Performance Analysis of an Associative Caching Scheme for Client-Server Databases

Julie Basu
julie@cs.stanford.edu
Stanford University
and
Oracle Corporation

Meikel Pöss
poess@cs.tu-berlin.de
Technical University of Berlin
Computer Science Dept.
Germany

Arthur M. Keller
ark@cs.stanford.edu
Stanford University
Computer Science Department
Palo Alto, CA 94305-9020

September 29, 1997

Abstract

This paper presents a detailed performance study of the associative caching scheme proposed in [11]. A client cache dynamically loads query results in the course of transaction execution, and formulates a *description* of its current contents. Predicate-based reasoning is used on the cache description to examine and maintain the cache. The benefits of the scheme include local evaluation of associative queries, at the cost of maintaining the cached query results through update notifications from the server. In this paper, we investigate through detailed simulation the behavior of this caching scheme for a client-server database under different workloads and contention profiles. An optimized version of our basic caching scheme is also proposed and studied. We examine both read-only and update transactions, with the effect of updates on the caching performance as our primary focus. Using an extended version of a standard database benchmark, we identify scenarios where these caching schemes improve the system performance and scalability, as compared to systems without client-side caching. Our results demonstrate that associative caching can be beneficial even for moderately high update activity.

1 Introduction

1.1 Client-Server Databases with *Smart* Clients

A popular architecture for modern database management systems is the client-server configuration. This setup involves one or more server processes that manage a central repository of persistent data, and handle requests for data retrieval and update from multiple client processes. The configuration is non-shared memory, so that the server and the clients have mutually disjoint local address spaces, and communication between a client and the server occurs only through explicit messages across a network. The server provides concurrency control, transaction management, and recovery facilities for the shared database.

Server response is a critical factor in the performance of a client-server system. The resources of the server are shared among many clients, and can become the bottlenecks in scaling the system to large amounts of data or larger number of clients. Optimizing the performance of the server has thus been a major focus of commercial systems. For example, the Oracle database [13] uses a main-memory “buffer pool” of cached pages to avoid disk traffic at the server. For clients however, a common assumption in the past has been that they have very limited resources. Accordingly, client functionality has been limited to transmission of queries and updates across the network to the server and presentation of the received results to the user. In this type of system, known as a *query-shipping* system, all query execution tasks occur at the server. Most relational client-server databases available commercially fall under this category.

The revolution in computer hardware technology has made client resources relatively cheap and plentiful. Today, clients processes often run on powerful desktop machines with substantial CPU and memory. These *smart* or *thick* clients are capable of performing intensive computations on their own, using the database as a remote resource that is accessed only when necessary. Better utilization of client resources can offload the central server and improve system performance and scalability. Client memory and CPU should therefore be taken into consideration for data caching and query evaluation purposes. Despite the potential cost of maintaining data cached at client sites, a number of recent studies [6, 18, 19] have demonstrated that inter-transaction reuse of cached data can decrease network traffic and query response times.

1.2 *Associative* Versus *Navigational* Caching

In contrast to the query-shipping strategy followed by relational databases, most object-oriented databases assume smart clients, and are built using a *data-shipping* approach. In this model, the client cache is essentially a pool of individual data items (either pages or objects), with a table of contents that lists the unique identifiers of cached items. Although the effectiveness of such data caching has been demonstrated by several studies [2, 5, 6, 18], cache access based on identifiers is adequate only for purely *navigational* operations such as *ReadObject* and *UpdateObject* in object-oriented databases. An *associative* query that specifies a target set of tuples or objects using predicates on the attributes of a relation or an object class, e.g., through a WHERE clause in a SELECT-FROM-WHERE SQL statement, is not easily supported by these caching schemes.

Associative queries are widely used in relational and object-relational database systems, and are indeed one of the major reasons for their success. Even in object-oriented databases, a common way of data retrieval is by using predicates on one or more attributes of an object type, with navigational access occurring subsequently as objects in the result set are traversed. Evaluation of associative queries on locally cached data is therefore important for both relational and object-based database systems, and can potentially improve cache reuse and client CPU utilization in both cases. To address this problem, an associative caching scheme for client-server databases was proposed in [11]. This caching strategy, now called *A*Cache*, is an example of the *hybrid-shipping* model [7], and it attempts to exploit the resources of both the clients and the server in a flexible way. The aim of this paper is to evaluate the performance of this caching scheme through simulation of a client-server database, focusing primarily on the effect of updates.

1.3 An Example

We use an example below to illustrate the use of associative caching and the issues in maintaining an associative cache when updates are committed to the database. Consider a large software company with a primary Development Center at a single site and many international Sales and Support offices. Suppose that it has a BUG database to help track software defects encountered by its customers. A bug logged in the database has an associated product, a customer, a problem description, and a current status, and is represented using the following relational schema:

BUG (*Bug#, Product#, Customer, Description, Status*),

where each row records the details of a particular bug.

Usage of this database is as follows. Development and Support personnel are assigned to specific products, so that queries to the database typically retrieve a set of bugs satisfying a predicate such as (*Product# = 100 AND Status='Open'*). Other queries include those that gather statistics on bug activity for different products or customers. Bugs may be updated after bug fixes or in response to customer interaction.

Let us assume that the BUG database is implemented using the client-server model, and that the central server is at the primary Development Center. Clients may be located at the Development Center or at any Support site in the world. Clearly, client-side caching would be beneficial in this scenario, since latency to retrieve data from the central server would be substantial otherwise. Note that clients of the database are accessing data in an associative way, e.g., based on the product information and the status of bugs. Thus, for effective cache reuse at the client, associative access to cached data is needed.

Suppose that a client C caches only the result of a query for open bugs in product number 100, along with a predicate description P for these tuples:

P : (*Product# = 100 AND Status = 'Open'*)

Assuming that none of these bugs are updated, a subsequent query at the client C for all open bugs in product number 100 reported by customer X, i.e., those bugs that satisfy the predicate:

Q : (*Product# = 100 AND Status = 'Open' AND Customer = 'X'*)

can be answered using containment reasoning to detect that the query predicate Q is a subset of the cached predicate P. Cached data can also be updated locally without getting server locks, if the client optimistically assumes that the data is current, and is not being updated elsewhere (in this case, update conflicts would be detected later, e.g., at commit).

Now suppose that a bug satisfying the condition (*Product = 100 AND Status = 'Open'*) has its status updated to 'Closed' by a different client. This update potentially affects all caches that have cached query results on the BUG relation. In the case of client C, there are three possible ways to reflect the update on its cache: (1) it can delete the tuple from the cache, since no cached predicate is satisfied by the updated tuple, (2) it can update the tuple and retain it in the cache, but since the tuple does not satisfy predicate P any longer, it must also augment its cache description to include the new tuple, and (3) it can purge the predicate P from its cache, which might be preferable if it is known that the query result is not used frequently. For another client with a cache description (*Customer = 'Y'*), updating the modified tuple if it is present in its cache is an appropriate maintenance action that preserves the validity of its cache.

1.4 The A*Cache Scheme

We briefly review below the scheme presented in [11]. Queries submitted to the server are used to dynamically load data into the client-side cache, and *cache descriptions* derived from the query predicates are stored at the client as well as at the server to examine and maintain the cache contents.

When a transaction submits a query or update, it is intercepted locally by the client, and compared against the cache description. If the cache is found to contain the required data, the client executes the operation locally on the cache, following an optimistic concurrency control protocol. If all the required data is not available locally, a request is submitted to the server to fetch either the missing data only or all of it, depending on the costs involved. The returned results may or may not be cached upon return to the client, based on the space availability and other local conditions. A remote request is accompanied by any local (uncommitted) updates of which the server has not yet been informed, since the transaction must be able to see the effect of its own updates for all local as well as remote operations. As described in [11], A*Cache follows a *semi-optimistic* concurrency control protocol that retains the serializability of transactions.

A commit flushes all local updates performed previously at the client to the server. In order to ensure serializability, the server must ensure that the client has seen its most recent notification message before the commit is confirmed, and the deferred updates are posted to the database. This checking is done by assigning a sequential message number to every notification.

1.5 Cache Consistency in A*Cache

An important concern in any caching scheme is consistency maintenance of the cached data — client caches may become out-of-date as updates are committed on the central database at the server. Handling updates is more complex for an associative caching scheme compared to a identity-based one, since it is not sufficient to consider data items individually. Instead, predicate-based reasoning must be employed to determine which client caches are affected by an update. For example, a tuple that has been updated elsewhere might have to be inserted into the cache in order to satisfy a cached predicate. Techniques for incremental maintenance of materialized views [9] are applicable in this context.

The A*Cache scheme uses a notification mechanism to maintain the validity of cached data, as described in [11]. Each client must register with the server a *subscription* that describes the query results cached by it. Using these subscriptions and incremental view maintenance techniques, the server generates any required notifications for a client whenever an update is committed at the central database. A client subscription at the server may be 'liberal' [11] in nature, implying that it represents a superset of the actual set of tuples and predicates cached at the client. A liberal approximation in the client subscription and in the notification procedure guarantees that no relevant notification will be missed, although some irrelevant notifications may be generated.

Several design issues are involved in A*Cache maintenance. An important point is that the maintenance is performed at the level of predicates. Based on the maintenance policy, such as data refresh or invalidate, tuples might need to be inserted, updated, or deleted from the cache in order to maintain the validity of its description. Modification of the cache description might also be required, e.g., if data is invalidated or purged from the cache. Update notifications may additionally affect a client transaction running at the time. Depending on its data consistency requirement, a transaction may have to be aborted when a conflict with its local reads or writes is detected upon notification of an update at the database.

1.6 Outline of the Paper

Our focus in this paper is on evaluating the performance of the A*Cache scheme. Containment reasoning is necessary on the cache description to determine if a query can be evaluated locally. The generation and processing of update notifications at the server and at the client also requires reasoning with query predicates to detect overlap of updated regions with client cache descriptions. Predicate-based reasoning in a dynamic caching environment naturally raises questions of feasibility and performance, which are addressed in this paper.

We develop a simulation model of the A*Cache system using queueing networks. The model takes into account the client-side caching facilities and the associated retrieval and maintenance costs, as well as the cost of notification generation by the server. We also introduce an optimized version, called *A*Cache_Opt*, of the basic A*Cache scheme that considers partial containment of a query in the cache and fetches only the missing data from the server. Simulation experiments are performed for a variety of workloads with different data usage and contention patterns, with a no-caching system as the basis of comparison. We first investigate the effectiveness of A*Cache and A*Cache_Opt in a read-only scenario, and then explore their behavior in the presence of update notifications and cache maintenance activity. Our results demonstrate that both the A*Cache schemes can provide substantial performance benefits even under moderately high update loads.

The rest of this paper is organized as follows. Section 2 reviews related work. In section 3, we briefly describe the components in an A*Cache system. The simulation environment with the system model is described in Section 4. Section 5 defines the workload models, and reports the results of simulation experiments for A*Cache and A*Cache_Opt. Finally, we summarize our conclusions in Section 6.

2 Related Work

Client-side data caching that is based on object or page identifiers has been studied quite extensively in [6, 18, 19]. However, associative caching has not been examined in any of these papers.

Recently, a couple of studies [3, 16] have examined associative access to a client cache. Both of these studies are limited to read-only scenarios. The *semantic* caching study in [3] examines cache replacement policies for no-update workloads. The design of an cache manager called WATCHMAN for caching read-only query results in data warehousing environments is reported in [16]. The important issue of cache maintenance when there are database updates is not examined in these papers.

Evaluation of queries from locally stored views is supported in the ADMS$\pm$ system, which uses the *ViewCache* technique and *incremental access methods* [15]. A simulation study [4] examined the performance of this and related schemes. One major difference in A*Cache is that its update propagation scheme is based on asynchronous notification of committed updates, instead of on-demand lookup of server update logs. Additionally, the concurrency control scheme in A*Cache is semi-optimistic, and permits both local reads and writes with a commit verification step.

A caching subsystem that can reason with stored relations and views is proposed in the *BrAID* system [17] to integrate AI systems with relational DBMSs. Some aspects of *BrAID* that pertain to local query processing, such as query subsumption and local versus remote query execution, are relevant for A*Cache. Unlike A*Cache, *BrAID* does not consider database updates.

Distributed and replicated databases are also related to the A*Cache scheme. However, the A*Cache environment is different in several respects: (1) Caching is dynamic in nature, unlike static partitioning in replicated systems; (2) client caches are not full-fledged databases; e.g., they do not provide local commit or recovery services; and finally, (3) the central database is the single 'point of truth' for all persistent data, and the commit always takes place at the server. Therefore, conflict resolution issues for multiple copies of replicated data do not arise in A*Cache.

Maintenance of materialized views has been the subject of much research [9], and is relevant for consistency maintenance of A*Cache. For example, the update notification scheme in A*Cache is related to the filtering of updates that are irrelevant for a view [1]. However, performance issues in handling large numbers of dynamic views in a client-server environment, as in A*Cache, have not been considered in these papers. Query containment [10] is a topic closely related to the cache containment question, and techniques developed in this area are directly applicable for A*Cache.

3 Architecture of an A*Cache System

We now describe the physical architecture of an A*Cache system. The persistent data store is resident at the central server, and transactions are initiated by autonomous clients across a network. Separate subsystems exist at each client site and at the central server for cache management. For this paper, we assume that a client runs a single transaction at any given time. Thus, we do not consider local concurrency control and lock management at the client, although they can be incorporated within the A*Cache framework. Figure 1 shows the components in an A*Cache system for a single client.

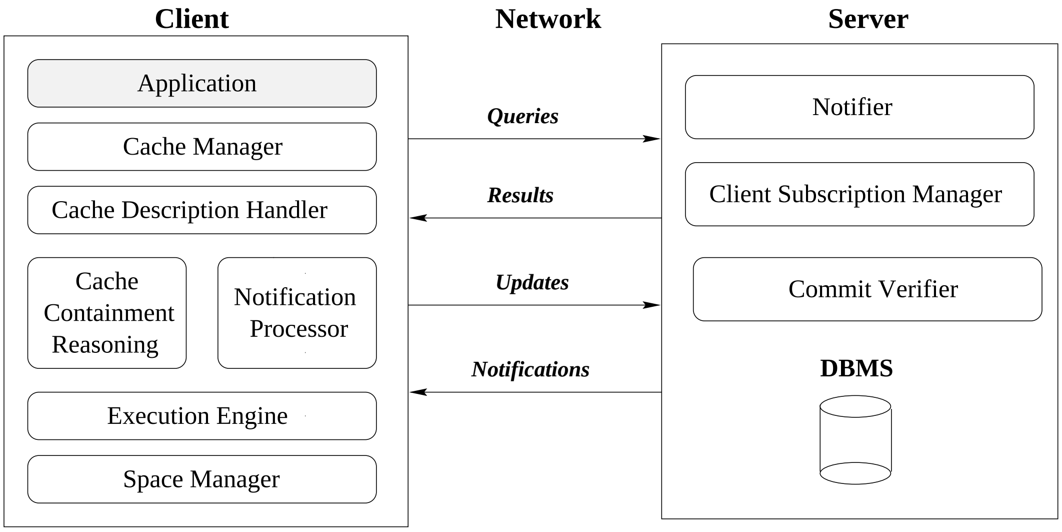

Figure 1: *Components in an A*Cache System*

3.1 Client-Side Components

The client-side subsystem of A*Cache consists of six distinct components:

- A *Cache Manager* which is the interface to the client cache. This module serves as the 'dispatcher' component at the client. It intercepts database operations requested by client transactions, as well as the update notifications received from the server, and routes the request appropriately. It also participates in the commit verification algorithm for our semi-optimistic concurrency control scheme, to ensure that a committing transaction did not read or write old data (Section 1.5).
- A *Cache Description Handler* for inserting and deleting cached predicates. Modifications to the cache description may be required (1) when a new query result is stored, (2) when a previously cached result is purged from the cache, and (3) during processing of update notifications. This module also keeps track of usage information for predicates, which is used for space management purposes.
- A *Containment Reasoning* subsystem for detecting cache containment. This module 'conservatively' [11] compares a query predicate against the cache description to determine whether some or all of the required data is contained in the cache. This module is invoked when: (1) a new query is submitted, and (2) a new notification message arrives from the server, and it must be checked if the cache is affected. Query containment algorithms [10] are employed in this process.
- A *Query Execution Engine* that operates on the data in the cache. Queries and updates need to be executed locally in the cache in three cases: (1) when there is a cache hit, (2) in response to an update notification message, and (3) for reclaiming space from the cache. For cache hits, query execution plans are constructed at the client, and the query or update is executed on the cached data. Note that the execution system must also provide local rollback and abort facilities for local updates made in the cache, in case the transaction is later aborted due to serializability considerations.
- An *Update Notification Handler* for cache maintenance operations. Upon receiving an update notification from the server, the Notification Handler first invokes the Containment Reasoning system to determine whether the cache is actually affected. This step is required since notification is liberal and involves network delays, and a notification may be irrelevant for the current contents of the cache.

  Containment analysis on a notification may have three outcomes: (1) the cache is not affected, (2) only the cache, and not the current transaction, is affected, and (3) both the cache and the current transaction are affected. For this paper, we assume that the cache is updated upon notification, and that a client transaction is aborted and restarted if a conflict is detected with local reads or writes.
- A *Space Manager* for loading and discarding query results. The Space Manager decides whether a new query result should be cached, and implements a cache replacement policy when the cache is full. When predicates and associated tuples are purged from the cache, the cache descriptions at the client and the server must be changed accordingly. As described in [11], reclamation of space is done using a reference counting scheme, so that a cached tuple is purged only when it is not referred to by any cached predicate. Advanced functionality may include defining local indexes on the cached data to improve the efficiency of data retrieval.

3.2 Server-Side Components

Special facilities also exist at the server for supporting and maintaining client-side caches. These modules are described below.

- A *Client Subscription Manager* for registering client subscriptions. Before a query is cached at the client, it must first be registered with this server module, so that the client is notified of all relevant updates. This module also handles requests for deleting predicates from a client subscription when it has been purged from the cache for space reclamation.
- A *Commit Verifier* to ensure serializability of transactions. The commit process is enhanced to support serializability of transactions that evaluate queries locally at the client [11]. A 'handshake' may be required with the client to ensure all applicable notifications have been processed by it, before the commit is finally confirmed by the server. The commit verifier interacts with the Cache Manager module at the client to enforce this additional check.
- A *Notifier* to generate update notifications. The Notifier is triggered whenever a transaction commits updates on the database, and it uses the Client Subscription Manager to determine which clients are affected by the updates committed. Generation of notifications requires determining overlap of client subscriptions with updated predicates. Techniques for incremental maintenance of materialized views [9] are applicable in this regard.

4 Simulation Environment

The sections below describe the setup for our performance analysis through simulation.

4.1 Caching Schemes for Performance Comparison

A client-server system with no client-side caching serves as a baseline for comparison of our results. The model for this system is a reduced version of A*Cache, with the differences that queries are always sent to the server, and there is no caching at client sites and no update notification by the server. Thus, no cost is incurred in cache maintenance and lookup, or in transaction aborts due to update notification.

For client-side caching, we consider two schemes that are based on the A*Cache model: (1) A basic A*Cache scheme, simply called A*Cache; and (2) an optimized extension of A*Cache, called *A*Cache_Opt*. The basic A*Cache scheme does not consider partial cache hits. Thus, if some of the tuples in the result set of a query are already in the cache, those tuples are still re-fetched from the server. Tuples required for an update are first fetched into the local cache, updated locally, and then flushed to the server at the next remote operation.

The A*Cache_Opt scheme introduces two optimizations in the basic A*Cache model:

- An optimization for queries to utilize partial cache hits. This optimization consists of detecting partial containment in the client cache, and fetching only the missing tuples from the server. Considering partial hits on the client cache can reduce the data traffic across the network compared to the basic A*Cache scheme.

- An optimization for updates on a cache miss. In this optimized scheme, a cache hit is handled in the same way as in the basic A*Cache scheme, with updates being performed locally and flushed to server with the next remote request. However, a cache miss is handled differently — it is executed completely at the server, since a network round-trip is necessary in any case. In contrast, the basic A*Cache scheme fetches all tuples required for the update, some of which may already be present in the cache, and performs the update locally, subsequently flushing these updates to the server.

  This modified update policy for a cache miss minimizes the chances of a transaction being aborted upon notification, since remote updates at the server acquire long-duration write locks for the modified tuples. It is therefore less optimistic about data writes than the basic A*Cache scheme, and can result in fewer transaction aborts.

The A*Cache_Opt scheme thus utilizes a cache miss as an opportunity to optimize network traffic for data reads, and also to reduce update conflicts for data writes.

4.2 Simulation Model

We have constructed a detailed simulation model of the A*Cache architecture as described in Section 3 above. The model is based on queueing networks. We describe below the various system resources, with the corresponding simulation parameter denoted in parenthesis.

The resources at the server consist of a CPU (*ServerCPU*) and a disk (*DiskSpeed*), along with a main-memory buffer (*ServerBuffer*) for avoiding disk traffic. The CPU is modeled as a FIFO queue. Disk pages are buffered in the server buffer following a simple LRU page replacement policy. Costs are assigned for lock and unlock actions on tuples (*LockInst*) and for detecting deadlocks. The deadlock detection algorithm is invoked when a lock request on a particular tuple fails *DeadlockInterval*-times, and it chooses a random victim among the deadlocked processes.

The database, the server buffer, the disk I/O and the data shipping over the network are organized in pages of size *PageSize*. The storage unit of the client cache is a tuple of size *TupleSize*, with each tuple being associated to one or more cached predicates. The network is a simple FIFO queue that models traffic in both directions. The network costs of sending or receiving a message include the actual wire time *NetBw*, a fixed CPU cost *MsgInst*, plus a per-page CPU cost *PerPageMI*.

There are *NumClients* clients in the system. Each client has a CPU (*ClientCPU*) modeled as a FIFO service and a main-memory cache of variable size (*CacheSize*). The storage unit of the client cache is a tuple of size *TupleSize*, with each tuple being associated to one or more cached predicates. The cost of comparing two predicates for overlap or containment is *CompCost*.

Table 1 lists the default values of the various system parameters. Values of some of these parameters, such as *BufferSize* and *CacheSize*, are varied in our experiments. The table lists the default parameter value that is used in an experiment unless noted otherwise.

Parameter	Value	Description
ServerCPU	50	Server CPU speed in MIPS
LockInst	1000	Instructions to get/release a lock
DeadlockInterval	5	Deadlock detection interval
BufferSize	500,000	Server disk buffer size in bytes (=25% of the database)
PageSize	4096	Size of a data page in bytes
DiskSpeed	6.5	Server disk speed in msecs.
DiskInst	5000	Instructions to read/write a page
NetBw	10	Network bandwidth in Mbit/sec
MsgInst	20,000	Instructions to send/receive a message
PerPageMI	12,000	Instructions to send/receive a page
NumClients	10	Number of clients
ClientCPU	25	Client CPU speed in MIPS
CacheSize	100,000	Client cache size in bytes
CompCost	1000	Instructions to compare two predicates

Table 1: *System Parameters and their Default Settings*

4.2.1 Cache Management Policies in our Simulation

The following policies are chosen for cache management in our simulation:

- Predicates in the cache descriptions at the client and at the server are not indexed, but are placed on an unordered list. Thus, detecting overlap with a query or updated predicate requires a sequential search through the list of cached predicates.
- The server assumes by default that each remote query result will be cached by the client. Accordingly, a query predicate is inserted in the subscription of the associated client for each query executed at the server. This step is completed before the query result is transmitted to the client, so that its cache is notified of all relevant updates.
- Cache replacement is done using an LRU policy at the level of predicates with reference counts for tuples, so that a tuple is purged only when its reference count is zero.
- Local indexes on cached data are not considered at client sites.

It must be emphasized that these policies are a feature of our simulator, and not of the A*Cache scheme in general. The above choices were made for simplicity of simulation logic, and for ease of interpretation of experimental results. Note that each choice is biased *against* A*Cache, in that A*Cache performance is not improved by it. A*Cache performance is only expected to be better if an alternate scheme is chosen, for example, if predicates or data are indexed at the client. Exploring alternate cache management policies, as well as extensions such as availability of client disks, is the subject of future work.

4.3 Workload Model

The workload characteristics are described below. We consider a single relation with 10,000 tuples of 200 bytes each in a Wisconsin-style setting [8]. The relation has two indexed attributes *unique1* and *unique2* that are unclustered and clustered respectively. Each query is a linear range selection either on *unique1* or on *unique2*, and its length in terms of the number of tuples retrieved is 0.5% of the database size. The predicates in a cache description are linear intervals corresponding to query ranges. In order to make a large set of experiments feasible, we kept the database small and scaled the client cache and the database buffer sizes proportionately. Table 2 shows our workload parameters.

Parameter	Value	Description
DBsize	10,000	Database size in tuples
TupleSize	200	Tuple size in bytes
QueryLength	0.5%	Query length as a percentage of the database size
PrivateRatio	50%	Private region as a percentage of database size
PrivateAccessProb	80%	Access probability of the private region
SharedAccessProb	20%	Access probability of the shared part ($= 1 - PrivateAccessProb$)
PrivateWriteProb	0%—40%	Percentage of update transactions in the private region
SharedWriteProb	0%—40%	Percentage of update transactions in the shared region

Table 2: *Workload Parameters and their Default Settings*

The Wisconsin setup is extended with locality parameters [6] to model different degrees of shared data contention among clients. The database is logically split into a shared part and a non-shared (i.e., *private*) one. Each client *owns* a section of the private region and does not access data in another client's private region. The private section for client i is defined by the linear interval:

$$\left[\frac{DBsize * PrivateRatio}{NumClients} * (i-1), \quad \frac{DBsize * PrivateRatio}{NumClients} * i\right).$$

For the private and shared regions of the database, we define a data access probability (*PrivateAccessProb* and *SharedAccessProb* respectively) and an associated update probability (*PrivateWriteProb* and *SharedWriteProb*) for transactions in that region. These quantities are the same for every client. Transactions consist of a single associative query or update, with an update transaction reading and writing half of all tuples that it accesses. Writes do not modify the indexed attributes *unique1* and *unique2*. Contention of shared data can be controlled either by decreasing the *PrivateAccessProb*, or by increasing the percentage *SharedWriteProb* of update transactions in the shared region.

4.4 Validation of the Simulator

The simulator is implemented in C++/CSim [12], and consists of about 5000 lines of code. The behavior of the simulator without client-side caching and for read-only workloads has been validated by running experiments against a commercial relational database (Oracle 7.3.2). Query traces were generated for the workloads, and run on the database. The simulator was run in trace-driven mode, and the results of the real and simulated runs were compared. The difference in query response time was found to be with 5%, and number of disk reads to be within 10%. Details of the validation experiments can be found in [14].

5 Simulation Experiments and Results

We conducted a variety of experiments to evaluate the performance of A*Cache under different conditions. Some interesting results are reported below. Our primary performance metric is the query response time. We also tracked other important simulation output such as client cache and server buffer hit ratios, CPU utilization, number of disk reads and writes, network performance, etc., in order to understand the details of system behavior. We use these quantities to analyze and explain our results. Due to space constraints, detailed description of the behavior of each output quantity is not possible in this paper. All experiments were performed for large number of queries (5000 or more), so as to eliminate any transient effects of the warmup of client caches and the server buffer, and to ensure that we observe steady-state behavior.

5.1 Read-Only Performance Results

We discuss below the results of workloads in which there are no writes. These results best demonstrate the benefits of A*Cache, since there is no cost of cache maintenance; they also help us interpret the behavior of the system for read-write workloads.

5.1.1 Effect of Varying Cache Size

Figures 2 and 3 show the effects of varying *CacheSize* for clustered and unclustered access respectively. The *PrivateRatio* is set to 50% (5000 tuples) in these experiments, so that each of the 10 clients has 500 tuples in its private region. Thus, a cache of 100KBytes, i.e., 5% of the database, is large enough to hold the entire private region of a client in the A*Cache schemes.

For both the clustered and unclustered cases, the query response time of the A*Cache schemes drops as the cache grows larger, the gains being most significant until the cache size reaches 5% of the database, which is the size of the client-private region. The cache hit ratio (not shown here) increases with cache size, and the response improves since remote accesses are fewer. Notice that A*Cache_Opt performs consistently better than the basic A*Cache scheme, due to reduced network traffic on partial cache hits. For cache size beyond 5% of the database the response time saturates for both schemes, not showing any significant improvement for cache sizes upto 40% of the database. This behavior is expected, making the cache bigger than 5% holds data in the shared region, which is accessed only 20% of the time (*SharedAccessprob*).

Now, comparing the A*Cache schemes against the no-caching system, we see that substantial speedups are obtained for both clustered and unclustered access, with the improvement in response time in the unclustered case being an order of magnitude larger than that of clustered access. The reason for this behavior is that for unclustered disk reads by clients, the utilization of the server buffer is poor, and the benefits of local caching are enhanced.

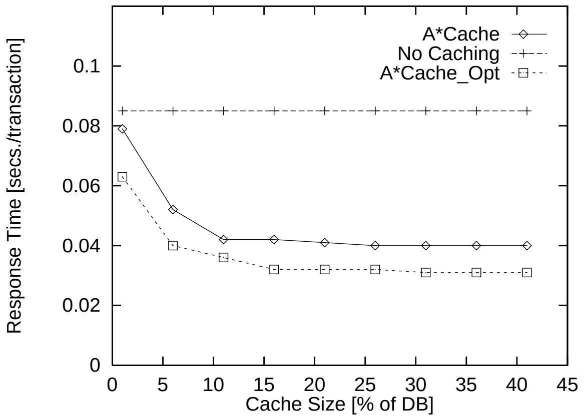

Figure 2: *Varying CacheSize, Clustered Access*

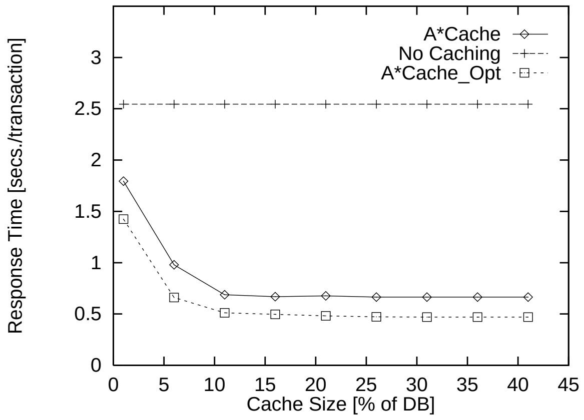

Figure 3: *Varying CacheSize, Unclustered Access*

5.1.2 Effect of Varying *PrivateAccessProb*

Figure 4 shows the result of varying *PrivateAccessProb* on the query response time. This experiment also uses a *CacheSize* of 100 Kbytes with a *PrivateRatio* of 50%, so that the private region of each client fits entirely in its cache. A rather unexpected result is obtained in this experiment — the query response time of the no-caching case remains almost invariant over the entire range 30% to 90% of *PrivateAccessProb*, while that of A*Cache falls substantially. The result can be explained as follows. For no-caching, disk pages accessed by clients are placed in the server buffer, which is 25% of the database. Since *PrivateRatio* is set to 50%, the private regions of all the clients taken together does not fit into the shared server buffer. Therefore, even though the *PrivateAccessProb* increases, the response time does not improve for the no-caching system. In contrast, for the A*Cache schemes, increasing *PrivateAccessProb* causes a direct increase in the cache hit ratio for each client, and a corresponding improvement in the query response time. A*Cache_Opt out-performs A*Cache again, because of less network data transfer on partial cache hits.

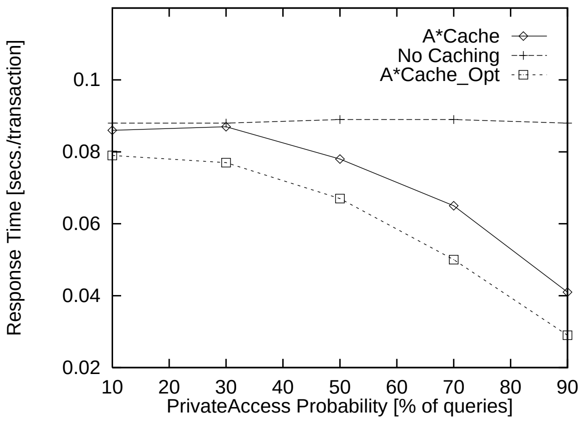

Figure 4: *Varying PrivateAccessProb*

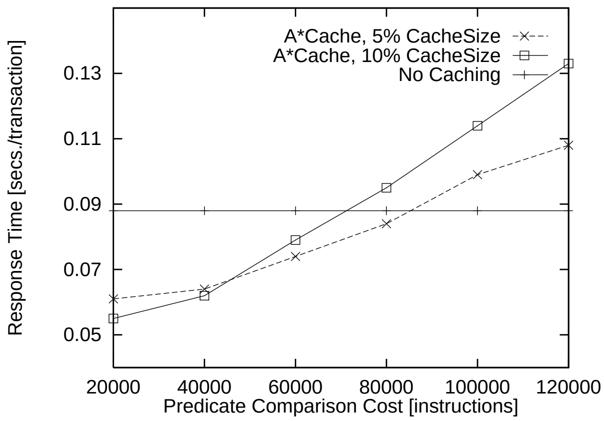

Figure 5: *Varying Predicate Comparison Cost*

5.1.3 Effect of Varying the Predicate Comparison Cost

The effect of varying *CompCost* for the basic A*Cache scheme is displayed in Figure 5. For this experiment, *PrivateRatio* is set to 50%, and two different settings, 5% and 10%, of *CacheSize* are used. For small values of *CompCost*, A*Cache performs better than no-caching. However, when *CompCost* is increased beyond 40,000 instructions, A*Cache performance starts to decrease substantially. The crossover in performance happens at about 85,000 instructions for a relative *CacheSize* of 5%, and at about 75,000 instructions for a relative *CacheSize* of 10%. In interpreting this result, it must be kept in mind that our simulator does not employ predicate indexing (Section 4.2.1), so that in the worst case (a partial or whole cache miss), all predicates are compared in checking a query against the cache for containment. Therefore, as *CacheSize* is increased, more predicates are compared and the cost of containment checking rises, causing an earlier crossover with the no-caching response. The performance of A*Cache can be significantly enhanced with predicate indexing, and the impact of the cost of predicate comparison will also be diminished in that case.

5.1.4 Summary of Read-Only Results

We summarize the results of our read-only experiments. For cache hits, A*Cache performance is independent of the data clustering on server disk, since the server is bypassed entirely. Thus, the benefits of A*Cache are better realized for unclustered workloads, since the utilization of the shared server buffer is quite sensitive to the pattern of disk reads. As expected, large cache sizes cause more cache hits and local query evaluation, and thus improve the query response. The A*Cache schemes also demonstrate other benefits expected from caching systems, such as decreased sensitivity to shared resources like the server buffer and CPU, disk speed, and network speed. Therefore, these systems scale better than the no-caching system for larger number of clients. The performance of A*Cache_Opt is consistently better than A*cache, since it minimizes network traffic on partial cache hits.

5.2 Read-Write Performance Results

Below, we report the results of our experiments with update transactions. For low-contention environments, the general trends exhibited by the read-only experiments also carry over to the read-write scenario. For example, for small values of *SharedWriteProb*, the A*Cache schemes perform better with increasing cache size, since most updates are to client-private regions and the cost of notification is small. However, higher contention increases the notification processing cost for a large cache, and can cancel the performance gains obtained from the local evaluation of larger number of queries, as shown below.

5.2.1 Effect of Varying Cache Size

Figure 6 shows the effect of varying the cache size for clustered access. In this set of experiments, the *PrivateWriteProb* is 0, and the *SharedWriteProb* is 20%. The response of both caching schemes improves at first as the cache size is increased, but then rises as the cache size grows beyond 5% of the database (size of the client-private region). However, for the unclustered case shown in Figure 7, the response time decreases consistently with increasing cache sizes. This behavior is due to the fact that for the unclustered experiments, the cost of a remote operation is higher, and the gains from query local evaluation out-weigh the cost of notification processing. It can be seen from these results that the A*Cache schemes perform substantially better than the no-caching system for both clustered and unclustered scenarios, the benefits of caching being more enhanced for unclustered access.

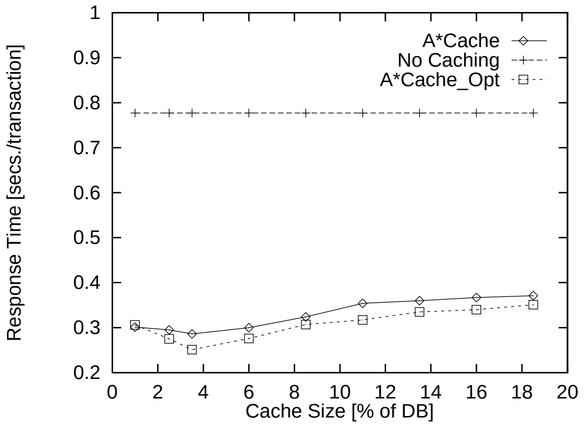

Figure 6: *Varying CacheSize, Clustered Access*

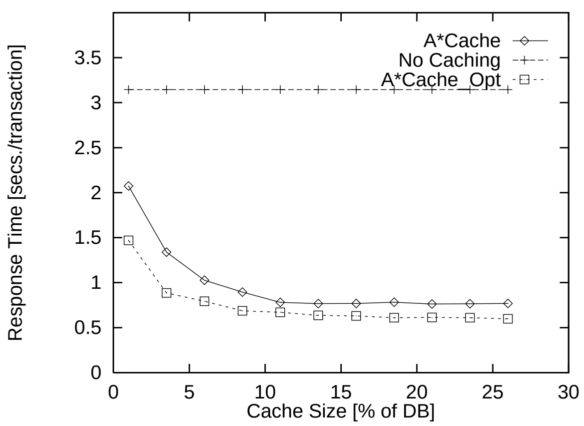

Figure 7: *Varying CacheSize, Unclustered Access*

5.2.2 Updates via Clustered Index

We now investigate the effect of varying write probabilities and PrivateAccessProb for both clustered and unclustered access by update transactions in the basic A*Cache scheme. First, we consider data access and update via the clustered index *unique2*. Figure 8 shows the effect of varying the ratio of update transactions, called *UpdateRatio* (*PrivateWriteProb* and *SharedWriteProb* jointly) over a 0% — 40% range for two different settings of the pair of parameters *PrivateRatio* and *PrivateAccessProb*.

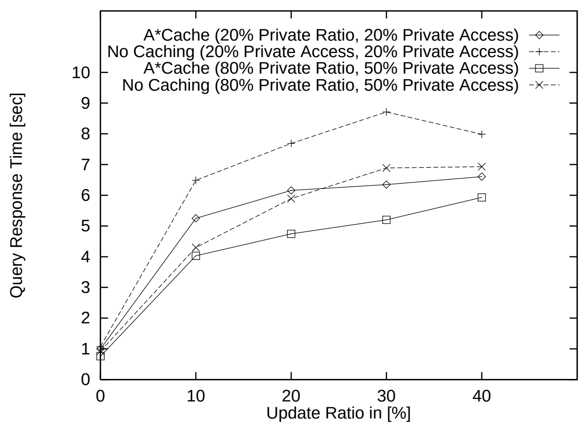

Figure 8: *Clustered Writes with Varying UpdateRatio*

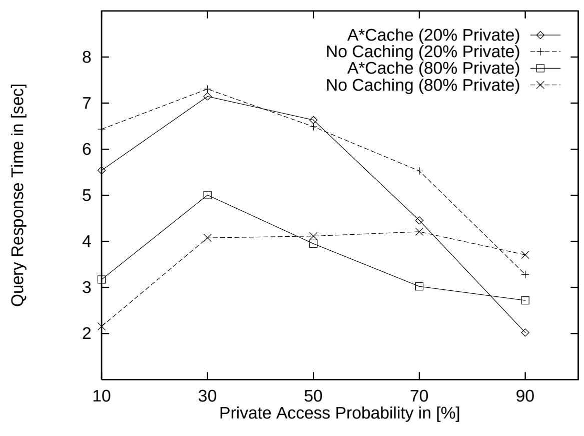

Figure 9: *Clustered Writes with Varying PrivateAccess, 40% UpdateRatio*

In one set of experiments, *PrivateRatio* was kept constant at 20% (2000 tuples), so that 200 tuples were present in the private region of each of the 10 clients. The remainder of the database with 8000 tuples was shared among all the clients. *PrivateAccessProb* was set to 20% so that most of the data accesses were directed to the shared region of the database.

In another set of experiments, *PrivateRatio* was set to 80%, so that each client had 800 tuples in its private region and shared access to the 2000 common tuples with the other 9 clients. *PrivateAccessProb* was set to 50% so that data accesses were equally divided between the shared and private regions of a client.

In both sets of experiments, query response time of the A*Cache scheme is seen to increase as the update ratio is increased from 0. This behavior is corroborated by the fact that the number of notifications increases for higher update loads, causing more conflicts and transaction aborts. In contrast, the cache hit ratio remains nearly constant over the entire range of update ratios; this behavior is explained by the fact that in our simulation, the cache is always updated, and not invalidated, upon a notification. In this case, cache hit is determined by the data access pattern and not by the amount of writes.

Now comparing A*Cache performance with no caching, the response times for A*Cache is seen as better in each of the two experiment sets. The reason for this result is that under the conditions given in each case, the cache maintenance cost for A*Cache is not large enough to offset the savings in local data reuse. It is found that the performance with no caching for 80% *PrivateRatio* and 50% *PrivateAccess* is comparable with the A*Cache scheme for 20% *PrivateRatio* and 20% *PrivateAccess*, when notification processing costs are higher for A*Cache due to frequent access to the shared region.

Figure 9 presents the results of varying the *PrivateAccessProb* over a range of 10% to 90% for a fixed 40% ratio of update transactions. The *PrivateRatio* parameter is kept constant at 20% in one set of experiments and at 80% in the other. In both cases, the query response times for the A*Cache and the no-caching schemes are found to first increase and then decrease as more accesses occur in the private regions. This behavior seems rather unexpected at first, but can be explained as follows.

For the 10% initial setting of *PrivateAccessProb*, most data accesses occur in the shared region of the database, and the server buffer is therefore effectively reused by all clients. However, as *PrivateAccessProb* is increased, more accesses occur on client-private data, so the server buffer utilization falls due to less sharing of buffer space among different clients. Although the number of conflicts and notifications also decreases with higher *PrivateAccessProb*, the drop in server buffer sharing dominates, resulting in a net effect of higher query response times over the 10% to 30% range of *PrivateAccessProb*. As the ratio of access to client-private data is increased further beyond 30%, fewer conflicts and higher cache hit ratios produce a drop in query response, with better A*Cache reuse and utilization. In the no-caching case, the data used by each client fits better into the *server* buffer, so disk traffic is reduced. This way of using the server buffer is dependent on the number of clients. The benefits are more pronounced for A*Cache, as the local cache reuse also reduces network traffic; and A*Cache is also less dependent on the number of clients (as is evident in the later discussion on Figure 12 below).

Varying the private access probability has somewhat different effects for the A*Cache and the no-caching cases. When the private access probability is low, most access is to the shared portion of the database. In the 20% private case, the shared portion is large (80%) and does not fit in the server buffer. But contention is low also, so what does fit in the A*Cache is effectively used. Hence the A*Cache does better. In the 80% private case, the shared portion is small (20%) and does fit into the server buffer. So overall performance is improved. The A*Cache performs worse because of the high contention. When private access probability is high, most access is to a different private section of the database for each client. With low contention and a local cache reducing network traffic, A*Cache performs better than no-caching. When the private access probability is very high, having the private portion be small means it to fit better into memory at the client or the server, and hence performs better than large private portions.

5.2.3 Updates via Unclustered Index

Next, we considered updates via the unclustered index *unique1* for the basic A*Cache scheme. Figure 10 shows the experiments corresponding to Figure 8 but for unclustered writes. We note that the query response times are an order of magnitude larger than those in clustered case. Unclustered disk access and poor server buffer utilization are reasons for this behavior. For A*Cache, the performance is always better than no-caching; for cache hits, the lookup is by query predicates, which is independent of the data clustering on disk. This savings in data access offsets the update notification costs.

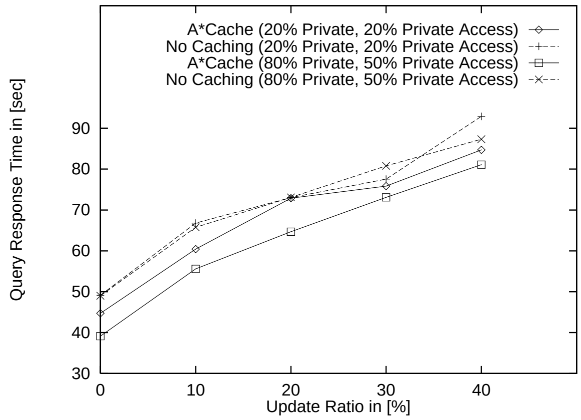

Figure 10: *Unclustered Writes with Varying UpdateRatio*

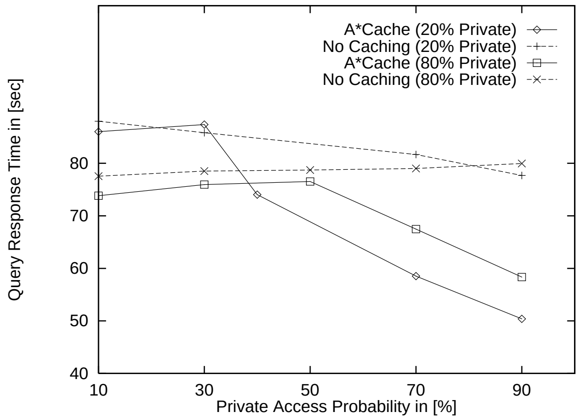

Figure 11: *Unclustered Writes with Varying PrivateAccess, 40% UpdateRatio*

Figure 11 is the counterpart of Figure 9 for unclustered updates. Again, response times are much higher compared to the clustered case, due to unclustered disk access and low server buffer reuse. Because of this, the query response time for no-caching remains relatively constant even though the *PrivateAccessProb* rise to 90%. In contrast, the performance of A*Cache improves as the *PrivateAccessProb* is increased to access more data in client-private regions, which results in a higher A*Cache hit ratio and reduces network trips to the server. For each of the two settings of *PrivateRatio*, the A*Cache scheme performs better or as well as no-caching because of effective client cache reuse.

5.3 Sensitivity Analysis

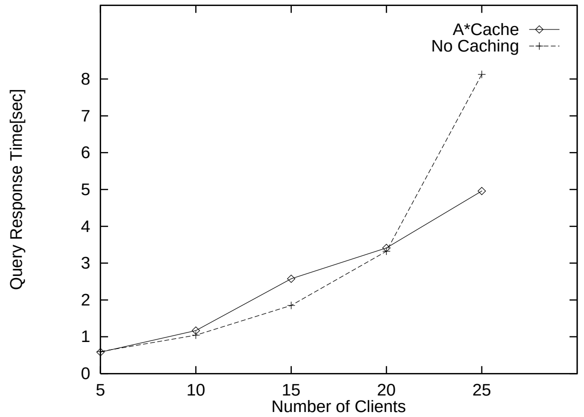

Figure 12: *Varying Number of Clients*

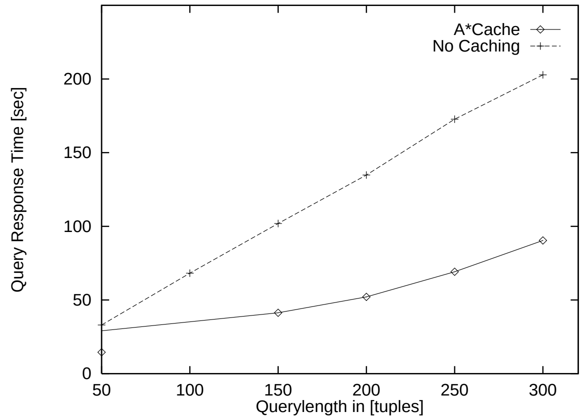

Figure 13: *Varying Query Length*

We ran numerous experiments to determine the sensitivity of the system with respect to the various simulation parameters. For example, the cache size, query length, network bandwidth, write probabilities, number of clients, etc., were varied. The results of two sensitivity experiments are presented here. To determine the scalability of A*Cache, we enlarged the database size to 100,000 tuples (20 MBytes) and increased the number of clients to 25. Figure 12 shows the A*Cache benefits gained when the system is scaled up. The number of clients is varied from 5 to 25 while other parameters, such as the buffer size at the server, remain the same. Here, 50% of the queries are within the private region, which is 80% of the database, and the *UpdateRatio* is 20%. Although A*Cache has a large number of notifications with more clients, the query response time increases almost linearly, whereas in the no-caching scheme it increases exponentially. This behavior is due to utilization of the CPU and memory of the additional clients by A*Cache. It shows that A*Cache is highly scalable with respect to the number of clients in the system.

In the second experiment, we demonstrate the sensitivity of A*Cache to the query length. The results are shown in Figure 13. We varied the query length from 50 tuples (1KByte) to 300 tuples (6KBytes). Other parameters are the same as in Figure 12. This is another example of the good scalability properties of A*Cache. Although we increase the number of notifications by enlarging the query length, A*Cache performs better than the no-caching scheme. The reason for the linear increase in the no-caching query response time is the almost linear increase in execution time at the server, which causes the server CPU and buffer utilizations to increase, but also increases the query response time. In the case of A*Cache, local data is used effectively (*CacheHitRatio is about 63%*), so that the server is not contacted very often; therefore, the query response time for A*Cache increases less than linearly with the query length.

6 Conclusions and Future Work

Associative caching is an important technique to improve the performance and scalability of a client-server database by better utilization of client CPU and memory. This paper investigates the performance of A*Cache, which is an associative caching scheme introduced in [11]. We investigate both read-only and read-write workloads, and focus on the effect of updates on cache maintenance in particular. Through detailed simulation, we investigate the behavior of the system under different contention and update loads, and compare it to a no-caching scheme. Our results demonstrate that A*Cache performance can be good even for fairly large number of updates.

Our results show that A*Cache performance exceeds that of the no-caching approach when there are no updates, unless the cost of cache containment reasoning is high. When updates are few, notification and contention are infrequent, so A*Cache still performs better. It is only when updates are high that A*Cache does not always dominate no-caching schemes. Therefore, this paper has focused on mostly on high update cases (40% of transactions perform updates). Here, A*Cache performs at least as well as no-caching except when there is very high contention on a small portion of the database by all clients. This situation happens when most of the access is to a small shared portion of the database, which fits in the server buffer. In this case, the server effectively uses its buffer for no-caching, while the need for notifications reduces performance in the A*Cache case.

An important benefit of A*Cache is that the only contention possible is on data; indexes are created independently by clients, if necessary for local query execution. A*Cache reuse and lookup is therefore also independent of the data clustering on the server disk.

Future work includes the investigation of advanced cache maintenance policies, such as invalidation of infrequently used query results and update of frequently used ones. Additional knowledge or learning ability of data usage patterns of clients is required for such intelligent maintenance. Data caching on client disks has not been examined in this paper, but is possible in the A*Cache scheme as proposed in [11]. An LRU-based predicate replacement algorithm is adopted in our simulation; examining more sophisticated cache replacement algorithms, such as extended LRU schemes, or those that consider both spatial and temporal localities of cached predicates [3], are other interesting research issues.

Limited associative access may be supported in an identity-based client cache by using indexes defined at the server database. If the query uses an indexed attribute, then the relevant indexes can be examined to determine which objects satisfy the query. However, unlike A*Cache, reference to server indexes is required, which may be a bottleneck. Comparison of the performance of such schemes with A*Cache is another interesting area to be explored.

Acknowledgements

We gratefully acknowledge the many useful discussions with Kurt Shoens of Hewlett-Packard Laboratories and Prof. Gio Wiederhold of Stanford University. Their insights have helped us greatly. Kurt Shoens also ran several experiments on a commercial database to help validate the behavior of our simulator.

References

[1] J.A. Blakeley, N. Coburn, and P.-A. Larson, "Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates," *ACM Transactions on Database Systems*, Vol. 14, No. 3, 1989, pp. 369-400.

[2] M. Carey, M.J. Franklin, and M. Zaharioudakis, "Fine-Grained Sharing in a Page Server OODBMS," *Proc. ACM SIGMOD Conf.*, Minneapolis, MI, May 1994, pp. 359-370.

[3] S. Dar, M. J. Franklin, B. Jonsson, D. Srivastava, and M. Tan, "Semantic Data Caching and Replacement," *Proc. VLDB Conf.*, Bombay, India, September, 1996.

[4] A. Delis and N. Roussopoulos, "Performance and Scalability of Client-Server Database Architectures," *Proc. VLDB Conf.*, Vancouver, B.C., Canada, 1992.

[5] D.J. DeWitt, D. Maier, P. Futtersack, and F. Velez, "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems," *Proc. VLDB Conf.*, Brisbane, Australia, 1990.

[6] M.J. Franklin, "Caching and Memory Management in Client-Server Database Systems," *Ph.D. Thesis*, Technical Report No. 1168, Computer Sciences Department, University of Wisconsin-Madison, 1993.

[7] M.J. Franklin, B.T. Jonsson, and D. Kossmann, "Performance Tradeoffs for Client-Server Query Processing," *Proc. ACM SIGMOD Conf.*, 1996.

[8] Jim Gray, "The Benchmark Handbook for Databases and Transaction Processing Systems,", Morgan-Kaufmann Publishers Inc., 1993.

[9] A. Gupta and I. S. Mumick, "Maintenance of Materialized Views: Problems, Techniques, and Applications," *IEEE Data Engineering Bulletin*, Vol. 18, No. 2, June 1995.

[10] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava, "Answering Queries Using Views," *Proc. PODS Conf.*, 1995.

[11] A.M. Keller and J. Basu, "A Predicate-based Caching Scheme for Client-Server Database Architectures," *The VLDB Journal*, Jan 1996.

[12] Mesquite Software, "C++/CSim User's Guide," Austin, Texas, USA, August 1994.

[13] Oracle Corporation, *Oracle 7 Server Concepts Manual*, Redwood Shores, California, USA, Dec 1992.

[14] M. Poess, "Simulation Analysis of SQL*Cache," Master's Thesis, University of Karlsruhe, Germany, Jan 1997.

[15] N. Roussopoulos, C.M. Chen, S. Kelly, "The ADMS Project: Views R Us," *IEEE Data Engineering Bulletin*, Vol. 18, No. 2, June 1995.

[16] P. Scheuermann, J. Shim, and R. Vingralek, "WATCHMAN: A Data Warehouse Intelligent Cache Manager," *Proc. VLDB Conf.*, Bombay, India, 1996.

[17] A.P. Sheth and A.B. O'Hare, "The Architecture of *BrAID*: A System for Bridging AI/DB Systems," *Proc. Intl. Conf. on Data Engineering*, Kobe, Japan, April 1991.

[18] Y. Wang and L.A. Rowe, "Cache Consistency and Concurrency Control in a Client-Server DBMS Architecture," *Proc. ACM SIGMOD Conf.*, Denver, Colorado, May 1991.

[19] K. Wilkinson and M.-A. Neimat, "Maintaining Consistency of Client-Cached Data," *Proc. VLDB Conf.*, Brisbane, Australia, August 1990.