Performance Analysis of Associative Caching Schemes for Client-Server Databases

Julie Basu	Meikel Pöss	Arthur M. Keller
jbasu@us.oracle.com	*mpoess@us.oracle.com*	*ark@cs.stanford.edu*
Oracle Corporation	Oracle Corporation	Stanford University

August 31, 1998

Abstract

This paper presents a detailed performance study of three associative caching schemes for client-server databases. In all the three schemes, the client cache loads query results dynamically in the course of transaction execution, and supports evaluation of associative queries on the cache, at the cost of refreshing the cached data and maintaining transaction serializability through asynchronous update notifications from the server. The schemes differ in the way they handle cache containment reasoning and data updates. In one scheme, the cache derives a description of its current contents from the stored queries, and uses predicate-based reasoning to examine and maintain the cache. The second scheme is an optimized version of the first, and considers partial containment of cached queries. It also performs writes at the central server in case of a cache miss. The third caching scheme operates in terms of tuples only, requiring trips to the server to determine cache containment of associative queries. We compare through detailed simulation the relative performance of these three caching schemes under different workloads and update contention profiles for multiple clients, and contrast their behavior with that of a no-caching system. Our simulation results clearly demonstrate the effectiveness of associative caching for read-only environments, and also for read-write workloads with moderately high data update probabilities.

1 Introduction

1.1 Caching in Client-Server Databases

A popular architecture for modern database management systems is the client-server configuration. This setup involves one or more server processes that manage a shared repository of persistent data, and handle requests for data retrieval and update from multiple clients. 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 centralized concurrency control, transaction management, and recovery facilities for the shared database.

Response of the shared server is a critical factor in the performance of a client-server system. The server can become the bottleneck in scaling the system to large amounts of data or increasing number of clients. Today, the revolution in computer hardware technology has made client resources relatively cheap and plentiful. 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. Despite the potential cost of maintaining data cached at client sites, a number of recent studies have demonstrated that inter-transaction reuse of cached data can decrease network traffic and query response times [8, 18, 19].

1.2 *Associative* Versus *Navigational* Caching

Most object-oriented databases assume smart clients, and are built using a *data-shipping* approach. In these systems, the client cache is 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 [5, 8, 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.

1.3 An Example of Associative Caching

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. Queries to the database typically retrieve a set of bugs satisfying an associative condition such as (*Product# = 100 AND Status='Open'*).

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. Also, the data is being accessed associatively, e.g., based on the product information and the status of bugs. Thus, for effective data reuse at the client, navigational retrieval is not adequate, and associative access to cached data is required.

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, provided that transaction serializability is still retained through the use of an appropriate concurrency control protocol.

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 with the cache description P above, there are three possible ways to reflect the update on its cache:

- 1. The tuple can be deleted, since the updated value no longer satisfies predicate P.
- 2. The tuple can be updated and retained in the cache, but since it does not satisfy predicate P any longer, the client must also augment its cache description to include the new tuple.
- 3. The predicate P can be purged from the cache description, 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 cached is an appropriate maintenance action that preserves the validity of cached results.

Handling updates is thus more complex for a predicate-based cache compared to an identity-based one, since it is not sufficient to consider data items individually. Instead, containment 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 [10] are applicable in this context.

1.4 Outline of Paper

This paper investigates through simulation the performance of three associative caching schemes for client-server databases. In all the three systems, the client cache loads query results dynamically in the course of transaction execution, and supports evaluation of associative queries on the cache. The benefits are offset by the cost of refreshing the cached data in the presence of updates, which is done through asynchronous update notifications from the server. The schemes differ in the way they handle cache containment reasoning and data updates, resulting in different performance characteristics. The simulation model is based on queueing networks, and takes into account the client-side caching facilities with the associated retrieval and maintenance costs, as well as the cost of notification generation by the server. Our workload model is based on the standard Wisconsin benchmark [9] for a single user, extended with locality and contention models for multiple clients [5]. We report the results of simulation experiments performed for a variety of workloads with different data usage and contention patterns, and analyze the effectiveness of associative caching in both read-only as well as read-write scenarios. Our results demonstrate that associative caching can provide substantial performance benefits for read-only environments, and even under moderately high update activity.

The rest of this paper is organized as follows. In section 2, we describe the three caching algorithms. The simulation environment with the system and workload models is described in Section 3. In Section 4, we report the results of our simulation experiments, and analyze the observed behavior. Section 5 reviews related work. Finally, we summarize our conclusions in Section 6.

2 Three Associative Caching Schemes

We describe the three caching schemes investigated in this paper. One of the schemes, now called A*Cache, was originally proposed by us in [13] and [12]. In this scheme, the cache derives a description of its current contents from the stored queries, and uses predicate-based reasoning to examine and maintain the cache. The second scheme, named A*Cache_Opt, is an optimized version of the first, and considers partial containment of cached queries. It also performs writes at the central server in case of a cache miss. The third caching scheme is named Tuple-Cache, and it operates in terms of tuples only, requiring trips to the server to determine cache containment of associative queries.

2.1 A*Cache Algorithms

We briefly review below the scheme presented in [12]. 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. If all the required data is not available locally, a request is submitted to the server to execute the query and return the results. A remote request is accompanied by any local (uncommitted) updates of which the server has not yet been informed, in accordance with the standard SQL isolation policy where a transaction is able to see the effect of its own updates [1], for all local as well as remote operations.

2.1.1 Cache Consistency in A*Cache

An important concern is consistency maintenance of the cached data, which may become out-of-date as updates are committed on the central database at the server. The A*Cache scheme uses a notification mechanism to maintain the validity of cached data, as described in [12]. Each client registers a *subscription* with the server that describes the query results cached. Using these subscriptions and incremental view maintenance techniques, the server generates notifications for a client whenever an update is committed at the central database. Based on the maintenance policy adopted at the client, 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 if data is purged from or loaded into the cache.

2.1.2 Concurrency Control in A*Cache

In A*Cache, update notifications from the server are also used for concurrency control purposes. For a cache hit, A*Cache allows the data to be updated locally. On a cache miss, the data is fetched in from the server, and then updated locally. All local updates performed previously at the client are flushed to the server at each server operation and at the final commit, so that the uncommitted updates are visible for remote operations. In order to ensure serializability, the server must verify that the client has seen its most recent notification message before confirming the commit, and posting the deferred updates to the database. This checking is done by assigning a sequential message number to every notification. A notification received at a client aborts a transaction when a conflict with is detected in its local reads or writes with the updates committed at the database. This concurrency control protocol is *semi-optimistic* in nature, permitting local updates but enforcing serializability at commit through sequenced notifications.

2.2 An Optimized Extension of A*Cache

The second scheme, which we call *A*Cache_Opt*, utilizes a cache miss as an opportunity to optimize network traffic and also to reduce data update conflicts among different clients. It introduces the following two optimizations in the basic A*Cache model:

- An optimization for queries to utilize partial cache hits. A*Cache_Opt detects partial containment in the client cache, and fetches only the missing tuples from the server. Considering partial cache hits can reduce the data traffic across the network compared to the basic A*Cache scheme.
- An optimization on a cache miss for update operations. A cache hit is handled in the same way as in basic A*Cache, with updates being performed locally and flushed to server with the next remote request. However, a cache miss or a partial cache hit in A*Cache_Opt updates tuples 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, even if some are already present in the cache, and performs the requested update locally, subsequently flushing these updates to the server. This optimization in A*Cache_Opt minimizes the chances of a transaction being aborted by update notification from the server, since remote updates at the server acquire long-duration write locks for the modified tuples. A*Cache_Opt is therefore less optimistic about data writes than basic A*Cache, and also compared to the Tuple-Cache scheme described below.

2.3 A Tuple-Based Caching Scheme

In this caching scheme, called *Tuple-Cache*, it is assumed that the client cache stores only tuples and does not maintain any associated query predicates. Therefore, whether the result of an associative query is contained in the cache cannot be determined locally, and the server must be contacted for each query to fetch the zero or more missing tuples in its result. Consider, for our BUG database example introduced in Section 1, that the client executes the following query:

```
SELECT *  FROM BUG  WHERE Product# = 100;
```

Execution of the above query will cause all the bugs for Product 100 to be placed in the client cache. However, if the same query or a query that defines a subset of the original query, such as

```
SELECT  *  FROM BUG  WHERE Product# = 100  AND  Status = 'Open';
```

is submitted again, then Tuple-Cache cannot determine locally that it has the entire query result. Unlike A*Cache, it does not store the query predicate ($Product\# = 100$) describing the tuples, and thus cannot perform cache containment reasoning locally. Only the server can determine which tuples belong in the query result, and send the set of tuples that are missing from the cache. This lookup of missing tuples can be done efficiently only if the queries use indexed attributes, as is the case in our simulation. For such queries, the server can use its index pages to determine the set of tuples satisfying a query, thereby avoiding full-fledged query execution on the database. Only those result tuples that are missing from the cache can be retrieved from the database and returned to the client. In this sense, the Tuple-Cache scheme is related to index-based associative caching systems [3, 20], with the difference that in Tuple-Cache, indexes are located and maintained centrally at the server and are not cached by clients. Thus, the issue of distributed caching and maintenance of index pages at client sites does not arise for Tuple-Cache.

2.3.1 Cache Consistency in Tuple-Cache

The client in a Tuple-Cache scheme must maintain the cached tuples to reflect database updates. As in A*Cache, the server sends update notifications for tuples cached by the client, which are used to refresh the locally cached tuples. Notification processing in Tuple-Cache is simpler compared to A*Cache, since old copies of the modified tuples are directly replaced with their new images, and there are no cached predicates to be analyzed and maintained by the client.

2.4 Concurrency Control in Tuple-Cache

In Tuple-Cache, updates are made locally at a client after fetching the tuples missing from its cache. We choose this update scheme for Tuple-Cache, since it is comparable to the update-handling mechanism in the basic A*Cache model. To guarantee transaction serializability in the tuple-caching environment, the server must still ensure that tuples within the scope of the query predicates of a currently-running client transaction are not updated by another transaction. In Tuple-Cache, we have adopted a semi-optimistic concurrency control scheme with a commit verification step to detect conflicts, which is similar to the policy chosen for A*Cache. This scheme is described below.

The server obtains short-duration read locks as it retrieves the tuples for a query or an update. These locks are released after retrieval of the tuples, but the access predicate (or index range lock) for the command is recorded by the server. Tuples that are in the result set but are missing from the cache are then sent to the client. If another transaction originating from a different client now commits updates, the server checks for serializability violations using a commit verification scheme as in A*Cache. Upon each successful commit, the server sends to a client all modified tuples that are either stored in the client cache, or that affect the transaction currently running at the client. The latter condition is detected using the access predicates recorded at the server for in-flight transactions. As in A*Cache, this concurrency control scheme is semi-optimistic in nature, with a transaction being possibly aborted upon notification of a conflicting update.

2.5 Comparison of Caching Schemes

2.5.1 Comparison of Query Processing Strategies

Each of the three schemes, Tuple-Cache, A*Cache, and A*Cache_Opt, processes queries quite differently. Below, we summarize these differences:

- Tuple-Cache stores tuples at the client, but it does not maintain any predicate-based cache descriptions. Therefore, a client cannot determine locally if the result of an associative query is available in the cache, and must request the server for missing tuples, if any. The cost of a network trip to the server is incurred by every query, with only those tuples that are missing from the cache being sent by the server as the query result.
- The basic A*Cache scheme uses its predicate-based cache description to determine cache hits at the level of entire queries. For local hits, a round-trip to the server is obviated, thereby saving network traffic and server processing costs. However, partial hits to the cache are not considered in the basic A*Cache scheme. The server is requested for the entire query result, causing higher data traffic on the network compared to Tuple-Cache.
- A*Cache_Opt, like the basic A*Cache scheme, also has the ability to use its predicate descriptions for determining cache hits of associative queries, thus avoiding remote trips to the server. Additionally, the A*Cache_Opt scheme takes advantage of partial cache hits to request the server for only those tuples that are missing from its cache. Thus, it incorporates the same optimization for fetching remote data as in Tuple-Cache, saving disk reads and network traffic, and reusing cached tuples more effectively.

The query processing characteristics of the caching schemes are summarized in Table 1.

Query Processing Characteristic	**Tuple-Cache**	**A*Cache**	**A*Cache_Opt**
Tuples cached at client	$\surd$	$\surd$	$\surd$
Local containment check of assocative queries		$\surd$	$\surd$
Fetch missing tuples only	$\surd$		$\surd$
Predicate descriptions at the clients and at server		$\surd$	$\surd$
List of cached tuples at the clients and at server	$\surd$		
Need server indexes for cache containment check	$\surd$		

Table 1: *Comparison of Query Processing Methods*

2.5.2 Comparison of Data Update Strategies

Below, we provide a summary of the differences in the update strategies for Tuple-Cache, A*Cache, and A*Cache_Opt. With respect to the steps taken by a client on a request for data update, the following differences exist among the three systems:

- In the basic A*Cache scheme, a cache miss causes a request to be sent to the server, to fetch all the rows in its result set. Partial cache hits are handled in the same way as cache misses. Upon return from the server on a cache miss, or after a cache hit, the client performs updates locally. These updates are up-loaded to the server at the next trip.
- Tuple-Cache follows a procedure similar to basic A*Cache. It first fetches zero or more missing tuples required for the update from the server, and then makes local updates that are sent to the server with the next request. The initial round-trip to the server to detect missing tuples is always necessary for associative queries, since Tuple-Cache does not store predicate descriptions of its cache contents.
- A*Cache_Opt handles cache hits and misses quite differently. A cache hit is handled in the same way as in basic A*Cache, with the client performing the updates locally, but a cache miss or a partial cache hit causes a round-trip to the server, which is also used to perform the updates centrally. All updated tuples, as well any unmodified tuples that satisfy the access predicate but are missing from the client cache, are returned as the result.

Handling of update notifications is another area where the schemes differ substantially. We list below the specific actions taken by the server and the clients with respect to update notifications.

- Both A*Cache and A*Cache_Opt describe and maintain their caches in terms of query predicates. In effect, this feature requires the maintenance of the cached materialized views. Update notification messages are generated by the server to inform the clients of modifications to the database, and of potential conflicts with currently-executing transactions. Registered client subscriptions, that are based on query predicates, are used by the server to detect possible overlap of updated regions with cached predicates, and to filter out updates that are irrelevant for a client. This filtering is at the level of *predicates*, and is not based on individual tuples. Thus, it is possible that some modified tuples sent in a notification message are not pertinent for the client cache; such tuples are discarded at the client upon receipt. The predicate-level filtering of updates can produce increased data volumes for A*Cache updates, since the server notification is liberal in terms of tuples.

  Clients in the A*Cache and A*Cache_Opt schemes process notification messages received from the server as follows. If the update is no longer relevant for the client cache (e.g., due to purging of cached predicates for space reclamation), and also does not affect the currently-running transaction, then the message is ignored. Otherwise, the relevant cached tuples, and possibly also the cached predicates, are updated to reflect the changes, with the current transaction being aborted on a conflict.

- Tuple-Cache also uses a notification mechanism to refresh its cache. In contrast to both of the A*Cache schemes, the notification in Tuple-Cache is based on *tuples*, and not on cached predicates. The server maintains a list of tuple identifiers for the data stored at each client. Whenever a transaction commits updates at the server, the set of tuples it modified is compared against the list of tuples cached by each client. If a cached tuple was updated, or the modifications affect the set of tuples read or written by a currently-running client transaction, then the new values of the relevant tuples are delivered in a notification message to the client. The client refreshes its cached tuples using these new values, and aborts the current transaction in case of a conflicting update that violates serializability.

The distinguishing characteristics of update handling methods in the four types of caching systems are summarized in Table 2 below.

Update Processing Characteristic	**Tuple-Cache**	**A*Cache**	**A*Cache_Opt**
Local update at client on cache hit	$\surd$	$\surd$	$\surd$
Remote fetch + local update on cache miss	$\surd$	$\surd$	
Remote update at server on cache miss			$\surd$
Update notification from the server	$\surd$	$\surd$	$\surd$
Tuple-level filtering of updates by server	$\surd$		
Maintain predicates		$\surd$	$\surd$
Refresh updated tuples	$\surd$	$\surd$	$\surd$

Table 2: *Comparison of Update Handling Methods*

3 Simulation Environment

We now describe the simulation system that we developed to evaluate the performance of A*Cache, A*Cache_Opt, and Tuple-Cache schemes. First, we list the caching policies and assumptions for the different caching schemes, and then discuss the simulation parameters and workloads.

3.1 Caching Policies for Simulation

Following are the caching policies and assumptions that are adopted in our simulation environment:

- The client stores data in main memory and client-side disk caching is not considered in our study. Presence of disks at clients will affect the local data capacity and response time, but is not expected to alter the relative performance of the different data caching schemes. Additionally, our simulation does not consider local indexes on cached data.
- The server always assumes that the client will cache the result of a remote query. Thus, for A*Cache and A*Cache_Opt schemes, a query predicate is entered into a client subscription at the server before the result is sent back. For Tuple-Cache, the server maintains a list of unique identifiers for the cached tuples client, and updates this list as tuples are sent to the clients. In all the three schemes, the result tuples from a remote query are always cached at the client. Cache descriptions in the A*Cache and A*Cache_Opt schemes, and the list of cached tuples in Tuple-Cache are appropriately updated whenever a new query result is cached.
- For A*Cache and A*Cache_Opt schemes, the 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.
- Cache replacement in all three schemes follows an LRU policy. For A*Cache and A*Cache_Opt, the space management is at the level of predicates with reference counts for tuples, so that a tuple is purged only when it has no predicates referring to it. For Tuple-Cache, space is reclaimed using an LRU scheme at the level of tuples.
- Update notifications always refresh the cache. Examining the performance of other maintenance schemes, such as invalidation, is the subject of future work.
- For the updates in our simulation workloads (Section 3.3), modified attributes do not affect the predicates used to access the data. Hence, only those tuples affected by a notification are updated in A*Cache and A*Cache_Opt; previously-cached predicates do not need to be altered.
- Transactions are always aborted if a conflict is detected with concurrent updates. Thus, full serializability of transactions is enforced through notification message numbers, as described in Section 2.

It must be emphasized that these policies are a feature of our simulator, and not of the caching schemes in general. The above choices were made for simplicity of simulation logic and easy interpretation of experimental results. The policies are generally biased *against* the caching schemes, in that their performance is expected to be better if an alternate scheme is chosen, for example, if the cached predicates are indexed. Exploring alternate cache management policies, as well as extensions such as client disks, is the subject of future work.

3.2 Simulation Model

We have constructed a detailed simulation model for the caching schemes described 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 *PerPageInst*.

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 for comparing two predicates for overlap or containment is *CompCost*.

Table 3 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.

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

Parameter	Description	Usage	Default Value
ServerCPU	Server CPU speed in MIPS	Variable	50
LockInst	Instructions to get/release a lock	Fixed	1000
DeadlockInterval	Deadlock detection interval	Fixed	5
BufferSize	Server disk buffer size as % of database	Variable	25%
PageSize	Size of a data page in bytes	Fixed	4096
DiskSpeed	Server disk speed in msecs.	Fixed	6.5
DiskInst	Instructions to read/write a page	Fixed	5000
NetBw	Network bandwidth in Mbit/sec	Variable	5000
MsgInst	Instructions to send/receive a message	Fixed	20,000
PerPageInst	Instructions to send/receive a page	Fixed	12,000
NumClients	Number of clients	Variable	10
ClientCPU	Client CPU speed in MIPS	Fixed	50
CacheSize	Client Cache size as % of database	Variable	0—40%
CompCost	Instructions to compare 2 predicates	Variable	1000

Table 3: *System Parameters*

3.3 Workload Model

Table 4 shows our workload parameters. We consider a single relation with 10,000 tuples of 200 bytes each in a Wisconsin-style setting [9]. In order to make a large set of experiments feasible, we kept the database small (2 Mbytes) and scaled the client cache and the database buffer sizes proportionately. The relation has two indexed attributes *Unique1* and *Unique2* that are unclustered and clustered respectively.

Transactions consist of a single associative query or update. Each query is a linear range selection either on *Unique1* or on *Unique2*, and its default length in terms of the number of tuples retrieved is 0.1% of the database size. The predicates in a cache description are linear intervals corresponding to query ranges. Each update transaction reads all tuples that it accesses, and modifies half of them randomly. Writes do not modify the indexed attributes *Unique1* and *Unique2*, hence no index updates are involved.

The Wisconsin setup is extended with locality parameters [8] 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:

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

Parameter	Description	Value
DBsize	Database size in tuples	10,000
TupleSize	Tuple size in bytes	200
QueryLength	Query length as % of the database size	0.1—1%
PrivateRatio	Private region as % of database size	50%
PrivateAccessProb	Access probability of the private region	20%—80%
SharedAccessProb	Access probability of the shared part (= 1 − *PrivateAccessProb*)	80%—20%
PrivateWriteProb	Percentage of update transactions in the private region	0%—40%
SharedWriteProb	Percentage of update transactions in the shared region	0%—40%

Table 4: *Workload Parameters*

In the default setting of our experiments, PrivateRatio is set to 50%, so that a cache that is 5% of the database holds the entire private region for each of the 10 clients. For the private and shared regions of the database, we define a data access probability (*PrivateAccessProb* and *SharedAccessProb* respectively) and an associated probability (*PrivateWriteProb* and *SharedWriteProb*) for update transactions in that region. These quantities are the same for every client. Contention of shared data can be controlled either by decreasing the *PrivateRatio*, or by increasing the percentage *SharedWriteProb* of update transactions in the shared region.

3.4 Validation of the Simulator

The simulator is implemented in C++/CSim [14], 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). Details of the validation experiments can be found in [15]. 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 average difference in query response time between simulated and actual executions was found to be with 5%, and number of disk reads were within 10%, which is considered close enough for our purposes.

4 Simulation Results

We present some interesting results from a variety of experiments performed for many different workloads. Our primary performance metric is query response time. Other important simulation output such as client cache and server buffer hit ratios, CPU utilization, number of disk reads and writes, network performance, etc., were also tracked in order to understand the details of system behavior [2]. 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), so as to eliminate any transient effects of the warmup of client caches and the server buffer, and obtain steady-state behavior.

4.1 Read-Only Performance Results

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

4.1.1 Effect of Varying CacheSize

Query response times for both clustered and unclustered access are reported in Figure 1, with *PrivateAccessProb* being 80%, and the size of the client cache varying from 0 to 40% of the database size. Unclustered access causes many more data pages to be accessed for a given query, diminishing the effectiveness of the server buffer and causing more disk reads compared to the clustered case. The query response times are correspondingly an order of magnitude larger for unclustered access, and the benefits of client-side caching are more prominent.

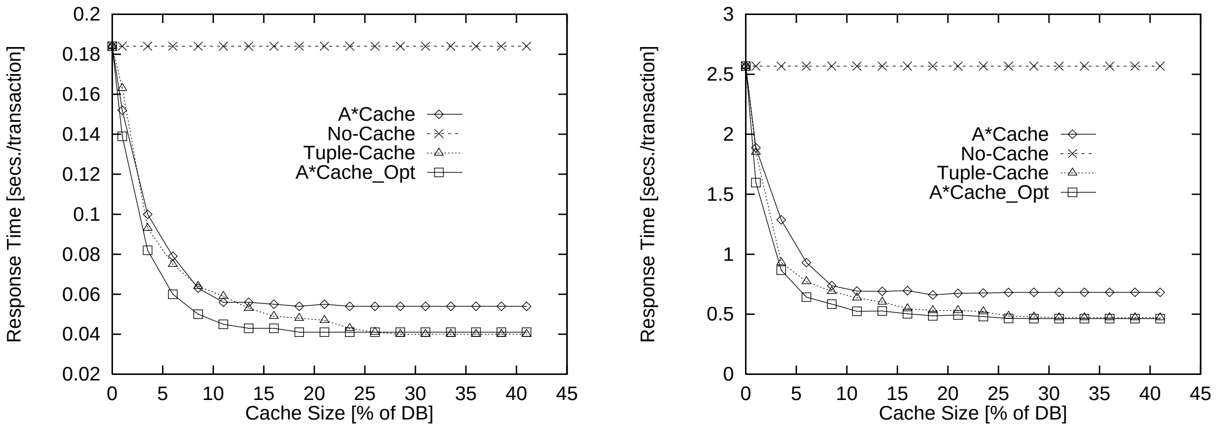

Figure 1: *Varying CacheSize, Read-Only Clustered and Unclustered Access*

We observe in Figure 1 that all three caching schemes show a large decrease in the 0% to 5% region of the cache size, with a smaller drop in the 5% — 10% region. The improvement saturates for cache sizes 15% and larger, the response time of all the caching schemes being over 60% better than the No-Cache case. Recall that for our experiments, a cache size of 5% is large enough to hold the entire private region of a client, to which 80% of all accesses by the client are directed. Therefore, cache sizes smaller than 5% cause remote trips and significant deterioration in the response time. Increasing the cache size to 5% produces rapid improvement due to higher cache hit ratios. A cache size larger than 5% enables the client to store shared data in addition to its private data, further reducing the number of remote trips, disk accesses, network volume, and response times. However, beyond a cache size of 10%, the response time saturates for all three caching schemes, showing no significant improvement with larger caches (up to 40% of the database size). This asymptotic behavior is a result of our experimental environment, in which a cache larger than 5% holds data from the shared region, which is accessed only with 20% probability. Therefore, the cache hit ratio increases only marginally with caches larger than 10

In terms of the other output parameters (not shown here), cache sizes beyond 10% result in at least 75% less disk reads than the No-Cache case, with the corresponding network volume being about 70% less. The fewer disk reads are due to less round-trips to the server, indicating effective use of the client-side data. The significant reduction in network traffic is expected in the read-only scenario, since there is no network cost for cache maintenance and update notification.

Comparing the relative performance of the three caching schemes, we find that for our experimental environment, query response time is the smallest for the A*Cache_Opt method throughout the entire range of cache sizes. Performance of basic A*Cache is about the same as Tuple-Cache for cache sizes of 0 through 10%, beyond which Tuple-Cache starts to perform better than A*Cache, and finally does as well as A*Cache_Opt for cache sizes 25% and larger. This behavior can be explained with disk and network usage analysis: A*Cache_Opt has the lowest number of disk reads and network traffic throughout the experimental range of cache sizes, while Tuple-Cache and A*Cache have about the same values for these parameters between 0 and 10% cache size. The disk reads in Tuple-Cache decrease with increasing cache sizes, being about the same as A*Cache_Opt for cache sizes 25% and larger. The reason for this disk access pattern is that Tuple-Cache must reference server index pages for each query, thereby having more index pages in the server buffer than either of the A*Cache schemes. Therefore, the effective size of the server buffer is reduced for Tuple-Cache, and fewer data pages are stored in it; this cost offsets the savings in Tuple-Cache from fetching missing tuples only.

4.1.2 Effect of Varying PrivateAccessProb

In the set of experiments for Figure 2, the probability PrivateAccessProb of reading data in the private region is varied for all the clients from 0% to 100%. Access frequency of the shared region changes accordingly. The cache size is set to its default value of 5% of the database, and PrivateRatio is 50%.

In the clustered setting, the query response times schemes are found to first increase, reach a peak around 30% PrivateAccessProb, and then decrease as more accesses occur in the client-private regions. This interesting response pattern seems rather unexpected, but is due to the nature of the

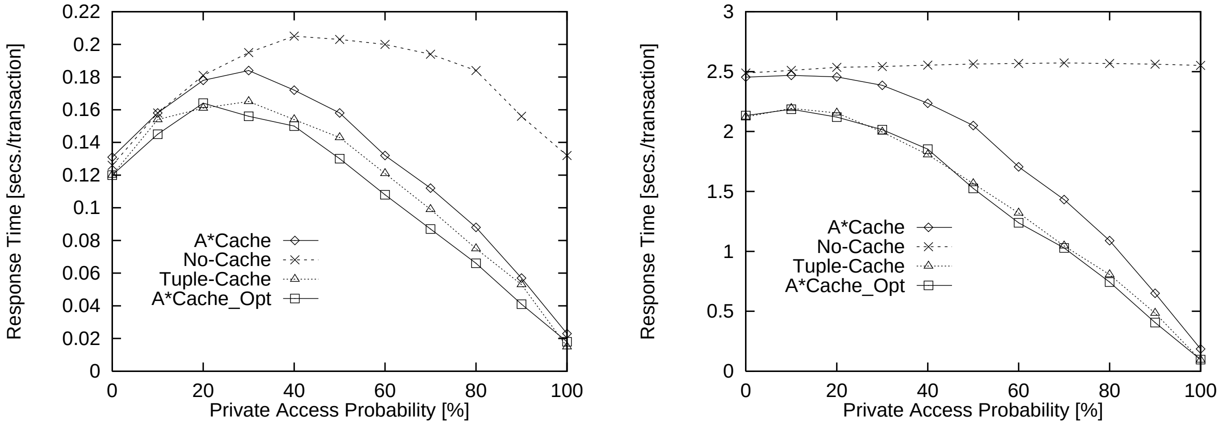

Figure 2: *Varying PrivateAccessProb, Read-Only Clustered and Unclustered Access*

Private workload model, and can be explained as follows.

For PrivateAccessProb values below 10%, most data accesses occur in the shared region of the database, which is 50% of the database. For these experiments, the server buffer is 25% of the database, i.e., half of the shared region. Therefore, the buffer hit ratio is high for small values of PrivateAccessProb, the buffered pages being effectively shared by all clients. The shared region is much larger than the client cache size of 5%. Hence, the local caches are not very effective, with the result that the response times of the caching schemes are nearly the same as the No-Cache case for PrivateAccessProb values up to 20%. The caching schemes start to perform better than No-Cache beyond 20% PrivateAccessProb, since data cached locally is better reused.

As PrivateAccessProb is increased beyond 20%, more reads occur on client-private data, so the server buffer utilization falls due to reduced sharing of buffer space among different clients. This drop in the buffer hit ratio initially results in larger number of disk reads and higher response times. When the ratio of access to client-private non-shared data is increased further beyond 30%, data in the client caches is reused more often, producing higher cache hit ratios, less network traffic, and fewer disk reads, and a corresponding drop in query response. In the No-Cache case, data used by each client fits better into the *server* buffer, so disk traffic is reduced, although the improvement is not as significant as in the caching schemes. Unlike local caches, this way of using the server buffer is dependent on the number of clients.

With regard to the relative performance of the three caching schemes, A*Cache_Opt performs the best in terms of query response, disk reads, and network traffic, followed closely by Tuple-Cache, and then A*Cache. Network volume and server CPU utilization of A*Cache is higher compared to the other two caching schemes, because on a cache miss, it fetches all result tuples in a query, even though some of the tuples might already be present in the cache.

Now considering the unclustered case in Figure 2, comparison with the clustered results reveals interesting differences in behavior, as discussed below:

- Query response times are an order of magnitude larger than those in the clustered case. Unclustered disk access and poor server buffer utilization are reasons for this behavior.
- The response of No-Cache is almost constant over the entire range 0 — 100% of PrivateAccessProb values. This behavior is also reflected in the disk reads, and is a result of the unclustered mode of data access. Compared to the clustered case, many more disk pages are accessed for each query involving the unclustered attribute. Thus, the server buffer cannot be effectively utilized.
- All three caching schemes demonstrate significant improvement in performance over the No-Cache case, the benefits increasing with larger values of PrivateAccessProb. For example, for 75% PrivateAccessProb the response time for A*Cache is 50% smaller than No-Cache, while A*Cache_Opt and Tuple-Cache are about 20% better than A*Cache. The benefits of local caching are better realized for the unclustered case, since data lookup in the local stores is independent of the data clustering on disk.
- The 'peak' effect on the response time and disk reads observed for the clustered case is missing here. This difference is due to the effect of unclustered access on the server buffer hit ratio. Queries on unclustered attributes examine many more disk pages than clustered, and so the server buffer is much less effective than for clustered queries. The peak in the clustered case is primarily caused by a drop in the server buffer reuse as access to the shared region is decreased, causing more disk reads. However, in the unclustered case, the buffer hit ratio is low (about 22%) to start with, and does not fall appreciably as PrivateAccessProb is increased. Instead, the number of disk reads falls steadily in the caching schemes, as data in the local caches is reused more often, reducing the dependency on the server buffer.

4.1.3 Summary of Read-Only Results

We have examined the behavior of our schemes in the read-only scenario, which inherently favors client caching, since there are no cache maintenance costs either at the shared server and on the network. Our results show that A*Cache_Opt performs the best for most read-only environments, followed by Tuple-Cache, and then basic A*Cache. For cache hits, response times are independent of the data clustering on server disk. Thus, the benefits of caching are better realized for unclustered workloads, since the utilization of the shared server buffer is sensitive to the pattern of disk reads. As expected, large cache sizes cause more hits and local query evaluation, and thus improve caching performance. In some cases, e.g., for large cache sizes in Figure 1, Tuple-Cache performance is comparable to that of A*Cache_Opt. There are many predicates in the cache description of a large cache, so that containment checking at the client as well as predicate management at the server consume more CPU cycles, offsetting the cost of the extra network trip in Tuple-Cache. Likewise, if the cost of predicate comparison is increased, then local containment checks and maintenance of cache descriptions are more expensive in the A*Cache schemes, degrading their performance.

4.2 Read-Write Performance Results

We report below the results of experiments with update transactions. The general trends exhibited by the read-only experiments carry over to the read-write scenario. For example, the caching schemes perform better for unclustered access and large caches, since the server buffer is bypassed. However, a large cache also increases the notification processing cost. Therefore, in the read-write scenario, the benefits of caching are maximized when *CacheSize* equals the size of the client-private region and *PrivateAccessProb* is high, so that update contention and cache maintenance costs are minimized.

4.2.1 Varying CacheSize with 20% Update Transactions

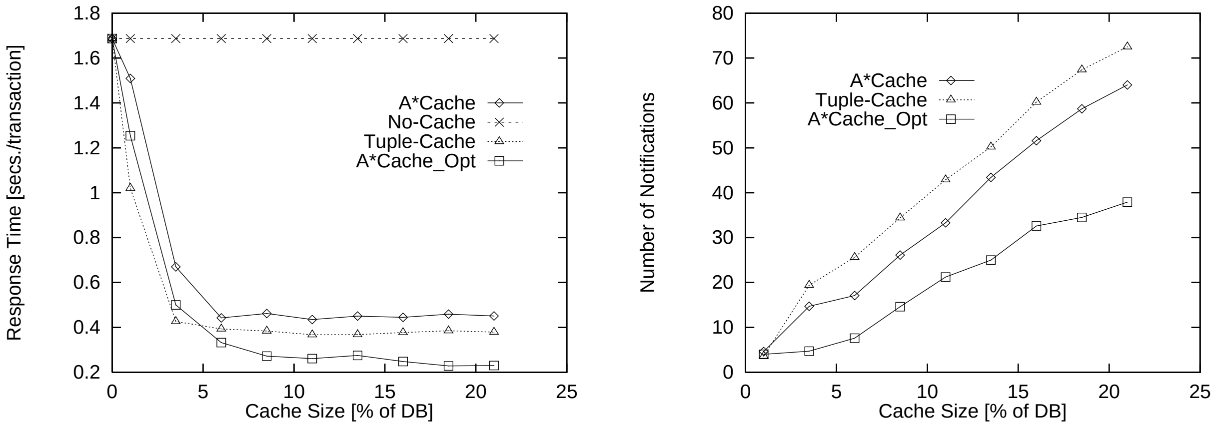

Figure 3: *Varying CacheSize, 20% Update Transactions, Clustered Access*

Figure 3 shows the response time and number of notifications with varying cache size for clustered access and 20% update transactions in both the private and shared regions. From the figure, we see that for cache sizes larger than 5%, A*Cache_Opt performs better than all the other schemes. As the cache size is increased, more shared data is stored at a client, which is only accessed 20% of the time for these experiments. This fact, coupled with the increased cost of update notification for shared data in large caches, offsets the savings from local data reuse, and the performance improvement essentially saturates at a cache size of 15%.

Another observation that can be made from Figure 3 is that the number of notifications of the three caching schemes rise with increasing cache sizes, Tuple-Cache having the highest number of notifications, followed by A*Cache, and then A*Cache_Opt. The reason for this difference lies in the different update handling policies followed for the three different schemes. As discussed in Section 2.5, Tuple-Cache fetches only those tuples from the server that are missing from its cache. With respect to concurrency control, this behavior implies that read locks are not obtained for tuples residing in the cache. In contrast, basic A*Cache fetches all missing tuples in a query that is not completely contained in the cache, increasing the network data volume and at the same time, locking the tuples at the server. Acquiring such remote locks on tuples serves to reduce conflicting concurrent updates by different clients, which would otherwise result in transaction aborts upon notification. For A*Cache_Opt, updates are performed entirely at the remote server in the case of a cache miss or a partial cache hit, thereby also acquiring the necessary locks and reducing conflicts. As a result of these differences in the update policy, a consistent trend in most of the read-write experiments is that Tuple-Cache has a higher number of notifications and transaction rollbacks than either of the two A*Cache schemes, with A*Cache_Opt having the least number of notifications of the three.

4.2.2 Varying CacheSize with 40% Update Transactions

In this set of experiments, 40% of all transactions perform updates in both the private and shared regions. Figure 4 shows the results for clustered and unclustered data access patterns.

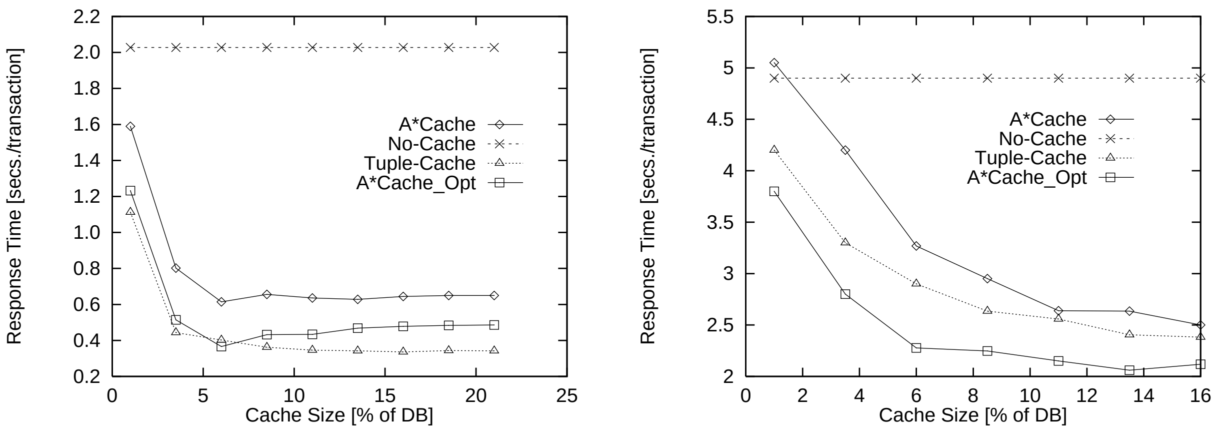

Figure 4: *Varying CacheSize, 40% Update Transactions, Clustered and Unclustered Access*

For the clustered case, it can be observed that Tuple-Cache actually performs the best for cache sizes larger than about 7% of the database, while A*Cache_Opt is the best for all values of cache size in the unclustered case. This difference in behavior arises from that fact that the server response is slower for unclustered data access compared to clustered, and thus the cost of predicate-based notification in A*Cache and A*Cache_Opt is relatively more expensive for the clustered case.

4.2.3 Effect of Varying PrivateAccessProb

This set of experiments varies the probability PrivateAccessProb of using data in the private region of the client, with a *PrivateWriteProb* of 0 and a *SharedWriteProb* of 20%. Thus, private data is read-only, but 20% of transactions in the shared region perform updates. The cache size is set to 10% of the database. Figure 5 shows the response time and number of notifications generated for this experimental environment.

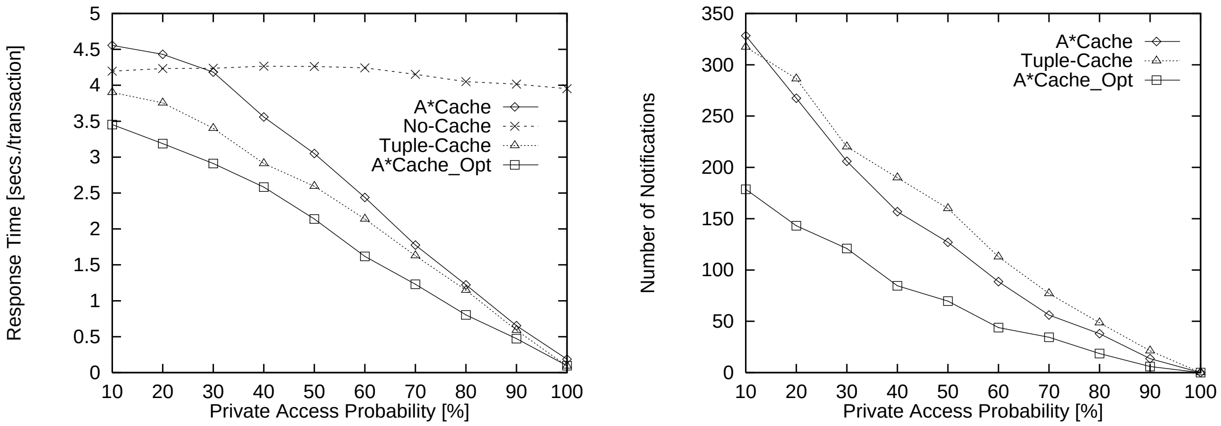

Figure 5: *Varying PrivateAccessProb, 20% SharedWriteProb, Unclustered Access*

Response time of all the caching schemes is seen to decrease nearly linearly with increasing private access, while the improvement for No-Cache is minimal. This behavior can be explained as follows. For our workloads, the private region of each client is 5% of the database size, and it fits entirely into the client cache, which is 10% of the database in these experiments. Thus, for the caching schemes, the server buffer is accessed only for the shared region. In contrast, the client-private data in the No-Cache scheme also resides in the server buffer, decreasing its efficiency. The unclustered scenario of this experiment compounds this effect, since many more disk pages are accessed for a query compared to the clustered case, and occupy pages in the server buffer. The net effect is that performance of the No-Cache scheme does not improve significantly with increasing frequency of private access, whereas the caching schemes demonstrate large improvements. The number of notifications decreases as more updates are targeted towards private regions, since contention and notification is lower. For any given PrivateAccessProb, Tuple-Cache has the largest number of notifications followed by A*Cache and A*cache_Opt respectively, due to the same reasons as explained in Section 4.2.1.

4.2.4 Summary of Read-Write Results

To summarize our read-write results, we find that all the caching schemes can yield significant benefits in the presence of moderate amounts of data update, up to 40% in certain scenarios

(Figure 4). Non-shared writes by a client yield better improvements for all the caching systems, since notification messages are not sent to other clients. In general, server CPU usage is higher for the caching schemes because of the cost of generating notifications, while the number of disk accesses is lower due to caching. With regard to the different caching schemes, A*Cache_Opt provides the best overall performance for most of our simulation experiments, followed by Tuple-Cache and then A*Cache. In some cases, such as the clustered access experiment in Figure 4, Tuple-Cache performance is comparable to or better than that of A*Cache_Opt, because of the increased cost of maintaining large predicate-based caches when there are many updates in the shared region. This result is not unexpected, but in this context it must also be noted that Tuple-Cache is efficient only when queries use indexed attributes to access data, as in our simulation; otherwise, full-fledged query execution is required at the server to determine the result set of an associative query, reducing the benefits of tuple caching.

4.3 Sensitivity Analysis

We ran numerous experiments to determine the sensitivity of the system with respect to the various simulation parameters, such as network bandwidth, server buffer size, server CPU etc., and also performed scalability experiments with larger number of clients in the system. We briefly summarize our results below, since the details cannot be presented here due to space constraints. For all read-only cases, the caching schemes exhibit greatly decreased sensitivity to shared resources, and scale much better with increasing number of clients than the system with no cache. The reason is that CPU and memory of additional clients are utilized in the caching schemes, and the load on the shared server resource is therefore much less than in the No-Cache case. With respect to larger query lengths, the following behavior was observed. Larger queries have predicate intervals that are more likely to overlap with each other, resulting in a greater number of cache hits and a slower growth in the query response time for increasing query lengths in the two A*Cache schemes. Tuple-Cache has approximately the same performance characteristics as A*Cache in this scenario, increasing slower than No-Cache with larger queries. However, Tuple-Cache does not beat A*Cache_Opt in performance, since it uses extra network trips and server cycles to detect cache containment at the server. A*Cache_Opt performs best among all four schemes, having the smallest increase in response time for queries with larger results sets. The results observed in the read-only scenario also extend to the read-write case, although the cache maintenance costs somewhat offset the savings from local data processing. For moderate values of the update probability (20% update transactions with 50% tuple writes), substantial benefits are still realized for all the caching schemes.

5 Related Work

Client-side data caching that is based on object or page identifiers has been studied quite extensively in [8, 18, 19]. However, associative caching has not been examined in any of these papers. Recently, a couple of studies [6, 17] have examined associative access to a client cache. Both of these studies are limited to read-only scenarios. The *semantic* caching study in [6] 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 [17]. However, it does not address the important issue of cache maintenance when there are database updates.

Evaluation of queries from locally stored views is supported in the ADMS$\pm$ system, which uses the *ViewCache* technique and *incremental access methods* [16]. A simulation study [7] 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.

Distributed and replicated databases are also related to the caching schemes considered in this paper. However, our 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 our case.

Maintenance of materialized views has been the subject of much research [10], 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 [4]. 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 [11] is a topic closely related to the cache containment question, and techniques developed in this area are directly applicable for A*Cache. The Tuple-Cache scheme is related to index-based associative caching schemes studied in [3, 20], with the difference that in Tuple-Cache, central indexes used by the server for determining the tuples in an associative query are not cached by clients. Thus, the issue of concurrency control of index pages at client sites does not arise in Tuple-Cache.

6 Conclusions and Future Work

In this paper, we have investigated through simulation the performance of three caching schemes that support evaluation of associative queries. One of the schemes, called A*Cache, was introduced by us in [13], and uses predicate-based descriptions to examine and maintain the cached data. The second scheme, named A*Cache_Opt, is an optimized version of the first, while the third, called Tuple-Cache, is a tuple-based scheme that uses server indexes to support local data reuse for associative queries. We first explained the operation of these caching schemes, noting in particular the differences in their query and update processing strategies. Next, we developed simulation models for the three caching schemes, and designed a workload to represent associative data usage by multiple clients. Our workload model is based on the single-user Wisconsin benchmark [9], augmented with locality and contention models for the client-server scenario [5]. It models both read-only and read-write transactions. We presented our experimental results for many different parameter settings, and analyzed the relative behavior of the caching systems. The benefits of caching were illustrated with respect to the baseline of a no-caching scheme. An important characteristic of the caching schemes studied here is that the only contention possible is on data; server indexes are not cached. Cache reuse and lookup is also independent of the data clustering on the server disk.

In conclusion, our results demonstrate that associative caching is an important and feasible technique to improve the performance and scalability of a client-server database through better utilization of client CPU and memory. We have shown that for our experimental settings, the performance of all the three caching schemes exceeds that of a no-caching system when there are no updates. When updates are present, our results demonstrate that under many scenarios substantial benefits can still be obtained for the caching schemes, even with moderately high data update probabilities such as 40%. When updates are targeted to client-private regions, contention and notification are infrequent, and associative caching is still effective. The caching schemes perform worse than no-caching when there is high contention on a small portion of the database shared 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 of the caches. The trade-off is between update notifications and local data processing, with larger caches generating more notification traffic, while the number of disk accesses is lower due to local data availability.

Comparing the relative performance of the three caching schemes, we find that A*Cache_Opt exhibits the best overall performance over a wide range of workloads, having the least query response time, network traffic, and number of disk accesses. This results from the fact that unlike A*Cache, A*Cache_Opt minimizes disk reads and network traffic for partial hits, and for full hits on the cache, it does not consume any server cycles in checking for tuple availability, as does Tuple-Cache. Thus, A*Cache_Opt yields the best results. The basic un-optimized version of A*Cache performs worse than Tuple-Cache in general. Tuple-Cache incurs an extra network trip to perform its cache containment checks at the server using central indexes, but it fetches only those tuples in the result set that are missing from its cache. Thus, network traffic in Tuple-Cache is smaller than in basic A*Cache, which fetches entire query results, disregarding the local availability of individual tuples.

The trade-off between the different caching schemes lies in the complexity of reasoning with query predicates, and the pattern of data reads and writes. We use simple range queries in our workloads, the query predicates being integer intervals. For more complex queries, the increased cost of predicate comparison can make the A*Cache schemes worse than both Tuple-Cache and No-Cache. It must be kept in mind however, that the index-based cache containment check in Tuple-Cache is effective only when queries use indexed attributes to access data; otherwise, full-fledged query execution is required at the server to determine the tuples in a result set, reducing the benefits of local caching. In contrast, predicate-based cache descriptions in the A*Cache schemes can handle more general query predicates than Tuple-Cache, since containment reasoning is performed on the predicates themselves, without using central indexes to determine the result set. In this sense, schemes using the A*Cache framework are more powerful than index-based caching methods like Tuple-Cache.

Future work includes the investigation of advanced cache maintenance policies, such as invalidation of infrequently-used query results and update of frequent ones. Additional knowledge or learning ability of data usage patterns is required for such intelligent maintenance. Instead of assuming a main-memory cache, the effect of client disks can be studied; however, it is not expected to alter the relative performance characteristics reported in this paper. Comparison of our A*Cache schemes with the index-based schemes proposed in [20] in the presence of index updates is another interesting issue to be explored. Examining more sophisticated cache replacement algorithms, such as extended LRU schemes, or those that consider both spatial and temporal localities of cached predicates [6], are other interesting research areas.

Acknowledgements

We gratefully acknowledge the many useful discussions with Kurt Shoens of Hewlett-Packard Laboratories and Prof. Gio Wiederhold of Stanford University. Kurt Shoens also ran several experiments on a commercial database to help validate the behavior of our simulator. We thank Prof. Hector Garcia-Molina for his suggestion to examine the tuple caching scheme.

References

[1] ANSI X3.135-1992, "American National Standard for Information Systems — Database Language — SQL," November 1992.

[2] J. Basu, "Associative Caching in Client-Server Databases," *Technical Report No. STAN-CS-TN-97-1606*, PhD Thesis, Stanford University, Computer Science Department, March 1998.

[3] J. Basu, M. Poess, and A. M. Keller, "Performance Evaluation of Centralized and Distributed Index Schemes for a Page Server OODBMS," *Technical Report No. STAN-CS-TN-97-55*, Computer Science Department, Stanford University, March 1997.

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

[5] 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.

[6] S. Dar, M. J. Franklin, B. Jonsson, D. Srivastava, and M. Tan, "Semantic Data Caching and Replacement," *Proc. 22nd VLDB Conf.*, Sept., 1996.

[7] A. Delis and N. Roussopoulos, "Performance and Scalability of Client-Server Database Architectures," *Proc. VLDB Conf.*, 1992.

[8] 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.

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

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

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

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

[13] A.M. Keller and J. Basu, "A Predicate-based Caching Scheme for Client-Server Database Architectures," *Proc. 3rd Intl. Conf. on Parallel and Distributed Information Systems*, Sept. 1994.

[14] Mesquite Software, "C++/CSim User's Guide," August 1994.

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

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

[17] P. Scheuermann, J. Shim, and R. Vingralek, "WATCHMAN: A Data Warehouse Intelligent Cache Manager," *Proc. 22nd VLDB Conf.*, Sept. 1996.

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

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

[20] M. Zaharioudakis and M. Carey, "Highly Concurrent Cache Consistency for Indices in Client-Server Database Systems";, *Proc. ACM SIGMOD Conf.*, May 1997.