure and a fully structured database.

Debbie is a relational database system that includes the normal relational operations: projection, selection and join. Besides the usual data manipulation operations Debbie offers partial string matching and approximate string matching [8]. Although Debbie is basically relational, it includes special features that enable it to be used as an object based system.

Debbie allows stored data to be viewed as methods which can be invoked. The methods can consist purely of Debbie's own operators or they can include external procedures. Execution of methods can be parallel. A relation table can be viewed as an object class and each row of the table as an object. Execution of the same method for all objects in a class can be triggered with one Debbie operation. Failing external procedures do not imply the failure of the entire operation, instead a partial result is generated.

To accommodate the multitude of requirements set by distributed systems, the database schema can be manipulated. For example, new object classes or subclasses can be permanently introduced at run time. Also, all data stored into Debbie has free format. Fields have no predefined sizes or types. In individual operations Debbie makes automatically a difference between numerical and textual data and is able to handle each type with suitable mechanisms.

Internally Debbie stores databases in virtual memory. This makes Debbie work fast but it limits the sizes of the databases. However, with today's memory sizes manipulation of tables of several megabytes in size is possible. Externally Debbie tables are stored as flat files. Thus the tables can easily be manipulated also with other tools.

Debbie offers locking. The granularity of locks vary from whole tables to field based locks.

The command language is by a design decision not SQL like. An interpreter for an SQL based query language could be constructed, but we did not consider this an essential design goal.

Debbie operations are usually invoked through a database library interface. The interface is designed for the application programmer to use and it allows operations to be activated over a network. To utilize the services of the library a database server must be running, either or the same or a different host as the client.

To achieve high performance, Debbie uses lazy evaluation and converts tables from external to internal form only when the values are really needed for the first time. All tables generated through invocation of external procedures are cached and regenerated only when the values are too old to be considered valid. In addition to this, a pre-evaluation of frequently used externally generated tables is automatically activated. The cache timeout and the pre-evaluation threshold can be set individually for each object class.

The Debbie implementation runs in several UNIX environments (including SunOS, AIX, Ultrix, OSF/1, System V, Linux [5]) and in a restricted form also in VMS and MS-DOS environments.

3 Measurement results

The number of interesting parameters which affect the performance of a database system is relatively large. Studies on performance evaluation of real-time databases are available, see e.g. [1]. These studies mainly concentrate on parameters, such as locking, which are relevant for the database system design itself, but not enough when gathering information to be used as a basis for performance design of future distributed system as whole. We therefore focus mainly on communication, concurrency and caching issues.

We use the response time as the fundamental performance measure. We define it as the average elapsed time to process a query, as seen by a client process. Sometimes it is advantageous express this quantity as a query completion rate, i.e., queries per second.

The measurement results were obtained by using an instrumented database client program which generated the required synthetic workload. All measurement results represent the averages for at least 2000 processed queries and all measurement series were repeated at least three times.

The actual measurement environment consisted of PCs based on 90 MHz Pentium processors running Linux version 1.2.8. The database server run on a machine with 32 MBytes memory, the clients had 16 MBytes. The PCs were connected using normal 10base2 Ethernet. The PCs were dedicated for measurements, but there was, nevertheless, a light background load on the Ethernet.

This environment represents technology which is currently readily available. By replacing Ethernet by an ATM network and the PCs with more powerful special purpose processors, higher performance can be achieved. However, there can be some inherent limits in the TCP/IP protocols which prevent linear speedup with hardware power increase.

3.1 Effect of database size and communication

Although our main interest is not in traditional performance parameters, we first wanted to consider the effect of the database size on the response time. At the same time we measured the effect of different communication scenarios. The results of these measurements are shown in Figure 1.

The database table consisted of two columns, a key and a value, all keys being non-numeric and unique. The number of rows in the table varied from 100 to 100,000. The query consisted of the selection of one row based on a random but existing key.

The three communication scenarios were the following:

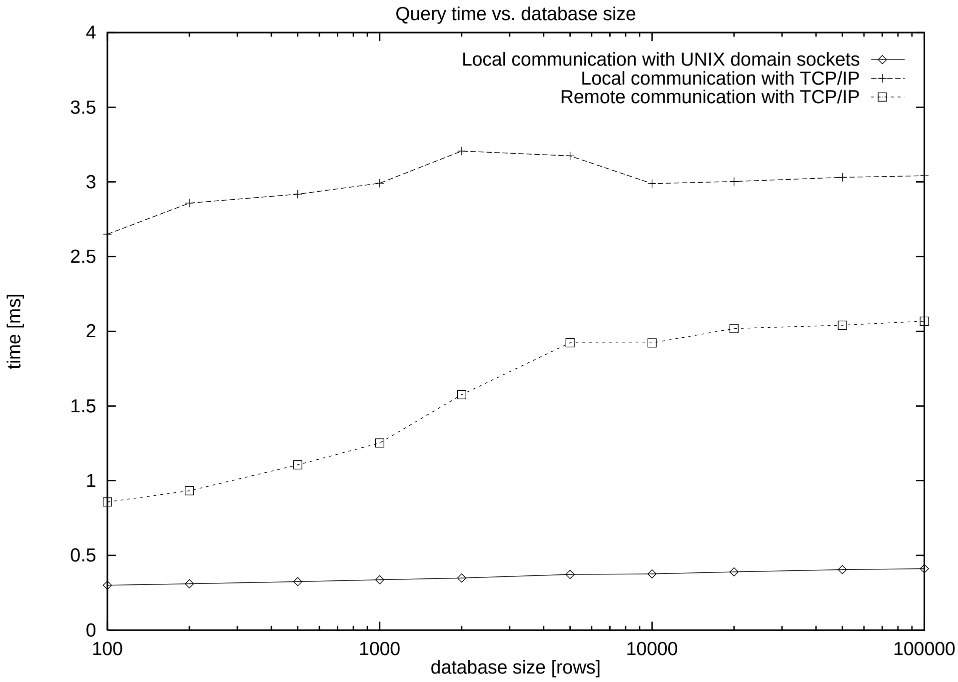

Figure 1: Query time vs. database size.

- The client and the server are located on the same host. We use the fastest communication channel available between them. As these tests were run in a UNIX environment and as Debbie does not support shared memory as a communication channel, the fastest way is to use UNIX domain sockets. This scenario corresponds to a situation where, e.g., the objects realizing the trading function and the type repository function are located in the same node.
- Also in the second scenario the client and the database server are located on the same host. In this case, communication between them takes place over a complete TCP/IP communication stack. This corresponds to a situation where the communication parties are located in the same node but the server is prepared to communicate with remote clients as well. TCP/IP might not be the communication protocol of the future, but it represents a fairly well optimized contemporary solution and gives a good approximation of the amount of processing time required for protocol overhead.
- The third scenario is a true client-server model. The client and the server are located on distinct hosts and communicate over TCP/IP.

We should expect to see logarithmic behaviour. Indeed, the first scenario shows a very steady, logarithmically increasing average response time. However, even there we can see a comparatively large constant extent which is due to the communication overhead.

We can obviously achieve a response time of less than 0.5 ms (corresponding to a query processing rate of over 2000 queries per second) even with rather large tables.

Communication over TCP/IP shows a more fluctuating behaviour. The communication overhead dominates and is on a significantly higher level even when both parties are located on the same host. This is caused by the delay required to process queries going up and down the TCP/IP stack. Interestingly, the overhead is much smaller when processing smaller database tables. This is probably due to the memory caching in the computer hardware: when processing small tables, it is possible to have every bit of program code and data in the cache memory, but when table sizes get larger, references to the main memory are required.

The remote communication scenario shows a rather stable behavior in the 3 ms range. The slightly shorter response times with small tables are most likely caused by similar reasons as in the local TCP/IP scenario.

The typical usage pattern of a distributed system repository involves small bursts of independent queries. When there is no causal relationship between the answer of one query and the arguments of the next one, it is possible to package more than one query in one communication level message. The effect of the packaging or the message blocking factor is shown in Figure 2.

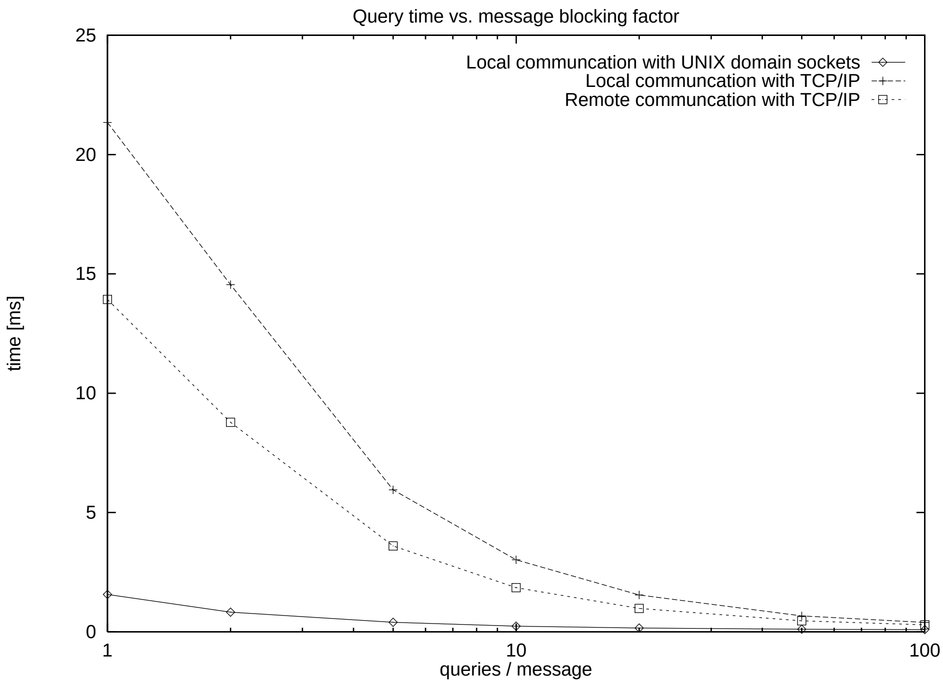

Figure 2: Query time vs. message blocking factor.

In this experiment we used a similar table and a similar query as in the previous one. The table size was fixed at 10,000 rows. The number of queries in a block varied from 1 to 100. The actual size of a single query, as transmitted between the client and the server, was about 50 bytes.

The results show that increase of the number of queries per message has a drastic effect when TCP/IP communication is used, either local or remote. The relative performance of the three communication scenarios seem to be independent of the message blocking factor.

A typical independent burst size of 10 queries was assumed in all the other experiments (including the first one).

3.2 Effect of concurrency

Next we considered a case where we can have more than one client generating the queries as fast as they can. Each client must wait for the responses to its previous queries before issuing a new burst of 10 queries. The individual clients are independent of each other. In this measurement all the clients were located on separate hosts and all communication takes places over TCP/IP.

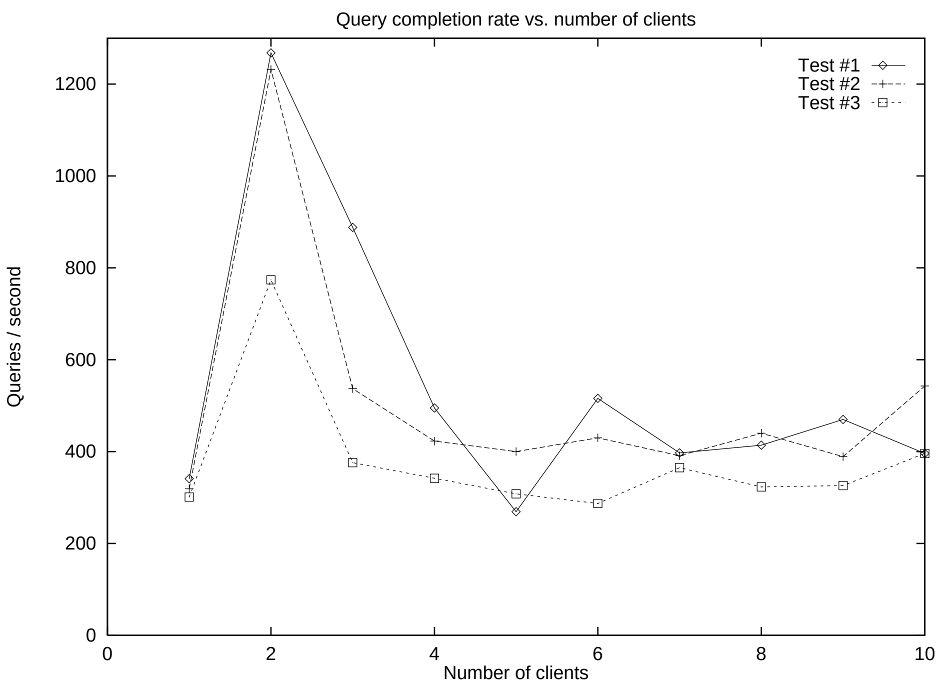

Figure 3: Query completion rate vs. number of clients.

The performance is presented as the server throughput. Due to the more unstable behaviour we present in the Figure 3 the results of all three test runs. Instability is of course inherent in the distributed environment. Although the clients were dedicated to the measurement, some instability was probably caused by different background Ethernet loads during each run.

We should expect to see a behaviour where the increase of the number of clients first improves performance, but having too many clients has a negative effect. The measurements confirm this hypothesis. We get the by far best results with two clients. Just one client can not keep the database server busy, whereas more than two or three clients cause extra overhead, especially when communicating over a shared media like the Ethernet.

3.3 Effect of dynamic values and caching

The management of transient data is specially important in the implementation of a trading system. A trader mediates services and therefore it needs to store properties of services and service providers. Some of the properties are so dynamic that they can not be stored statically in a database but need to be evaluated by demand. The object oriented features of Debbie can easily accommodate this requirement.

Because of the transient nature of the dynamic values and as there is no exact requirement when they have to be computed, the database system can exploit caching techniques. It can store old values and reuse them when responding to subsequent queries. Figure 4 illustrates the average respond time as the function of the number of (independent) dynamic values in a query. We show the measurement results without a cache as well as with a cache with 5, 10 and 20 seconds timeout. The client and the server were located on the same host. All the measurements concerning caching effects were made after a sufficient warm up period.

We used a synthetic external process to accomplish the evaluation of the dynamic values. The elapsed time of the external process was set to be always one second, and the processing time was negligible. This corresponds, e.g., to reading a sensor or to inquiring a parameter over a network.

The Debbie database system can invoke any number of external processes concurrently. For maximal performance we have to set a reasonable upper bound for the amount of concurrent processes. In this experiment we used an arbitrary value of 8.

In the case without a cache, the results show an expected behaviour where the external evaluation time dominates. The concurrency of the evaluations makes the behaviour sublinear and we can see small steps in performance at the multiples of the chosen concurrency limit of 8.

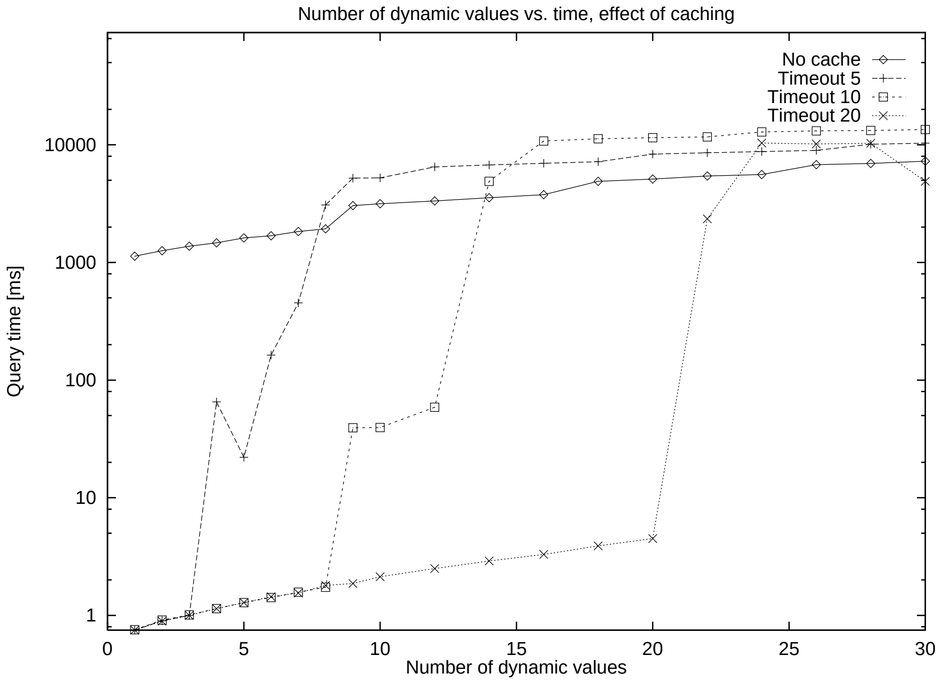

Figure 4: Query time vs. number of dynamic values.

With the cache, the behaviour is totally different. If the cache timeout is long enough, the majority of the requests are satisfied from the cache. As soon as we reach the limit, where the time required to process the queries exceeds the caching time, we see the performance to drop several orders of magnitude. This means that the choice of an appropriate cache expiry timeout is of great importance. With a careful selection of this parameter for each class of data, we can still reach the performance of several hundreds of processed queries per second.

3.4 Effect of cache prefetching

Prefetching is a well established technique to improve the performance of memory and disk caches. Prefetching means that the cache tries to predict what data are needed next and starts a fetch before the values are actually needed. It is possible to apply the same technique to the cache of dynamic values. If a table is referred frequently enough this can be used as an indicator to trigger a new evaluation. Debbie implements this by using a prefetch threshold which is shorter than the cache expiry timeout: the first reference to a table after the threshold returns the cached values, but also invokes pre-evaluation of the values in the background. The previously stated concurrency limits apply also to this background evaluation. The results with and without prefetching are shown in Figure 5.

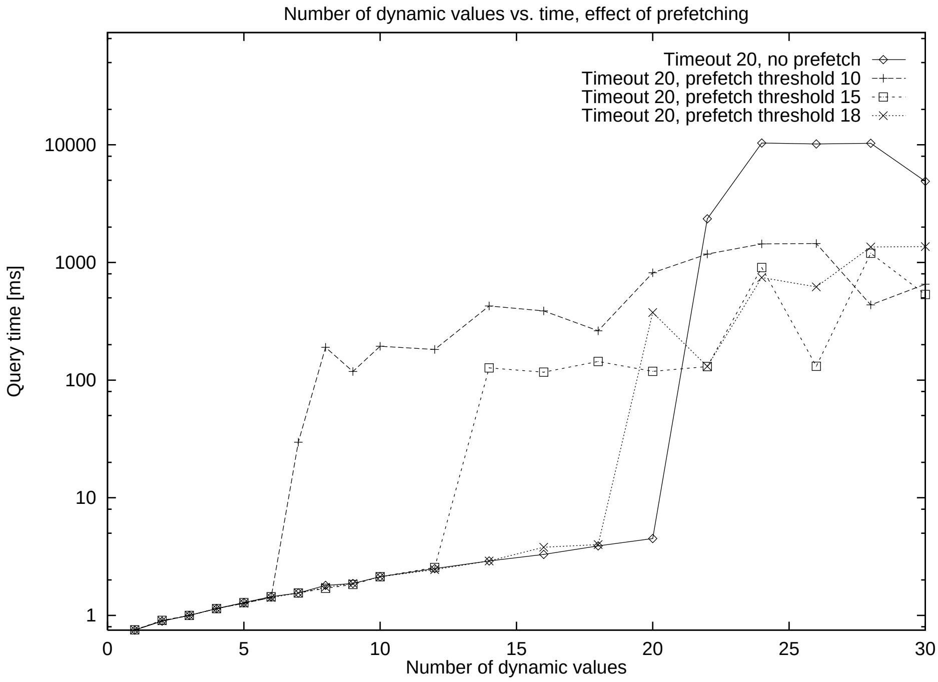

Figure 5: Query time vs. number of dynamic values, effect of prefetching.

The results show that prefetching may hamper performance. In the previous experiment we saw a drastic chance in performance at the point where the average total query processing time exceeds the caching time. If the difference of the expiry timeout and the prefetching threshold is large compared to the actual evaluation time, the inferior performance begins with a smaller number of dynamic values. This means that if the prefetch is started too early, fewer dynamic values can be efficiently managed. By choosing a prefetch threshold which is carefully aligned with the cache expiry timeout and the average evaluation time, we can overcome the problem almost entirely. Ideally, the prefetch should start so that the evaluation always finishes just in time before the cache timeout.

The more important observation is that by using pre-evaluation the unavoidable decrease of performance with a large number of dynamic values is smaller than without pre-evaluation.

4 Feasibility

In the the following, we consider the feasibility of realizing the trading function in the reflection of the presented measurement results. The study is based on an example given in [7].

The example describes a TINA consistent telephone service for an area of 3 million inhabitants. We consider the busy hour behaviour where all subscribers generate three calls per hour and all calls are answered. Call setup requires that several interfaces are located. The TINA infrastructure uses the trading function to locate the interfaces but can also cache the interfaces by itself. Therefore, not every interface lookup results in a trading request. A mixture of 80 % ordinary telephone calls and 20 % Freephone calls (which require more elaborate processing) generates at least 10 million trading requests per hour or about 2800 per second.

Based on the experience with the Dryad trader [4] we estimate that every trading request typically generates approximately 10–20 requests for the underlying database engine. This corresponds to an average of 42,000 queries per second.

A requirement to have just one trader and one database engine to process all the requests is not reasonable as stated in [7]. The requests are mainly independent of each other, and therefore partitioning the workload over, say, 20 traders is not unreasonable. This is similar of having a single trader for each 150,000 subscribers. If we assume an even division of load, this results in a requirement to be able to process 2100 queries per second per database engine.

If we compare this requirement with our measurement results, we observe that it can almost be satisfied even with today's technology, especially if the realization of the trading function itself is not distributed. Even if we allow distribution or employ a reasonable number of dynamic values the necessary performance increase to accommodate the requirement is only about tenfold. This should be easily achievable by using an ATM network and more powerful processors.

5 Summary

The functionality of modern, open and distributed, systems sets new requirements for the database systems they exploit. In this paper, we have identified those requirements, and introduced the Debbie database engine that has been developed to meet them.

Major requirements are the fast responses, the ability of manipulation on transient and missing data, and the flexibility of the database schemas. The Debbie database engine is a relational system, but memory based. It also supports features that makes object based usage possible. To achieve high performance, Debbie uses lazy evaluation and prefetching when applicable.

To evaluate the implementation against the requirements, we have carried out a set of measurements. In this paper, we have shown response time as functions of the database size, the number of concurrent clients and the number of external evaluations of dynamic data values, together with the effect of caching and cache prefetching.

As the objective is to support functions needed in open distributed system, we have considered a numerical example that involves a telephone call switching system. The system utilizes a trading system for several tasks and therefore has a considerable requirement of trading request throughput. Nevertheless, by the example we have shown that, under good conditions, the trading system can produce the required magnitude of throughput even with the current memory based database engine.

References

[1] Gruenwald, L., Liu, S., A Performance Study of Concurrency Control in a Real-Time Main Memory Database System. ACM Sigmod Record, 22,4 (1993).

[2] Information Technology - Open Systems Interconnection, Data Management and Open Distributed Processing - Basic Reference Model of Open Distributed Processing - Part 3: Prescriptive Mode. Draft International Standard. ISO/IEC DIS 10746-3 — ITU Rec. X.903. 1995.

[3] Information Technology - Open Systems Interconnection, Data Management and Open Distributed Processing - ODP Trading function. Draft International Standard. ISO/IEC DIS 13235 — ITU Draft Rec. X.9Tr. 1995.

[4] Kutvonen, L., Achieving interoperability through ODP trading function. Proceedings of the 2nd International Symposium on Autonomous Decentralized Systems (ISADS'95). Phoenix, 1995.

[5] Linux Information Page, http://www.cs.Helsinki.FI/linux/, University of Helsinki, Department of Computer Science, page introduced 1994.

[6] Nilsson, G., Depuy, F., Chapman, M., An Overview of the Telecommunications Information Networking Architecture. Proceedings of TINA'95. Melbourne, Australia, 1995.

[7] Parhar, A., Rumsewicz, M. P., A preliminary investigation of performance issues associated with Freephone service in a TINA consistent network. Proceedings of TINA'95. Melbourne, Australia, 1995.

[8] Ratcliff, J., Pattern Matching by Gestalt. Doctor Dobb's Journal, Issue 181, July 1988.