ASSOCIATIVE CACHING IN CLIENT-SERVER DATABASES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Julie Basu
March 1998

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Gio Wiederhold
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Dr. Arthur Keller

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Hector Garcia-Molina

Approved for the University Committee on Graduate Studies:

Abstract

The client-server configuration is a popular architecture for modern computing systems, and for databases in particular. Such a database configuration 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. It is common for client transactions to run on desktop workstations, and to communicate with the database server using explicit messages across a local-area or a wide-area network.

A traditional assumption in the design of client-server databases has been that the clients have limited resources. Accordingly, client functionality has often been restricted to transmission of queries and updates across the network to the server, and presentation of the received results to the user. The server is a potential bottleneck in such systems, especially for large database sizes and many clients. Due to significant advances in computer technology, today's client workstations are often high-performance machines with substantial CPU and memory. It is now possible to use the resources of such capable clients for data caching and query evaluation purposes, so as to reduce the server workload and improve system performance and scalability.

In this dissertation, we propose and study an *associative* caching scheme, that we call *A*Cache*, for client-server databases. A*Cache supports client-side data buffering and local evaluation of associative queries. The cache at a client dynamically loads query results during transaction execution, and uses query predicates to formulate *descriptions* of the cache contents. New queries are compared against the cache description using predicate-based reasoning to determine if the query can be evaluated locally. In contrast to *navigational* data access using object or page identifiers, A*Cache provides predicate-based access to the cached data, thereby improving data reuse. Descriptions of client caches are also maintained at the server, which generates notifications for updates committed at the central database.

The clients use these update notifications to maintain the validity of their respective caches, and also to detect conflicting updates of shared data.

The A*Cache system requires dynamic reasoning with predicates, naturally raising questions about its performance and scalability. The focus of this dissertation is on the feasibility and performance of our caching scheme in a practical environment. We first describe the architectural framework of A*Cache and its execution model for transactions, and examine various design issues. We then develop new optimization techniques for A*Cache that can potentially improve its performance. The behavior of the A*Cache scheme is investigated through detailed simulation of a client-server database system under several different workloads and data contention profiles. We compare the performance of A*Cache with a system with no client caching, with a caching scheme based on tuples, and with an optimized version of A*Cache. Using an extended version of a standard database benchmark, we identify scenarios where the A*Cache schemes are beneficial, resulting in lower query response times and better scalability, thus demonstrating the effectiveness of associative caching.

Acknowledgments

It has been a long journey, and I have been helped by many along the way. I would like to take this opportunity to thank some of them.

First, I express my sincere gratitude to my adviser Gio Wiederhold for his guidance and encouragement. This dissertation would not have been possible without his support. Special thanks are also due to Voy Wiederhold for her genuine warmth and kindness.

I am truly grateful to Arthur Keller for his help in identifying my research problem, and for working closely with me all through. This research has benefited much from his good insights and valuable suggestions. I also appreciate his cooperation on publications, and his patience and flexibility in accommodating my odd working hours.

I thank the other members of my thesis committee, David Beech, Prof. Hector Garcia-Molina, Prof. Kincho Law, and Prof. Jeff Ullman for their willingness to serve on the committee, and for their constructive suggestions and help.

I gratefully acknowledge the many useful discussions with Kurt Shoens, especially on the performance study and implementation topics. His experienced analysis and practical suggestions helped much in our work. Even when he was busy with his transition to a new start-up, Kurt very kindly took the time to run several experiments on an Oracle database to validate the behavior of our simulator (Chapter 6).

I am much indebted to Meikel Poess for his contributions in our simulation study. Meikel worked diligently through many weeks and weekends coding the simulator and running experiments, came up with good questions and efficient solutions, utilized our whiteboard effectively, put up with my erratic schedules, and provided much motivation and encouragement. His cheerful presence definitely made a big difference to my life as a research student. His work on validating the simulator formed a very useful basis for my subsequent simulation experiments for performance analysis. Tables 6.5 through 6.7, and Figures 6.1 through 6.4 in Chapter 6 (Simulation Model and Experimental Setup) are adapted from his thesis with permission. Meikel's help in reviewing the final drafts of my dissertation is especially appreciated.

I am grateful to Prof. John Hennessey for providing financial support during my first three years at Stanford. My colleagues in the Stanford database group have been really great to work with. Discussions with Catherine Hamon were very useful during the initial stages of this work. Marianne Siroker always gave quick and expert help with administrative tasks. Martin Frost and Andy Kacsmar did a super job administering the computer systems. I had interesting and fun conversations on various topics with my officemates Michel Bilello, Ron Burback, Brian Lent, Fang Min, Dallan Quass, and with many other DBgroup members. Use of the various DBGroup facilities and the weekly lunches are appreciated.

This research was supported in part by educational grants from Oracle Corporation. To mention a few among the many people at Oracle who have taken the time to give their help and advice: Steffen Hulegaard, who persuaded me to continue my research; Bob Jenkins and Hakan Jakobsson, who participated in thought-provoking technical discussions; Alan Thiesen, who patiently listened to my professional and personal concerns; Sudheer Thakur, who proof-read my papers at short notice, and helped with my defense preparation; and Radhakrishna Hari, who contributed in many ways, from transplanting my plants to helping me move. I also appreciate the flexibility of my managers and co-workers in the PIDEV and SQLJ groups in accommodating my double-life as a student.

I also owe a great debt to the many people who encouraged me before I came to Stanford. The professors at Jadavpur University, Calcutta, and Indian Institute of Science, Bangalore, motivated and guided me in my studies. In particular, I thank Profs. A. K. Bannerjee, N. N. Biswas, S. Ghosh, L. M. Patnaik, and S. Rajaraman for believing in my abilities. Asis Goswami deserves special thanks for his friendship, help, and encouragement during my stay in Indian Institute of Science, Bangalore.

I am grateful to all the people who have been there for me one way or another. In particular, I thank Usha Anne, Nabanita and Gautam Bandyopadhyay, Lakshmi and Kamal Bakshi, Indrani and Subhendu Chaudhuri, Hanna Djajapranata, Vidya Jog, Basab and

Anu Maulik, Inderpal Singh Mumick, Sandhya Narayan, and Sujatha Subramani for their friendship and understanding in difficult times. The generous help from Sister Adelia has made, and still makes, a huge difference. Bill Youngster made my Commencement wonderful with his warm presence and special surprises. Many others go unnamed, but not unappreciated.

Last but not the least, I would like to thank my family. I would not have made it without them. My father has inspired me to strive beyond the ordinary, given me the confidence I often lacked, encouraged me to keep going when I faltered, and has provided active help and sound advice all through my life. My mother has been unfailingly supportive in every way, and has always listened to my troubles, big and small. My sister is an oasis of peace and friendship in a world that often seems stressful and chaotic. I am fortunate indeed to have such unquestioning love and support. This thesis is dedicated to them, and to the memory of my loving grand-parents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The client-server architecture [Elbe94] has proved to be very popular for modern database systems. This type of configuration involves one or more server processes that manage a shared repository of persistent data, and handle requests for data retrieval and update from multiple client processes. With the current popularity of distributed computing, it is common for client transactions to be initiated from desktop workstations, and to communicate with the database server using explicit messages across a local-area or a wide-area network.

1.1 Utilizing Client Resources

A traditional assumption in the design of client-server databases has been that client machines have very limited resources, and are likely to be fully loaded by local data processing. Accordingly, the common approach has been to design *query-shipping* systems [Fran96]. In these systems, client functionality is restricted to transmission of queries and updates across the network to the server, and presentation of the received results to the user. All DBMS tasks, such as query execution and transaction management, occur at the server. Since client memory or CPU is not taken into consideration for data caching or query evaluation purposes, such a system is said to have a *thin client* architecture. Most relational client-server databases available commercially today fall under this category. Figure 1.1 shows the overall architecture of a query-shipping system with a single client workstation.

The response of the server is a critical factor in the performance of a query-shipping system. Server resources are shared among many clients, and can become the bottleneck in

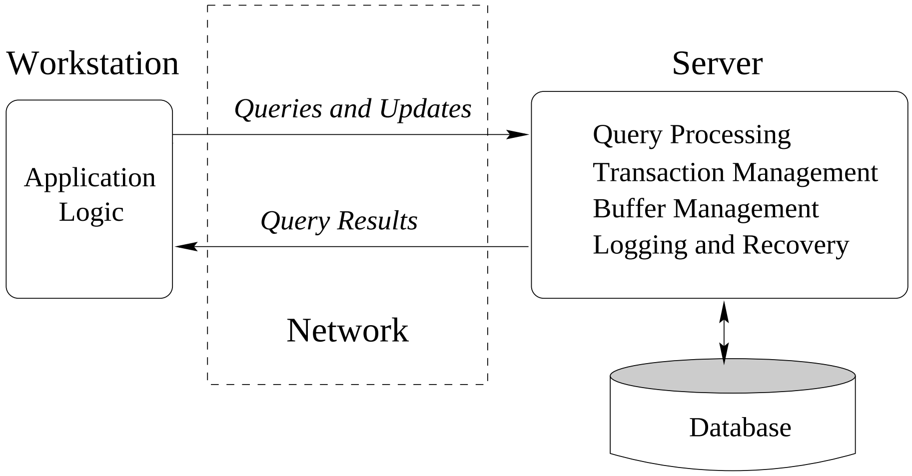


Figure 1.1: *The 'Thin Client' Query-Shipping Architecture*

scaling the system to large amounts of data or many clients. Optimizing the performance of the server has thus been a major focus of commercial systems. Consider, for example, the Oracle7 database [Orac96], which is a typical relational system with client-server configuration and a query-shipping architecture. Data caching is done on the server-side to avoid disk traffic, by using a main-memory 'buffer pool' of cached blocks, and queries can be parallelized for efficiency. However, there are no facilities for utilizing client resources.

Significant advances in computer technology have made powerful CPUs and large memories available at a small fraction of their earlier costs, so that today's clients are often high-performance machines with substantial processing power. These *smart* or *thick clients* are capable of performing intensive computations on their own, using the database as a remote information source that is accessed only when necessary. Increased client functionality can potentially improve the performance and scalability of the system by lowering query response times and off-loading the server, thereby allowing a larger number of clients.

1.1.1 Dynamic Data Caching at Client Sites

In addition to server buffers, easy availability of abundant local memory allows data caching at client workstations. Client-side data caching can enhance the overall performance of a client-server database, especially when the operational data spaces of client transactions are mostly disjoint, and contention on shared data is not excessively high. Dynamic caching of locally pertinent and frequently used data constitutes a form of replication, whereby each client dynamically defines its own data space of interest. Unlike static data replication schemes [Ceri84], dynamic caching is based on queries submitted by client transactions at runtime, and does not require that a fixed subset of the database be specified a priori.

There are two types of cache reuse in dynamic caching systems: *intra-transaction* and *inter-transaction*. In both of these methods, a single transaction can cache and reuse objects within its individual scope of execution in order to speed up the response. Inter-transaction caching additionally allows cached data to be reused by future transactions at the same client. Although additional costs are involved in maintaining the cache across transaction boundaries, longer-term reuse of cached items can increase cache utilization, and reduce network traffic and query response times.

Caching Based on Object Identity

Object-oriented databases generally assume smart thick clients, and are built using a *data-shipping* approach (Figure 1.2). In the data-shipping model, the clients request data pages or objects from the server, and cache them locally in memory. Cached items are looked up by their unique identifiers and reused at client sites, with page or object faults causing data to be fetched from the server as necessary. Query processing tasks do not occur at the server, but are distributed to the clients; the server is essentially a *page server* or an *object server* [Dewi90], with transaction management services. Inter-transaction reuse of the cache is supported by using an identity-based maintenance protocol to either update or invalidate locally cached copies of objects or pages.

Cache access based purely on identifiers is adequate for *navigational* ID-based operations, such as *ReadObject* and *UpdateObject*, which are common in object-oriented databases [Catt91]. However, local execution of more general *associative* queries cannot be supported at the client site in this type of caching scheme. An associative query specifies a target set of tuples or objects using predicates on the attributes of a relation or an object class, e.g., through a selection condition in the `WHERE` clause of a `SELECT-FROM-WHERE` statement in the database language SQL [ANSI92]. Such queries support *value-based* access to data, and are very common in relational databases. Even in object-oriented databases, an initial set of objects is often retrieved using predicates on the object attributes, with navigational access occurring subsequently as the objects in the result set are traversed in user memory. Evaluation of associative queries on locally cached data is therefore important for both relational

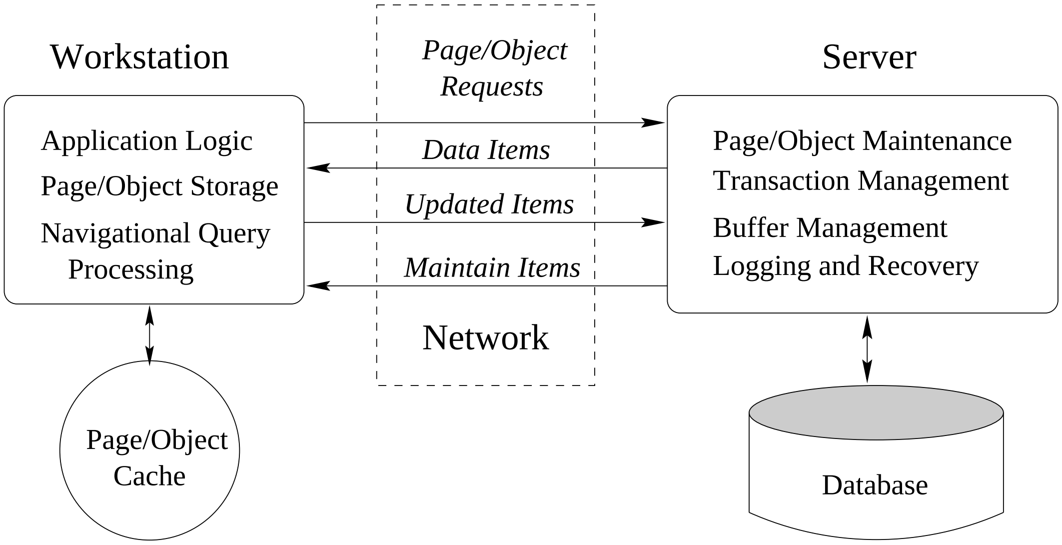


Figure 1.2: *The 'Thick Client' Data-Shipping Architecture*

and object database systems, and is the subject of the work reported in this dissertation.

Associative Caching

An associative caching scheme supports predicate-based access to cached data, instead of retrieval only through unique identifiers. Associative access can improve cache reuse, since not only navigational but also associative queries may be evaluated on a cache locally. Local query processing leads to significantly higher utilization of client CPU and memory, and can potentially improve the performance and scalability of the database system as a whole.

In comparison to identity-based caching, operation of an associative cache is more complex in two major respects: cache *containment* reasoning and cache *consistency* maintenance. In the former type of caching, it is obvious whether a desired object is cached or not, and the server can easily maintain information on which clients have which objects cached, so that the appropriate clients can be notified when an object is changed. In contrast, determining whether an associative query can be answered from the cache requires reasoning with cached query predicates. It is also not sufficient to consider data items individually for maintaining the consistency of an associative cache. Instead, predicate-based reasoning must be employed to determine which client caches are affected by an update.

1.1.2 An Example of Associative Caching: A BUG Database

We introduce an example here to illustrate the concept and utility of associative caching, and the issues in maintaining such a cache. This example will be elaborated and used through this dissertation to clarify various points.

The BUG Application

Consider a large software company with a primary Development Center at a single location 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 a unique bug number, a product number, customer name, priority, current status, and a problem description. Assume that it is represented using the following relational schema:
`BUG(Bug#, Product#, Customer, Priority, LogDate, Description, Status)`,
where each row records the details of a particular bug.

Usage of the BUG 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 may gather statistics on bug activity for different products or customers. Bugs may be updated after review, or in response to customer interaction. Once entered in the database, a bug cannot be deleted, although its status can be updated to *Resolved* or to *Not a bug*.

Let us assume that the BUG database is implemented using a relational 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 the latency to retrieve data from the central server could be quite substantial otherwise.

Determining Cache Containment

In our BUG example, users access data in an associative manner, e.g., based on the product information and the status of bugs. Thus, associative access to cached data is required for effective cache reuse at the client.

Suppose that a client $C1$ caches the result of a query for all open bugs in product number 100, along with a predicate P describing these tuples:

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

Assuming that none of these bugs are updated by any client, a subsequent query at the client $C1$ for all open bugs in product number 100 with *High* priority, i.e., those bugs that satisfy the predicate

Q : `(Product# = 100 AND Status = 'Open' AND Priority = 'High')`,

can be answered using the local cache associatively. In this case, containment reasoning is required to detect that predicate Q is a subset of the cached predicate P. Another query denoted by the predicate R,

R : `(Product# = 100)`,

asking for all bugs in product number 100 can only be partially answered from the cache of client $C1$. In this case, the client could request the database either for all bugs in product number 100, or only for those that are not open; this choice is a new and important optimization decision, called *query trimming*, that can potentially speed up data transmission and query processing.

Inserting New Bugs

Assume that a new bug is logged on product number 100. This insertion would cause a cache storing the predicate R to become incomplete, since the new bug falls under the scope of this predicate. In this case, either the new tuple could be inserted into the cache, or the predicate R could be eliminated from it.

Effect of Updating a Bug

Now assume that a client $C2$ commits an update that changes the status of a bug in product number 100 from *Open* to *Closed*. This update potentially affects all clients that have cached results, even a client $C3$ that stores a predicate `(Customer = 'Cust1')`. For client $C3$, updating the modified tuple if it is present in its cache is an appropriate maintenance action that maintains the validity of its cache. For client $C1$, the updated tuple previously belonged to the predicate P, but no longer satisfies it. The tuple may be dropped from the cache, if no other cached predicate is referring to it.

Thus, the situation can be quite complex if the cached data is out-of-date as a result of updates committed at the server. For some client transactions, reading data that is somewhat stale might be acceptable; for others, currency of the data could be crucial. Ideally,

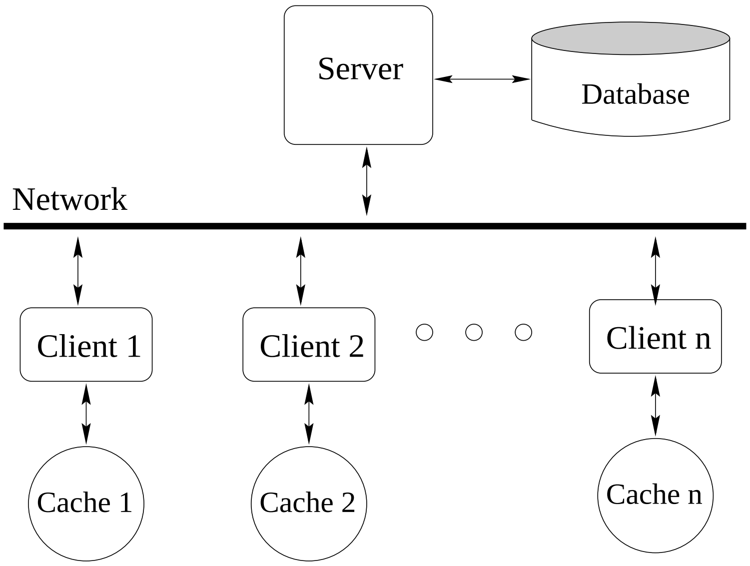


Figure 1.3: *A*Cache Client-Server Configuration*

the cache maintenance policy should be flexible enough to support varying requirements, such as specification of the maintenance mechanism per client or per query. For example, results of queries known to be frequently asked could be automatically refreshed, but a random query result invalidated and eventually purged from the cache.

1.2 Contributions of the Thesis

We propose and study an *associative* caching scheme that we call *A*Cache*. A*Cache is based on query predicates, and supports the execution of associative queries at client sites. Our focus is on the performance of an A*Cache system in a practical environment. The goal is to examine, through qualitative analysis and simulation, the trade-offs involved in improving the performance of a client-server database through A*Cache.

The configuration of an A*Cache system is shown in Figure 1.3. Multiple autonomous client workstations interact with a central server via messages across a network. The persistent data store is resident at the server and transactions are initiated from client sites, with the server providing facilities for shared data access.

Queries submitted to the server are used to dynamically load data into a client cache, and predicate-based *cache descriptions* derived from these queries are stored at both the

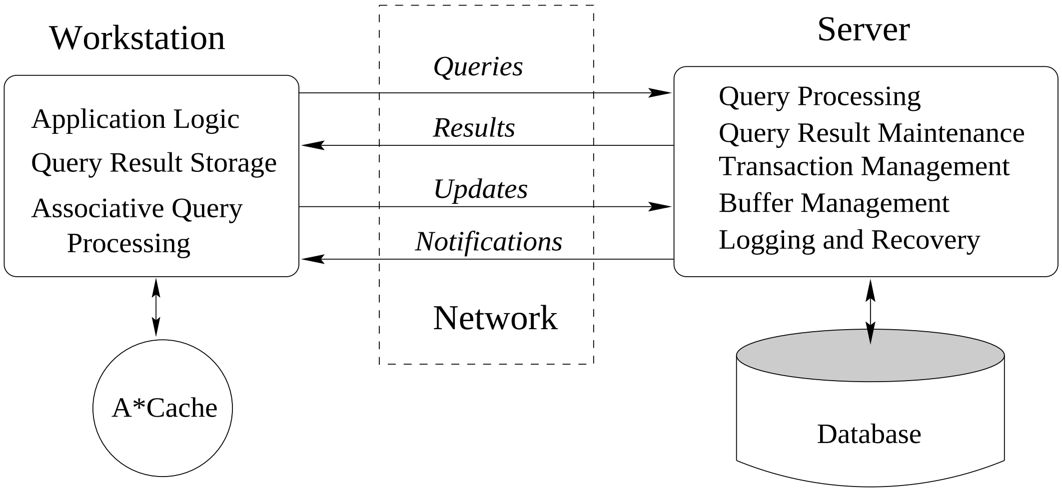


Figure 1.4: *A*Cache 'Hybrid-Shipping' Architecture*

client and the server in order to examine and maintain the cache contents. Figure 1.4 shows an overview of the architecture of an A*Cache system. Observe that the server in A*Cache has full query execution capabilities. Thus, A*Cache is neither a pure data-shipping nor a pure query-shipping system; rather, it is a hybrid of the two architectures [Fran96]. It attempts to exploit the resources of the client and the server in a flexible way.

When a transaction submits a new query or update, it is intercepted locally by the client, and checked for containment against its cache description. The operation may be executed locally, if the client can determine that the data required by the operation is entirely available in the cache. On the other hand, a cache miss results in a request to the server to evaluate and send the required data. The result of this remote query execution is optionally added to the client cache, and its descriptions at the server and at the client are updated appropriately.

Client caches may become out-of-date as updates are committed on the central database at the server, and an appropriate cache maintenance algorithm must be employed in order to satisfy data consistency requirements of client transactions. In A*Cache, the clients must register their cached predicates at the server in order to be notified of changes to their cached data. The server uses these client *subscriptions* to send notifications of committed updates that are possibly relevant for each cache. In essence, client subscriptions serve as *notify locks* [Gray93b, Wilk90], that are based on query predicates. The clients use these update notifications to refresh or invalidate predicates and data in their respective caches, and also to detect any conflicting access to data locally read or written.

In this dissertation, we first describe the architecture of the A*Cache system, and discuss various issues, such as concurrency control and cache consistency maintenance, that arise in its design. We address the performance concerns in using predicate-based reasoning by identifying new opportunities for optimization, and suggest techniques that promote good dynamic behavior. For example, descriptions of the cache that are used for determining cache containment need not be exact and may be *conservative*, so that all objects claimed to be in the cache are indeed available in it. It is inefficient but not incorrect to re-fetch a locally cached object from the server. On the other hand, the server may use a *liberal* description of a client cache, ensuring that a client is informed of all updates relevant for its cache, and possibly of some irrelevant ones. Other optimizations that can potentially speed up query evaluation and notification processing, such as predicate indexes in containment reasoning, predicate *merging* for compact cache descriptions, query *trimming* and *augmentation* to reduce caching costs, are also discussed.

We describe the steps in transaction execution in an A*Cache system, and the events that occur in the system in response to the different types of database operations. In this thesis, a *semi-optimistic* concurrency control protocol is assumed, in which a client uses notification messages from the server to detect any conflicting updates of the shared data by other clients. We demonstrate that serializability of client transactions is retained using such a concurrency control scheme.

We then investigate the behavior of the A*Cache scheme via detailed simulation of a relational database system. The performance of A*Cache is evaluated under many different workloads and compared against other caching systems, such as a *standard* client-server system with no client-side caching, and a *tuple-caching* system that caches only tuples and no predicates. An optimized version of A*Cache is also examined. Using an extended version of a database benchmark (the WISCONSIN benchmark [Gray93a]), we identify scenarios in which the A*Cache-based schemes are beneficial. The performance benefits are measured in terms of lower query response times, reduced network traffic, and better scalability with large number of clients. Our simulation results clearly demonstrate the benefits of associative caching for read-only environments, and for read-write environments with moderately high write probabilities.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 gives an overview of related work. We introduce the A*Cache scheme in Chapter 3, and discuss important design issues such as concurrency control and cache maintenance. In Chapter 4, we suggest new techniques for optimization that can be applied in A*Cache. Chapter 5 examines in detail the effect of various database operations on the system, and shows the serializability of transactions in A*Cache. In Chapter 6, we describe a simulation model for a relational database system with client-side caching using A*Cache, and also our workload model based on the WISCONSIN benchmark. Chapters 7 and 8 present the results of our simulation experiments. We summarize our contributions and outline future research issues in Chapter 9.

Chapter 2

Related Work

2.1 Background: The Penguin Project

The need for associative caching arose in the *Penguin* research project [Wied86, Bars90, Bars91] at Stanford University. The Penguin system materializes *view-objects* from data stored in a relational schema by using *data access functions*, which specify joins on the database relations for constructing the view-object. The materialized view-objects can be traversed navigationally at the client site. Fetching data from the server as *relation fragments* instead of a single flat relation was proposed in [Lee90], and its efficiency demonstrated. Local caching is likely to improve the performance of such a system by reducing the number of remote trips to the server. However, there is no notion of object identifiers for Penguin view-objects, and therefore, identity-based caching is inadequate in this case. Associative caching techniques are required to detect and reuse query results that were obtained earlier, but are still available for providing a local response to another query.

2.2 A Survey of Caching Schemes for Databases

The utility of data caching in improving system performance is well-accepted in databases. For example, [Alon90] describes issues in the design and performance of *quasi-copies*, which have client-specified coherency conditions that allow local copies to diverge in a 'controlled' fashion.

Over the last few years, caching in client-server environments has been studied quite extensively, mainly in the context of data-shipping object-oriented databases. We review below recent research on both navigational and associative caching techniques for client-server databases.

2.2.1 Identity-Based Caching Systems

In [Fran93], a comprehensive study of inter-transaction caching schemes for object servers and page servers is presented. Performance of various consistency control methods for cached data is investigated via simulation. Special locking and maintenance algorithms for optimization of caching performance in *page-server* systems are explored in [Care94]. Earlier works that deal with client caching issues include [Care91a, Fran93, Wang91, Wilk90]. More recent examples of work in this area are [Lome94, Adya95, Shri96], and [Cast97].

In all of the above schemes, storage, retrieval, and maintenance of cached objects at client sites are done based on object or page identifiers. As noted in the Introduction, this type of caching can only support the navigational identity-based lookup common in object-oriented databases. Unlike the A*Cache scheme proposed in this dissertation, associative queries that access data using predicates on relations or object classes cannot be answered locally at the client in these systems.

Index-Based Associative Access

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 index pages can be examined to determine which objects satisfy the query. These index pages may be used in either a centralized or a distributed manner. In the centralized scheme, indexes are managed solely by the server and cannot be cached by clients. A client submits each associative query to the server, which responds with a list of qualifying object IDs. The client then locally checks its cache for object availability, and fetches missing data objects or pages as necessary. The centralized index scheme requires communication with the server for all index-based queries. A distributed alternative is to allow clients to cache and maintain indexes locally. This requires the enforcement of a consistency control protocol on index pages. Because an index page relates to many more objects compared to a data page, it generally has very high contention, and may be subject to frequent invalidation or update. Distributed index maintenance is therefore likely to be expensive even in systems that have low to moderate update activity. Recent studies, such as [Basu97, Gott96, Zaha97], have investigated efficiency issues in accessing and maintaining cached indexes in client-server object databases.

2.2.2 Associative Caching Schemes

In associative caching schemes, server indexes need not be referenced to answer associative queries from the data in the cache. Instead, containment reasoning on query predicates is used to determine whether the cached query is available locally.

Associative access to cached data has been suggested in several studies. For example, [Fink82] is an early work that examined the reuse of data in common subexpressions of queries. A predicate-based *partial indexing* scheme for materialized results of procedure-valued relation attributes was outlined in [Sell87]. The ideas can be extended to conventional database systems, but were not developed and explored in that context, or for client-server architectures. The work of Kamel and King [Kame92] deals with associative caching of database queries, but is meant for applications that have a pre-determined set of queries requiring repetitive re-evaluation, such as for integrity constraint checking.

We identify below some associative caching schemes that are more closely related to A*Cache, and point out the differences of these systems with respect to it.

- ADMS±

  In [Rous91], a *ViewCache* scheme that uses the notions of *extended logical access path* and *incremental access methods* was proposed. A workstation-mainframe database system called ADMS± that is based on the ViewCache scheme is described in [Rous95]. In this system, query results retrieved from the server(s) are cached on local disks of client workstations. Updates are performed at the server, where update logs are maintained for the purpose of cache refresh. Queries against cached data at a client may result in an explicit refresh request to the server to compute and propagate from these logs the differential changes for the query. Performance of client-side caching schemes that are based on the ViewCache technique is studied via simulation in [Deli92]. In the 'Enhanced Client-Server' system investigated in this work, fetching incremental update logs from the server(s) was found to be a bottleneck with increasing number of clients and updates. The authors then propose and study a log buffering scheme to alleviate the problem.

The A*Cache scheme is different in several important ways from ADMS±. Firstly, cache maintenance in A*Cache is based on notifications of committed updates rather than on centralized update logs at the server. Clients participate in notification processing to maintain their caches incrementally, and also to detect conflicting access on shared data. The concurrency control scheme in ADMS± is very different from that of A*Cache, and uses a "derived-object" lock to protect access to the materialized views. Additionally, our notification scheme is designed to be flexible; for example, in some special cases of data updates, network traffic may be saved by propagating just an update command and not the updated data (Section 3.3.5). The client can execute the update query to refresh its cache. This strategy attempts to split the workload of maintaining cached results more evenly amongst the clients and the server than ADMS±.

Another significant difference of A*Cache with ADMS± is the data storage scheme at a client. In A*Cache, the data is placed in partial local copies of database relations, with multiple query results on a single relation being merged by their key attributes into the same table (Section 3.3.1). This storage structure is compact, and enables easy maintenance of cached tuples. In contrast, downloaded base relations and downloaded views in ADMS± are stored in different storage structures, with the base relation having the same schema as in the database, but the views using pointer arrays. Therefore, base relations and views at the workstation are maintained differently, the latter requiring manipulation of pointers. A special query optimizer was proposed in ADMS± to identify the relevant ViewCaches and generate alternative query plans for local query execution.

- *BrAID*

  A caching subsystem that can reason with stored relations and views is proposed in the *BrAID* system [Shet91] 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 also relevant for A*Cache. However, consistency maintenance of multiple client caches in the presence of database updates is an important issue not addressed in this work.

- Semantic Caching and WATCHMAN

  Recently, two studies [Dar96, Sche96] have examined associative access to a client cache for read-only scenarios. As noted in these papers, the schemes are related to A*Cache; however, unlike A*Cache, database updates are not considered. The *semantic* caching work presented in [Dar96] focuses on cache replacement policies for read-only workloads, and examines application of the scheme in a mobile computing environment. A simulation study is used to demonstrate the benefits of this approach over page-based and tuple-based caching and replacement strategies. A report on the design of an intelligent cache manager, called WATCHMAN, for caching query results in data warehousing environments appears in [Sche96]. Algorithms for cache admission and cache replacement that are based on query result sets instead of individual pages are investigated. The performance of WATCHMAN is evaluated using the TPC-D and Set Query benchmarks, and shown to be superior to traditional LRU replacement policies. However, the environment is limited to read-only workloads, and the effects of database updates and cache maintenance have not been addressed.

2.3 Related Work in Other Areas

Below, we survey work in areas that do not deal directly with caching, but have concepts related to A*Cache.

2.3.1 Predicate Locking

The A*Cache scheme is reminiscent of *predicate locks* used for concurrency control [Eswa76], where a transaction can request a lock on all items that satisfy a given predicate. Predicate lock implementations have not been very successful, mainly due to their execution cost [Hunt 79, Gray93b], and because they are pessimistic in nature and can reduce concurrency excessively. For example, consider two predicates that intersect in the attribute space of a relation, such as `(Product# = 100)`, and `(Status = 'Closed')`. Even if there is no tuple in their intersection for a particular instance of the relation, two different transactions will nonetheless be prevented from simultaneously write-locking these predicates. This rule protects against *phantoms*, but can cause fewer transactions to execute concurrently, and thus reduce the system performance.

In contrast, A*Cache uses predicate-based notify locks that are more optimistic, in that two transactions using cached predicates at different clients conflict, and are notified by the server of the conflict, only when a tuple in the intersection of shared predicates is actually updated or inserted. A similar scheme called *precision locks* was proposed in [Jord81] for centralized systems.

2.3.2 Query Containment

Query containment [Levy95, Sriv96] is a topic closely related to the cache containment question. Precursors to this work include [Lars87], which examines query evaluation on a set of *derived relations*. These techniques can be directly utilized for cache containment reasoning in A*Cache. Efficiency of the reasoning process is a valid concern, and is addressed in A*Cache using approximations and indexing techniques (Chapter 4).

2.3.3 Materialized Views

Cache consistency maintenance in A*Cache essentially involves maintaining *materialized views* that are the query results stored at client sites. Efficient incremental maintenance of materialized views has been the subject of much research [Blak86, Ceri91, Gupt93, Gupt95, Levy93, Stau96, Qian91], and many of these techniques are also applicable in A*Cache. For example, our update notification scheme involves detecting whether a client cache is affected by an update, and this issue is related to the filtering of updates that are irrelevant for a view [Blak89, Levy93]. However, these view maintenance schemes have been designed for relatively static situations with small numbers of queries or pre-defined views. Performance problems in handling large numbers of dynamic views in a client-server environment, as in A*Cache, have not been considered in these papers.

Concurrency control of transactions that read materialized views as well as base data is addressed in [Kawa96], and a serializability theory is introduced for this scenario. Our concurrency algorithm in A*Cache is based on notify locks, and differs from the usual read and write lock-based approaches investigated in the above study. Additionally, local writes on the cached views are permitted in A*Cache, with a commit verification scheme to detect any conflicting updates (Section 3.3.3).

2.3.4 Distributed and Replicated Databases

Extensive research has been done in the area of data distribution and replication, [Brei92, Ceri84, Davi85, Ston94] to name only a few. Client-side caching is a form of data replication, and many similarities exist between A*Cache and replicated or distributed databases, including a shared motivation to reduce the latency of data access. However, the design issues for A*Cache with its centralized server model are in some ways quite different from those of multiple databases operating in a distributed environment:

- Caching in our scheme is performed *dynamically* based on user queries. A*Cache does not have a static replication scheme defined a priori, and which query results are cached is determined only at application runtime.

- A client cache does not provide full-fledged database services at the client. Among other things, this implies that the caches do not support complex protocols for local logging of updates and crash recovery, thus simplifying their design.

- The client caches operate autonomously, and each client communicates only with the server. Thus, coordination among different clients occurs indirectly through the central server, and direct communication among individual clients is not considered.

- Lastly, we have the important difference that coupling of the local caches with the database server is 'tight', and the final commit must always take place at the server. The central database is the single 'point of truth' as far as persistence and durability of data is concerned (the 'D' in the transactional ACID property [Gray93b]), and the clients are only have *second-class ownership* [Fran93] of their cached data.

Despite the above differences, a major similarity between A*Cache and distributed databases is that queries may be executed at multiple sites, which in A*Cache are the server site and the client that originated the transaction. This mixture of query execution sites in A*Cache raises similar issues of distributed concurrency control [Ceri84]. In addition, the consistency concerns in replicated databases on reading potentially out-of-date copies and the possibility of multiple conflicting updates [Gall95, Wied90] also arise in A*Cache.

However, because of the tight coupling with the server in A*Cache, we detect conflicting updates before transactions can commit. This detection is performed as part of the notification processing at the client, and also during commit verification at the server (Section 3.1). Rules for conflict resolution of multiple updates to replicated data [Care91b] are therefore not required in A*Cache.

2.3.5 Active Databases and Integrity Constraints

Rule systems for triggers and active databases [Hans93] are related to our notification scheme, in the sense that efficient identification of applicable rules is desired for each database update. Such rules are generally specified in the *Condition-Action* or *Event-Condition-Action* format, where the *Condition* part is expressed as a predicate over one or more relations. Hence, detecting the firing of a rule involves determining satisfiability of predicates, and similar efficiency issues as in A*Cache arise in such systems. Integrity constraint checking [Bune78] is another area that involves similar predicate-based reasoning.

One feature of A*Cache different from active databases is that notification processing can be approximate as long as it does not cause erroneous behavior. False triggering of a rule is generally not acceptable in active databases or for integrity constraint checking. Additionally, our notification scheme has the capability of directly propagating certain update commands to clients for local execution on their cached data, instead of propagating the tuples modified by the update (see Section 3.3.5). Therefore, unlike rule systems in active databases that handle sets of tuple-level operations [Hans93], the A*Cache notification system handles not only instances of modified tuples, but also general predicates involved in update commands.

2.3.6 Caching in Distributed File Systems

Client-side caching has also been investigated in other areas, such as distributed file systems, for improving system performance and scalability. A survey of the many different schemes appears in [Levy90], one example being the Andrew File System [Howa88]). The unit of caching may be pages or entire files, and unlike A*Cache, is based on unique identity of cached items. A significant difference between caching in databases and in file systems is that transactions are typically not supported for files. The burden is generally on the user to coordinate any conflicting access, which is often done using an auxiliary source control mechanism implemented on top of the file system.

2.3.7 Caching in Shared-Memory Multiprocessors

Extensive research has been done in caching and cache coherence issues for shared-memory multiprocessors, which typically have private local caches to reduce network traffic. A large family of coherence protocols, called *snooping* protocols [Good83], depend on monitoring the traffic on a shared bus to detect updates. Unlike local area networks in client-server configurations, the bus is a cheap and fast broadcast medium. Also, cache lines are typically counted in tens of bytes, whereas query results from a database are measured in kilobytes. The communication costs are therefore very different for the two scenarios. Other protocols studied in this area include *directory-based* schemes [Agar88], in which location of cached copies is tracked so that maintenance messages can be directed appropriately. Similarities exists between this scheme and the notification mechanism in A*Cache; however, the notion of predicate-based access and update does not arise in shared-memory multiprocessors.

2.3.8 Other Related Areas

We now consider research that is related to ours in a broader perspective.

Data Consistency

A recent paper [Hull96] on data integration defines notions of consistency for *virtual*, *materialized*, and *hybrid* views of database relations. These consistency concepts are also applicable for A*Cache: virtual views are those queries that are executed at the server, materialized ones are cached queries, and hybrid ones are those that execute partially at the client and partially at the server. However, [Hull96] uses a model in which read-only queries are interleaved with update propagation. A*Cache additionally supports read-write transactions at client sites. A subject of future research is to extend the consistency model in [Hull96] to associative caching environments such as A*Cache.

Natural Languages

In the area of natural languages, queries are typically issued in context, allowing elliptical expression, such as ‘Where is the meeting that you are attending?’ followed by ‘How long are you staying there?’ or ‘In which hotel are you staying?’. The work of [Davi82] considered these elliptical queries and what they mean for databases, since the context is necessary to have well-formed queries. [King84] used the knowledge of context as an opportunity for optimization, taking advantage of the data cached from earlier queries to reduce the search space for the elliptical successor queries, much in the spirit of our A*Cache work.

2.4 Summary of Chapter

In this chapter, we have reviewed work related to our area of research. We surveyed the current work on caching systems in client-server databases, including identity-based schemes such as [Care94], and index-based studies [Zaha97]. Caching of materialized views, as in A*Cache, has previously been investigated in the *ViewCache* system described in [Rous91], and in *BrAID* [Shet91]. The differences of A*Cache with these schemes have been noted. More recent and closely-related studies include the semantic caching scheme of [Dar96], and the WATCHMAN system of [Sche96]. In contrast to the work reported in this dissertation, both of these studies consider only read-only access to the database, and focus primarily on cache replacement policies. We have also compared A*Cache with distributed and replicated databases — data replication schemes are static in these systems, unlike the dynamic environment of A*Cache.

Chapter 3

A*Cache Architecture

In this chapter, we introduce a client-side caching scheme called A*Cache for client-server databases, and discuss the issues and trade-offs in operating such a system. A major feature of A*Cache is that it allows inter-transaction and associative reuse of locally cached data. The primary design goals of such an associative cache are twofold: (1) to reduce network traffic and query response times by utilizing client resources, and (2) to increase system throughput and scalability.

3.1 Configuration of an A*Cache System

The A*Cache environment consists of a database managed by a central server, and multiple clients that interact with the server across a network. The persistent data store is physically located at the server, and transactions are initiated from client sites. The server provides transactional and recovery facilities for the shared database. The system has a non-shared memory architecture in which the server and the client processes have mutually disjoint local address spaces, and communication between a client and the server occurs only through explicit messages across the network.

Data is dynamically loaded from the server database into a cache based on queries submitted by client transactions, and the current contents of the cache are described by predicates derived from the query predicates. Thus, the A*Cache scheme essentially supports client-side caching of multiple views, which are the results of query evaluation during execution of client transactions.

The characteristics of the A*Cache system are as follows:

- Space for the cache may be allocated either in main memory or on secondary storage, such as a disk. We make no particular assumptions about the nature of the cache memory, except that it is private to the client.

- The operation of the clients and their caches are assumed to be autonomous. A client is not aware of other clients or of their caches; each client communicates only with the server.

- The server and the clients are each assumed to have query execution capabilities on the database and individual local caches respectively. Thus, unlike the page or object servers for OODBs [Dewi90], the server in our scheme does not merely ship data items to the clients; instead, it may actively participate in query execution. Unlike traditional relational databases, clients may also take part in query evaluation, thus making the A*Cache model a 'hybrid-shipping' one [Fran96].

- The server is kept informed whether a client is caching data locally, as well as which query results are being cached.

- Messages are received and processed at the client in the same order as they are sent from the server, with the network preserving the number and order of messages. These requirements are currently met by most networks.

- In this thesis, we assume that each client executes transactions sequentially, with at most one transaction active at any time. This assumption does not affect the cache containment reasoning and maintenance algorithms described in this thesis. The A*Cache framework can be extended to support simultaneous execution of multiple transactions at a given client, by having local concurrency control and transaction isolation capabilities on the shared cache.

3.2 A*Cache Components

In order to manage the operation of the client caches, a server-side subsystem is present at the central server, and a client-side subsystem exists at each client that uses a local cache. Figure 3.1 shows the components in an A*Cache system with a single client. We describe below the functions performed by these components; details of the interaction of these components during execution of transactions appear in the next chapter.

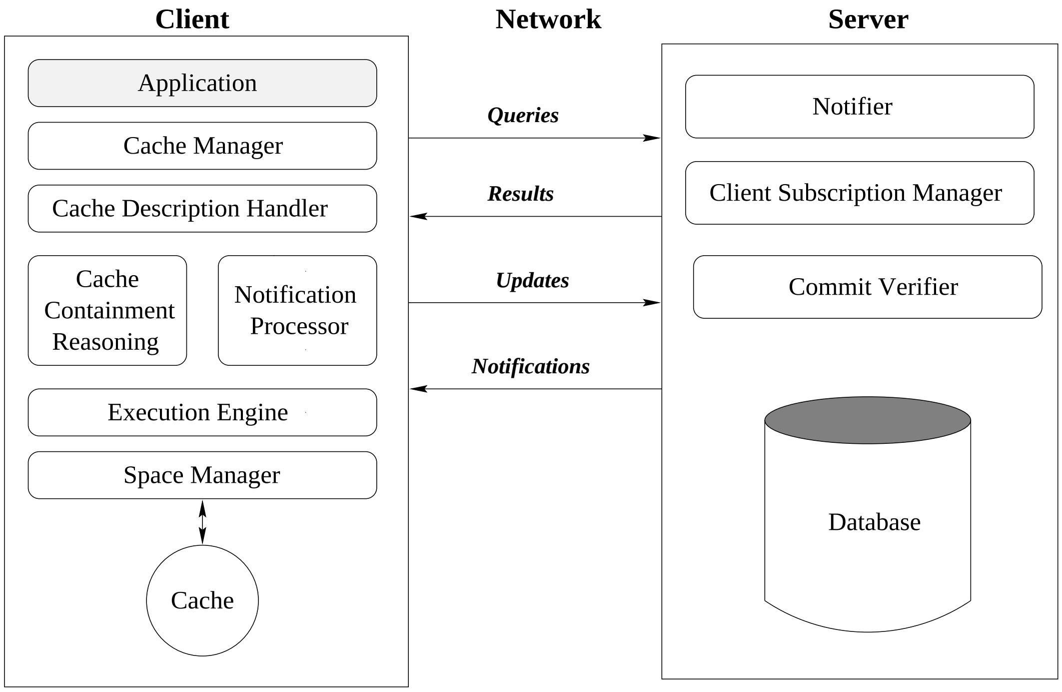


Figure 3.1: *Functional Components in an A*Cache System*

3.2.1 Client-Side Components

The caching system performs several functions at the client site — it decides whether a new query result should be cached, how to reclaim space when the cache is full, how to execute queries locally, and most importantly, whether a new query can be answered using previously cached results (the cache containment issue). Processing of update notifications sent by the server to maintain the validity of cached data is another important task.

Associated to each client-side caching subsystem is a *client cache manager*, which consists of six distinct components, as depicted in Figure 3.1:

- **Cache Manager**

  A client application may issue queries and updates to the database, as well as other commands such as commit, abort, rollback, savepoint etc. These operations on the database are intercepted by the Cache Manager, which is the 'controller' component at the client. It is the responsibility of this module to determine the type of database operation requested by the application, and whether it can be handled locally. If not, the Cache Manager routes the request to the remote server, and upon receiving a response, invokes the appropriate client-side modules for processing. The Cache Manager module consists of sub-components such as a command parser that analyzes user requests, and a dispatcher that routes the requests 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 3.3.3).

- **Cache Description Handler**

  Predicates are inserted into and deleted from the client cache description by the Cache Description Handler. Modifications to the cache description may be required when a new query result is stored, when a previously cached result is purged from the cache, and during processing of update notifications. This module also keeps track of usage information for predicates, which is used for space management purposes.

  Optimization issues in maintaining cache descriptions are discussed in Section 4.1.

- **Cache Containment Reasoning**

  The function of the Containment Reasoning subsystem is to *conservatively* (see Section 4.1) compare a query against the current cache description, and determine whether the query result is completely or partially 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. Containment reasoning is required during notification processing to determine if the cache is affected by an update. Algorithms in answering queries from materialized views [Levy95] can be used for this purpose.

  Efficiency issues in reasoning with query predicates for A*Cache are discussed in Section 4.2.

- **Notification Processor**

  The Notification Processor handles notification messages from the server and is responsible for propagating the effect of each notification on the cache. First, it invokes the cache containment reasoning system to determine whether the current cache contents are actually affected by the update notification. This step is required since notification by the server is liberal and involves network delays, so a cached predicate may have been flushed from the client in the meantime due to the replacement of query results in 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. The actions taken in updating cached data depend on the maintenance policy in effect and on the contents of the notification message. A discussion of the design choices available for cache maintenance appears in Section 3.3.4. In case of a conflict with the current transaction, serializability considerations may require that the transaction be rolled back (Section 3.3.3).

- **Query Execution Engine**

  Transactions may request data retrieval and update operations operations on the database. 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. It is in effect a 'lightweight' client-side database with basic capabilities for transaction execution.

  The A*Cache scheme does not require that all cached data be in main memory. Thus, efficient local evaluation of frequent queries involving joins or many tuples may require that appropriate indexes be constructed locally at a client, for either main memory or secondary storage. These local access paths will depend on data usage at individual clients, and may be different from those in place at the server. Such indexes defined at the client should be taken into account by the client-side query execution engine to improve the efficiency of local data access.

  The execution engine at the client only needs to handle retrieval and update of the cached data. Operations on the database meta-data, such as altering the schema of a relation through a DDL command, must be routed to the server.

- **Space Manager**

  It is the responsibility of the Space Manager to control the storage of new query results in the local data store, and to implement a cache replacement policy when the cache is full. It decides whether a new query result should be cached, and how to purge query results. Advanced functionality may include defining local access paths on the cached data to speed up query evaluation and cache maintenance.

  Cache replacement in A*Cache is done through a predicate-based algorithm, which works in conjunction with a reference counting scheme for individual data items. Algorithms for space management in A*Cache appear in Section 3.3.6.

3.2.2 Server-Side Components

In addition to the cache manager at the client site, special facilities also exist at the server for the purpose of maintaining the client caches. These server-side modules are depicted in Figure 3.1, and are described below:

- **Client Subscription Manager**

  The set of predicates cached by a client is recorded as its *subscription* at the server. The management of the client subscriptions is handled by the server-side Client Subscription Manager. Whenever a query is executed at the server, the server assumes by default that the query will be cached by the client. Accordingly, a query predicate for the cached result is inserted in the subscription of the associated client. This step must be completed before the query result is transmitted to the client, so that its cache is notified of all relevant updates. This module also handles client requests for modifying its subscription, such as deleting a predicate that has been flushed from its cache.

- **Commit Verifier**

  The commit process at the server must be enhanced to support serializability of transactions that evaluate queries locally at the client. As described in Section 3.3.3, this function may involve a 'handshake' 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 on commit requests from client transactions.

- **Notifier**

  An update propagation system known as the Notifier operates at the server site, and communicates with the cache managers at clients to maintain the data in their caches. The notifier is triggered whenever a transaction commits updates on the database, and it uses the cache descriptions recorded by the Client Subscription Manager to determine which clients are affected by the updates made by the transaction. Techniques proposed for incremental maintenance of materialized views [Ceri91, Gupt95] are applicable in this regard. As discussed in Section 4.1, the notification scheme follows a 'liberal' policy in general, so that each A*Cache client may receive some irrelevant notifications, but is guaranteed to receive all relevant ones.

Apart from the above tasks, the server must also receive and handle updates that were initially performed on the client cache. Server-side handling of local updates is described in detail in Section 3.3.3 below, where we discuss the concurrency control protocol and execution scheme for transactions.

3.3 Design Issues and Trade-offs in A*Cache

We now explore the various issues in designing an A*Cache system, and discuss the performance trade-offs of implementation choices. The discussion below is in terms of tuples in relational schemas, but the ideas can be extended to general data types.

3.3.1 Form of Cached Data

An important question is the format of the data stored in the cache. For reasons of efficiency, the following scheme is chosen for A*Cache.

In A*Cache, cached data has the same schema as the database, so that each relation stored in the cache has the same set of attributes as the associated base relation in the database. User queries are *augmented* (see Section 4.4) to include key attributes of all relations in the query, in case they were originally projected out. Non-key attributes projected out in a query are stored as empty or null in the cached copy of the relation; these attributes do not appear in the cache description, and therefore, their values are not used in local query evaluation.

The cached relations may thus contain a subset of the base relation tuples, and a cached tuple may also be a sub-tuple of a base tuple, if some attributes were projected out. For selection and projection queries, new tuples in a result set are merged into the appropriate local table based on their primary keys, so that a local table is both a tuple-wise and an attribute-wise union of the tuples belonging to the cached predicates. Any functions that are applied to the selected items in a query are assumed to be computed locally after the base data is fetched.

For join queries, there are two options: (1) the join may be performed at the server, and the result tuples split up into individual relations at the client for the purpose of caching, or (2) the server could be requested for the semi-join results instead, and the join performed at the client. The second approach avoids the denormalization of information that is common in joins, and the resulting increase in network traffic, but necessitates additional local processing of the user query. Cost analysis of such a scheme involving *relation fragments* was investigated in [Lee90] for data transmission in the Penguin view-object system.

The above scheme aims to reduce storage and maintenance costs for cached data. It can introduce an extra overhead for join queries, and for selection queries with aggregate functions, by fetching base data and including the keys of all relations participating in the join. However, since each cached tuple includes the key attributes, duplication of tuples is avoided. Update propagation is also simplified, since tuples can be identified and modified individually. A single tuple that belongs to two or more cached results does not have to be updated multiple times per notification, since it is not stored in duplicate. In contrast, a collection of materialized views stored in the form of individual query results can potentially have much duplication, and may also require that each query result be maintained separately.

Besides primary keys, an auxiliary reference count is also maintained for each tuple in the cache to count the number of different cached queries that the tuple belongs to. As discussed in Section 3.3.6, reference counts are used for the purpose of predicate-based space reclamation from the cache — a tuple may be purged only when its reference count is zero.

Example of Cache Storage Scheme

We illustrate the cache storage scheme for different query types through the following example.

Consider the BUG database introduced in Chapter 1. Let us assume there is another relation PRODUCT defined as PRODUCT(Product#, Name, Manager), where each row specifies the number, name, and manager of a given product.

Figure 3.2 shows the instances of the BUG and PRODUCT relations that are used in this example.

We use Q_i^j to denote the ith query cached by client j. Suppose that there are three clients, $C1$, $C2$, and $C3$, that have cached the results of the queries below:

Q_1^1 :

```
SELECT  *
  FROM  BUG
 WHERE  Product# = 100  OR  Product# = 300;
```

Q_1^2 :

```
SELECT  Bug#, Customer, Priority, LogDate, Description
  FROM  BUG
 WHERE  Product# = 100  AND  Status = 'Open';
```

Q_2^2 :

```
SELECT  Bug#, Product#, Description, Status
  FROM  BUG
 WHERE  Customer = 'Cust2';
```

Q_3^2 :

```
SELECT Bug#, Product#, Name, Customer, Priority, LogDate,
        Description, Status
  FROM  BUG, PRODUCT
 WHERE  Customer = 'Cust2'  AND  Status = 'Open'
   AND  BUG.Product# = PRODUCT.Product#;
```

Figures 3.3, 3.4, and 3.5 show the cache contents for each of the three clients, along with the reference counts of the cached tuples. Note that the 4th tuple in the cache of client $C2$ has a reference count of 2, since it belongs to both the queries Q_1^2 and Q_2^2. Data fetched by these two queries are merged together into the same cached copy of the BUG relation, using the key Bug# to identify the common tuple uniquely.

BUG

Bug#	***Product#***	***Customer***	***Priority***	***LogDate***	***Description***	***Status***
001	100	Cust1	Low	Jan 10, 97	xxxxxxxxx	Open
002	300	Cust2	High	Feb 14, 97	xxxxxxx	Closed
003	200	Cust1	Medium	Feb 21, 97	xxxxxxx	Open
004	100	Cust2	Low	Mar 5, 97	xxxx xxxx	Open
005	200	Cust2	High	Apr 5, 97	xxxxxxx	Closed
006	200	Cust1	Medium	Apr 15, 97	xxxxx xxx	Open

PRODUCT

Product#	***Name***	***Manager***
100	Product A	Amy
200	Product B	Ben
300	Product C	Chuck

Figure 3.2: *Example Instances of the BUG and PRODUCT Relations*

Cached Partial Copy of BUG

	Bug#	Product#	Customer	Priority	LogDate	Description	Status	Ref Cnt
Q11	001	100	Cust1	Low	Jan 10, 97	xxxxxxxxx	Open	1
Q11	002	300	Cust2	High	Feb 14, 97	xxxxxxx	Closed	1
Q11	004	100	Cust2	Low	Mar 5, 97	xxxx xxxx	Open	1

Figure 3.3: *Data Cached at Client 1*

Cached Partial Copy of BUG

	Bug#	Product#	Customer	Priority	LogDate	Description	Status	Ref Cnt
Q12	001		Cust1	Low	Jan 10, 97	xxxxxxxxx		1
Q22	002	300				xxxxxxx	Closed	1
Q22	003	200				xxxxxxx	Open	1
Q12, Q22	004	100	Cust2	Low	Mar 5, 97	xxxx xxxx	Open	2
Q22	005	200				xxxxxxx	Closed	1

Figure 3.4: *Data Cached at Client 2*

Cached Partial Copy of BUG

	Bug#	Product#	Customer	Priority	LogDate	Description	Status	Ref Cnt
Q31	004	100	Cust2	Low	Mar 5, 97	xxxx xxxx	Open	1

Cached Partial Copy of PRODUCT

	Product#	Name	Manager	Ref Cnt
Q31	100	Product A		1

Figure 3.5: *Data Cached at Client 3*

3.3.2 Class of Queries

Transactions may submit requests for executing queries, and for data insertion, deletion, and update. Queries may involve one or more relations, while the other commands modify data only in a single relation. We do not consider updates to multiple relations through join views, since ambiguity issues may arise in their interpretation [Kell85].

We consider queries that specify their target set of objects using *query predicates*, as in the `WHERE` clause of a `SELECT-FROM-WHERE` SQL query [ANSI92]. We allow general queries, such as a `SELECT-PROJECT-JOIN` operation over one or more relations, with the restriction that the keys of all relations participating in a query must be included in the query result. A user query that does not satisfy this constraint may be augmented (see Section 4.4) by a client to retrieve these keys from the server. Selection criteria in queries may contain negated terms, such as `(Status` $\neq$ `'Open')`. However, we do not consider queries involving the difference operator; they can be supported in A*Cache at the expense of more complicated reasoning in cache containment and maintenance.

User-defined or built-in aggregate functions, such as `max()` or `avg()` on a selected attribute, are not considered here for simplicity. We assume that base data is fetched for such queries, with the client applying the function locally to generate the final result. If any aggregate function appears in a query predicate and that query result is cached locally, then extended containment reasoning and incremental maintenance of aggregate views [Sriv96] must be used for its reuse and refresh. Issues in invoking stored procedures, stored functions, and triggers are also not examined in detail in this dissertation; here, we make the simplifying assumption that these operations are executed exclusively at the server, and not on data cached at the client sites. This assumption could be relaxed and stored procedures could be executed at the client, by enhancing the concurrency control and notification procedures to optimistically handle conflicting changes to the data and to the definition of the stored procedures themselves.

Query predicates specified as above are classified as *point* query, *range* query, or *join* query predicates. A point query predicate specifies a unique tuple (that may or may not exist) in a single relation, by selecting a single value for each attribute in its primary key, and possibly values for other non-key attributes as well. For our BUG example, `(Bug# = 001 AND Status = 'Open')` is a point query that returns the tuple for bug number 001 for the database instance shown in Figure 3.2, and `(Bug# = 1000)` is a point query that has no associated tuple. Point query predicates arise frequently during navigation among tuples of different relations through joins based on foreign key references, e.g., a query about a tuple in the relation PRODUCT based on the matching value in the foreign key `Product#` of a bug.

In contrast, a range query over a relation specifies either a single value or a value range for one or more attributes, and in general has zero, one, or more tuples in its target set. For example, (100 $\leq$ `Product#` $\leq$ 200) is a valid range query predicate on the BUG relation; another example is the query predicate `(Status = 'Open')`. Note that a point query is a special case of a range query; we distinguish between the two only because they are processed somewhat differently at the implementation level for reasons of efficiency.

A query predicate with a join condition, e.g., `(BUG.Product# = PRODUCT.Product#)`, may in general have one or more attributes specified in terms of the join attributes of other relations, and can involve one or more relations, possibly multiple times.

3.3.3 Concurrency Control and Execution of Transactions

In order to define the steps in executing a transaction, a concurrency control policy must first be adopted. Several different forms of concurrency control can be employed within our caching framework, possibly varying by client or even by transaction depending on the data consistency requirements. Below, we review some terminology for the levels of *optimism* in concurrency control [Bern87, Fran93]. A mechanism for concurrency control is called:

- *Pessimistic*, if it *prevents* the violation of the specified isolation constraints; lock-based concurrency control systems are an example of this approach.
- *Optimistic*, if it *detects* the violation of the specified constraints, at some time (possibly at commit) after the constraints have been violated. Multi-version or timestamp-based concurrency control are examples of optimistic algorithms.
- *Semi-optimistic*, if it is a neither purely pessimistic nor a purely optimistic scheme. For example, using update notifications from the server in conjunction with an optimistic policy makes the behavior semi-optimistic, since a transaction may abort before it reaches its commit point, if notified of conflicting updates that were committed at the server by another transaction.

In this thesis, we use the following semi-optimistic protocol for execution of transactions. It attempts to minimize unnecessary aborts of transactions while reducing communication with the server, and is suitable for systems without overly contentious data sharing among clients. The scheme is an extension of the concept of notify locks algorithm studied in [Wilk90] in the context of identity-based caching.

We assume that the server supports lock-based serializability [Gray93b] of transactions when there are no local caches in the system. It can be shown that if transactions are serializable in the original database, they will remain serializable in the presence of A*Cache using this concurrency control scheme (see discussion in Section 5.3 on this issue). If the original database provides protection against phantoms (e.g., by locking an index or key range), the same behavior carries over to this scheme. In fact, phantom protection is provided for all locally cached predicates in A*Cache because of predicate-based notification. We also adopt the isolation semantics that uncommitted updates made by an 'in-flight' transaction should be visible to itself as it executes, both at the server site and at the client. This isolation semantics is the standard adopted by ANSI SQL [ANSI92].

The client logic for execution of transactions is represented in the flowchart of Figure 3.6, and is described below.

Whenever the data required for a query or update is locally available, a client optimistically assumes that its cache is up-to-date — the transaction operates on local copies of tuples, and locks are not obtained immediately from the server but only recorded locally at the client. If the data is not available locally, the request is submitted to the server to fetch it. A request for remote query execution is accompanied by any local (uncommitted) updates of which the server has not yet been informed. Tuples accessed by the remote query and by the (uncommitted) updates are read-locked and write-locked respectively in the usual two-phase manner at the server during remote fetches. The uncommitted updates are also recorded as such by the server, and made visible to the remote query as it executes.

Transmission of local updates along with remote queries is necessary since a query within a particular transaction must be able to see the effects of all (as yet uncommitted) updates made by that transaction when the query is evaluated at the remote server site. This is required by our above assumption of the semantics of in-flight transactions.

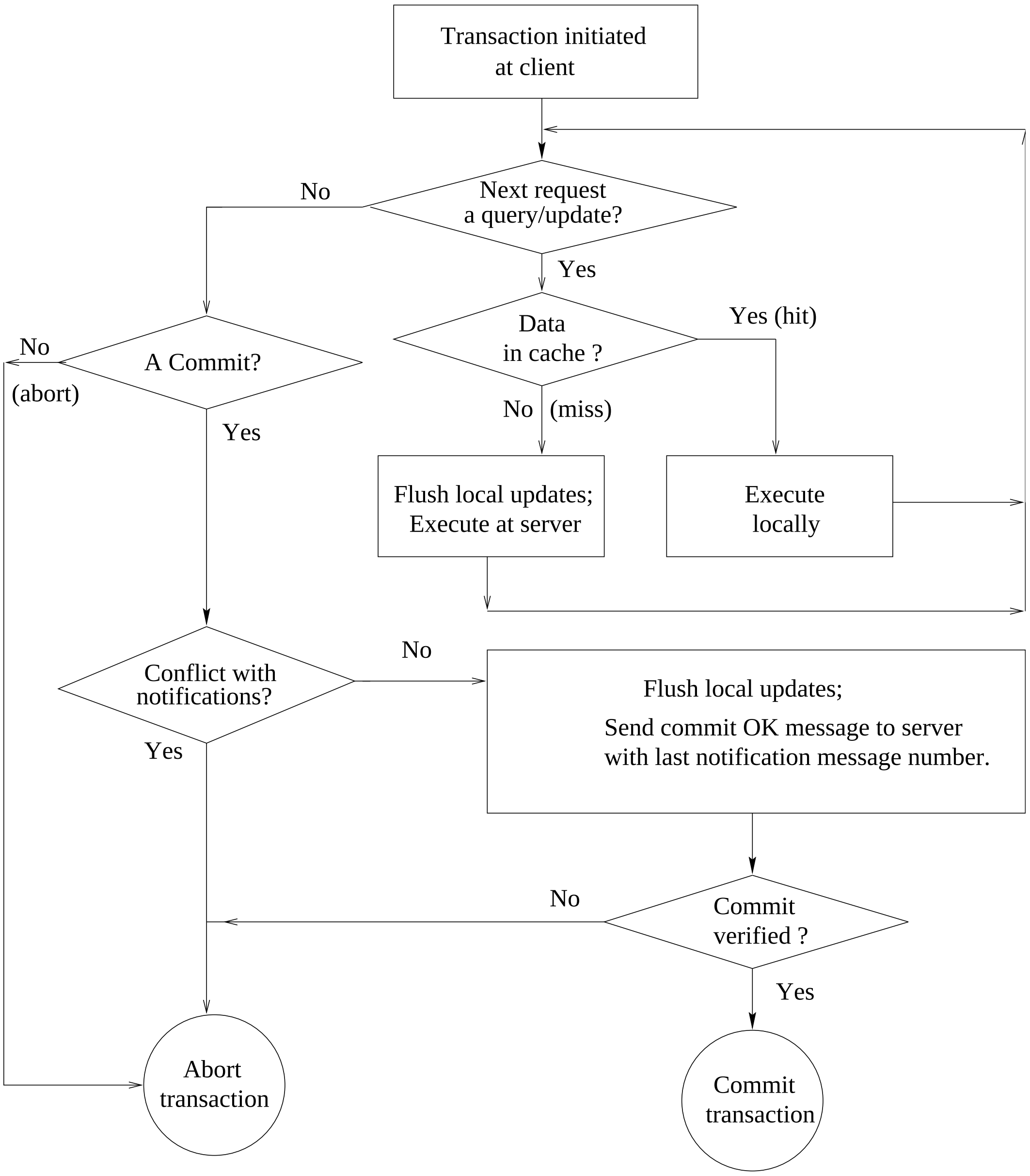


Figure 3.6: *Transaction Execution Logic at Client*

A remote query submission may also be accompanied by *deferred* read-lock requests for tuples read locally since the last communication with the server. Such locking can help reduce aborts of transactions due to concurrent conflicting updates by other clients and subsequent notification, without incurring much extra cost (since they are "piggy-backed" on remote requests). Note that these lock requests are only for those tuples that have been accessed via the local cache and not through a remote fetch within the transaction.

A commit at the client sends all remaining updates (and possibly any deferred read-lock requests) to the server. Observe that if a tuple was first read or written locally, and subsequently locked at the server during a remote fetch or upon a commit request, then there is a window of time between locally accessing the tuple and acquiring a lock on it at the server where it may have been modified by other transactions. In order to prevent missing conflicts due to network delays, 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. As described in [Wilk90], this checking can be done by assigning a sequential message number to every notification sent from the server to a client. The Commit Verifier at the server initiates a handshake with the client to ensure all applicable notifications have been examined by the client, before the commit is declared to be successful. More details on this concurrency control protocol, including a flowchart of the extra commit verification step, appear in Section 5.2.6.

Notification messages for update propagation are issued by the notifier at the server upon each successful commit of an update transaction. These messages may abort the current transaction at a notified client, if the data sets it read and/or wrote locally conflict with the updates made at the central database.

The above protocol thus handles cache hits and misses differently — for hits, an incremental notification-based semi-optimistic scheme is employed, whereas normal two-phase locking is done at the server for all cache misses. The reason for the difference is that a cache miss always implies communication with the server, which is utilized also to lock the tuples appropriately.

When a transaction commits or aborts, all locks held by it are released at the server and at the client. However, data fetched by the transaction need not be purged upon commit or abort, and may be retained in the cache for inter-transaction reuse (after rolling back any uncommitted updates). For correct maintenance of the cache, the server must be kept informed of all query predicates cached by the client past a transaction boundary, i.e., past a commit or abort. In effect, these predicates act as predicate-based notify locks, and are also used for the purpose of update propagation. Note that the cache description at the client as perceived by a transaction may need to be modified, since the data it updates could alter its view of the cache. Details of the actions in maintaining the client cache description appear in Chapter 5, where a correctness proof of the above concurrency control protocol is also presented.

Other Options for Concurrency Control

It is possible to use other concurrency control schemes that are either more or less optimistic than the scheme presented above. For example, the level of optimism can be increased by not obtaining any locks at all for local reads, not even deferred ones. The notification and commit verification process would still detect conflicts, possibly at a later point in the execution of a transaction.

For environments with contentious data sharing, a more conservative concurrency control scheme based on avoidance of conflicts can be used. One alternative is to disallow local updates. All update requests must then go to the server, which sets write locks on updated tuples, reducing the chances of conflicting updates. Stricter consistency control could also be enforced for local reads on the cached data. For example, a query need not be re-executed at the server if the result is locally available, but the objects involved in answering the query could be locked at the server. It is possible in A*Cache to do such locking, since keys or identifiers of all cached items are available. Predicate-based notification can still be used to support incremental maintenance of the cached data.

Notifications could be also sent to clients at different times. In this thesis, we have assumed that the notification is performed as transactions commit updates at the database. However, generation of notifications is not considered to be part of an update transaction; each update transaction simply triggers the separate notifier process to inspect and process its updates. This method ensures timely propagation of updates without delaying update transactions, while attempting to minimize unnecessary aborts of local transactions. The timing of generating notifications could be varied depending on the specific requirements of a system.

In the above discussion, we assumed that the server enforces fully serializable concurrency control, i.e., 3^o isolation [Gray93b]. Most commercial systems also support lower degrees of consistency [Bere95], since performance implications cause many practical applications to run with reduced consistency levels. For example, 2^o provides reads of committed data only, while 3^o is fully serializable with phantom protection [Gray93b]. In Section 5.3, we discuss the effect of reduced isolation levels on A*Cache operation.

3.3.4 Cache Maintenance Issues and Choices

There are several dimensions to the problem of cache maintenance, two major ones being the method of maintenance, e.g., the question of cache refresh versus invalidation, and the contents of notification messages. We discuss design issues for these two aspects in this and the following subsection. The two issues are not independent however, since a choice of one affects the other, as discussed below.

Refresh or Invalidate?

Several alternatives exist for maintaining the cached data as transactions commit updates at the server, including: (1) automatic refresh upon notification, (2) refresh upon demand by a subsequent query, (3) invalidation of cached data and predicates upon notification, and (4) automatic age-based expiration of cached predicates. Both automatic and by-demand refresh procedures may either be recomputations or incremental, i.e., performed either by re-execution of the cached query or by differential maintenance methods. Which method performs best depends very much on the usage characteristics of the database, such as the volume and nature of updates, pattern of local queries, and constraints on query response times.

In A*Cache, the maintenance method is allowed to be different for each client, and also for different query results cached at a single client. For example, a client may request that results of frequently-posed queries be automatically refreshed, and may choose to invalidate upon update a random query result. Cached query results may also have their method of maintenance upgraded or downgraded as data access patterns at a client change over time. This flexibility in the cache maintenance policy provides the ability to adapt to changing data usage patterns and varying system loads, which is important for dynamic caching schemes.

Applying Propagated Updates

For correct operation, the state of the cache must be 'transaction-consistent' at any given instant. That is, it must possess the all-or-nothing effect of data modifications made by transactions. Therefore, all updates committed by a single transaction that are possibly relevant for a cache C must be sent across the network 'batched' by transaction, and must be applied to the cache C such that all of them are made visible to a client transaction at the same instant (effectively), or that none of them are visible. That is, maintenance operations to propagate updates must obey transactional semantics at the client site, and must be appropriately grouped into *maintenance transactions*.

Actions taken in a maintenance transaction may vary depending on the chosen cache maintenance policy. It may involve insertion, update, or deletion of data, and/or modification of the cache description (details of these steps appear in Chapter 5). Maintenance of queries involving join predicates may require quite complex reasoning, and as demonstrated in the example below, auxiliary information might be needed from the server to refresh the join result locally. All cache maintenance operations, such as obtaining auxiliary information from the server to refresh cached results or altering the cache description, that result from the propagation of updates of a single transaction must be performed as part of the maintenance transaction itself.

Example: Maintaining Cached Joins

Consider a client that has cached the join result `BUG` $\bowtie$ `PRODUCT` through the join query predicate `(BUG.Product# = PRODUCT.Product#)`. Now suppose that a new bug for product number 300 is inserted at the server.

The information needed at the client to refresh its cache is dependent on: (1) the differential join involving the new tuple, i.e., the join of the new `BUG` tuple with the `PRODUCT` tuple having `Product# = 300`, and (2) whether this `PRODUCT` tuple is available locally. There are several ways in which this case can be handled:

- The client could invalidate the join result. It is possible for the invalidation to be temporary, with a differential refresh if the same query is posed subsequently.
- If the `PRODUCT` tuple for product number 300 is not available at the client, then the 'differential' join of the new tuple may either be computed at the server and sent

along with the notification, or it might be requested by the client after receipt and analysis of the notification message. Refreshing the join result is not possible in this case using just the new `BUG` tuple.

- Alternatively, if the exact or conservative cache description shows that the `PRODUCT` tuple for product number 300 is locally available at the client (possibly from other cached query results involving the `PRODUCT` relation), then insertion of only the new `BUG` tuple in the cache is sufficient to refresh the join result.

As shown in the above example, refreshing cached data may require that notification messages carry not only the updated data, but also other auxiliary information that is required for the refresh, but is absent at the client. This observation leads us to the topic of the following subsection.

3.3.5 Contents of a Notification Message

Notification messages from the server could contain a variety of information including: (1) only the primary keys or identifiers of updated tuples, or (2) the updated data, or (3) update commands only, or (4) not only the modified data and update commands, but also other auxiliary information required to refresh the cache (as in the example on cached joins above). The question obviously arises as to which notification method is to be chosen for any given scenario.

Instead of a fixed scheme, contents of notification messages in A*Cache are allowed to vary, depending on the type of cached queries, the nature of the update, and also on the cache maintenance policy in effect.

Sending only the identifiers of updated tuples is in general not adequate to maintain the contents of an associative cache. This scheme works only for notification on data deletion, and for invalidation-based maintenance. If a tuple identified by its primary key is deleted, then it may simply be removed from the cache. Cached predicates will still remain valid, but the transaction executing at the client may abort if it had read the tuple earlier.

If the update was not a delete, sending identifiers only of updated data may only make sense in the rather degenerate case where all cached predicates are *point* queries (Section 3.3.2), i.e., data is fetched in based on primary keys which serve as unique identifiers.

In our BUG example, such an application would be one that fetches individual bugs by their (unique) bug number. In this case, notification that a bug has changed can invalidate the tuple and its point query predicate, and purge them from the cache. However, a range query predicate, such as `(Status = 'Open')`, cannot be refreshed using simply the bug number of a bug. Therefore, the only option upon receiving such a notification is to invalidate *all* cached predicates that are not point queries.

When the number of updated tuples is not too large, then the updated data can be sent in the notification. Each tuple must be compared with the predicates in the cache description, and rejected if it does not satisfy any predicates in it (Section 5.2). Otherwise, it must be inserted into the cache, possibly overwriting a previous copy of the tuple (i.e., one having the same primary key).

Sending all updated data may be too expensive when many tuples are affected by the update. In such cases, the question arises whether notification could be done in terms of one or more update commands to be executed on the local cache. In many cases, incremental refresh of the cache cannot be supported using solely the update command and the locally cached data. Only invalidation-based cache maintenance is possible in such cases, a cached predicate being invalidated conservatively whenever it intersects an updated region. However, in some special cases, sending only the update command to the client is sufficient to refresh its cache, and this strategy can minimize network traffic and notification processing costs. We illustrate this point in the following example.

Consider a client C that caches all and only those bugs in product number 100. Suppose that the priority of bugs reported by a certain customer for product number 100 is updated elsewhere using the command:

```
UPDATE  BUG
   SET  Priority = 'High'
 WHERE  Customer = 'Cust2'  AND  Product# = 100;
```

One option is to transmit all updated bugs in the notification message, so that the cache at client C can be refreshed using the individual tuples in the notification. However, the set of tuples on which the update is to be applied is already available at the client, and the update command can instead be directly executed on its cache to refresh it.

As shown in the above example, a requirement for cache refresh using only the update command is that the set of tuples on which the update is applied must be currently cached at the client; in other words, the update must be *autonomously computable* [Blak89] at the client site from its set of cached results. In A*Cache, this condition can be detected using containment reasoning on the cache description and the predicate in the update command. A point to note is that detection of such autonomously computable updates must be based on the exact or conservative client cache description. A liberal approximation may falsely imply that the client has data that is not actually present in its cache.

3.3.6 Effective Management of Space

In this section, we discuss the issues in managing the space of data locally cached at the client, including the space reclamation policy.

To Cache or Not to Cache

When a new query result is fetched from the server, the space manager is faced with a choice of whether or not to cache the result set past the termination point of the current transaction. In order to estimate the long-term benefits of caching the new predicate and tuples, it must perform an approximate cost-benefit analysis similar to that outlined in [Ston90]. The A*Cache algorithm is LRU-based, but the analysis is in terms of predicates and not individual tuples. The algorithm is sketched below, and takes into account the sizes of cached result sets and their anticipated future usage patterns. Note that one-time costs of updating the cache descriptions, storing the tuples in the cache, etc., are not taken into account, since they do not affect the long-term benefits.

The following parameters play a role in the cost-benefit analysis:

S_i: Size of the result set for the ith cached predicate P_i

F_i: Cost of fetching tuples satisfying P_i from the server

R_i: Cost of accessing and reading the tuples in P_i if cached locally

U_i: Cost of maintaining the tuples in P_i if cached locally

r_i: Frequency of usage of predicate P_i at the client

u_i: Frequency of updates by other clients that affect the tuples in predicate P_i

The expected cost per unit time, T_i, of the caching the tuples in ith predicate P_i is:

$$T_i = \begin{cases} r_i R_i + u_i U_i & \text{if } P_i \text{ is cached} \\ r_i F_i & \text{if } P_i \text{ is not cached} \end{cases}$$

Thus, the expected benefit B_i of caching predicate P_i locally is:

$$B_i = r_i F_i - (r_i R_i + u_i U_i).$$

The above analysis represents a client's view of the costs and benefits of caching a predicate. A more detailed cost model that also considers cost parameters at the server and other modules in the system, such as the network, may be developed along similar lines. This extension is a subject of future research.

Reclaiming Space

When the cache is full and space needs to be reclaimed, predicates and tuples are flushed based on a predicate ranking algorithm and tuple reference counts. The *rank* N_i of predicate P_i is defined as the benefit per unit size:

$$N_i = B_i / S_i.$$

Cached predicates may be sorted in descending order of their ranks. At any point, only those predicates in the cache that have ranks equal to or higher than a certain cutoff rank may be kept in the cache.

Once a predicate P is chosen for elimination, a delete command with query predicate P can be executed on the cache to determine which tuples are candidates for deletion. Not all tuples satisfying P can be deleted however, since a tuple may be purged from the cache only if it does not satisfy *any* cached predicate for the relation. A reference count is associated with each cached tuple for this purpose, and it indicates the number of cached predicates that are currently satisfied by the tuple. A tuple may be flushed from the cache only when its reference count is 0.

Effect on Cache Descriptions

Reclaiming space based on predicate usage as discussed above will result in predicates being purged from the client cache description. The conservative description at the client *must* be changed to account for purging of tuples and predicates, whereas such a change *may* optionally be reflected on the client subscription at the server. Presence of extra predicates at the server can at most lead to an increase in irrelevant notifications to the client. Hence, updating the client subscription in this case can be treated as a low priority task to be performed at times of light load, and preferably before too many notifications are issued. This update can be made by sending a 'purge predicate' message to the Client Subscription Manager at the server site.

Thus, only asynchronous coordination with the server is required for purging of predicates from a cache. Incorrect operation can never result as long as the client description at the server *subsumes* the cache description at the client. This notion of subsumption of the conservative cache description by the liberal cache description is formalized in Section 4.1.

3.4 Summary of the A*Cache Scheme

In this chapter, we have introduced A*Cache, a client-side data caching and maintenance scheme based on query predicates. A major advantage of A*Cache is that it supports predicate-based access to the cache contents, allowing local execution of associative queries and effective reuse of the cached data. Increased utilization of client resources, less network traffic, and better scalability are some of the expected benefits of A*Cache over identifier-based caching schemes. In order to determine the practicality of using A*Cache in a dynamic and 'real-time' caching environment, we have examined various design issues and performance trade-offs in A*Cache operation. Design decisions made in A*Cache and their rationale have been explained, and the different choices available for an implementation have been discussed.

Chapter 4

Optimization Techniques for A*Cache

Examination and maintenance of the cache via predicate descriptions requires determining containment and satisfiability of predicates, and concerns about overhead and scalability may naturally arise over the cost of complexity when reasoning with large numbers of predicates in a real-time dynamic caching environment. However, there are also new opportunities for optimization. To reduce the costs in predicate-based reuse and maintenance of the cache, efficient retrieval and approximation techniques can be devised. We describe several such optimization schemes below.

4.1 Approximate Cache Descriptions

To reduce the complexity of reasoning with predicates, approximations can be applied to the cache descriptions at the client and server sites.

The cache description at the client site, which is used for determining cache containment, need not be exact and can be *conservative*. Data claimed to be in the client cache must actually be present in it, so that locally evaluated queries do not produce incomplete results. However, the cache containment reasoning may be more conservative than necessary. Thinking that an object is not in the cache when in fact it is will result in re-fetching the object from the remote information source; the approach may be inefficient, but is not

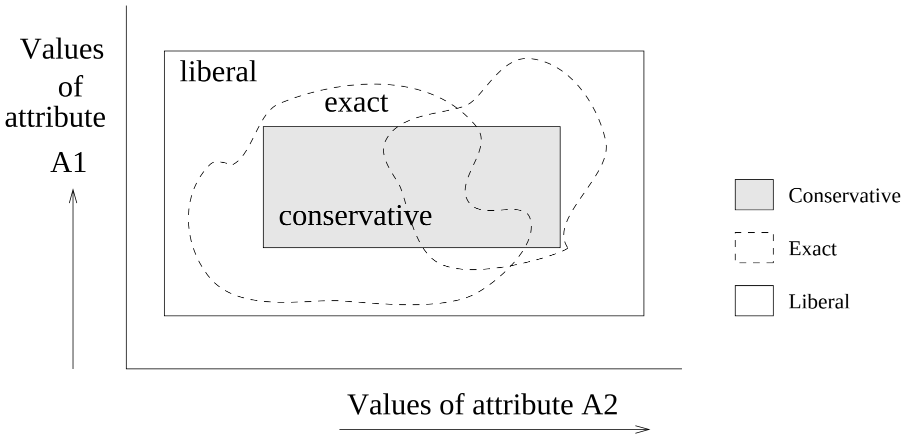


Figure 4.1: *Exact, Conservative, and Liberal Cache Descriptions*

incorrect. An example of conservative approximation of a query predicate is its simplification by discarding a disjunctive condition. Such *subset* approximations are guaranteed to always produce complete answers.

The server maintains a consolidated subscription of all client caches, and uses it to generate notifications as transactions commit updates. Since the server is a shared resource of many clients, each of which may be caching tens or even hundreds of query results, it is crucial to control the complexity of issuing notifications. The server may be able to use approximate descriptions of client caches for this purpose. However, in contrast to conservative cache descriptions at the client, the approximation used by the server may only be *liberal.* That is, occasionally notifying a client of an irrelevant update is not a critical problem, but failure to notify that a cached object has changed could result in erroneous behavior. A liberal description is expected to be a simpler *superset* of the actual one, but generates all necessary notifications.

A simple example of liberal approximation is ignoring projections in a cached query. Attributes projected out in a query *must not* be part of a conservative description, but *may* optionally be part of the liberal one. Thus, a client could receive updates involving the projected attributes, even though it is not caching them.

Figure 4.1 shows a pictorial representation of the exact, conservative, and liberal cache descriptions for cached query predicates on a single relation R with two attributes $A1$ and $A2$.

We formalize our terminology below using the usual predicate calculus notation. Suppose that there are n clients in the client-server system, with C_i representing the ith client, $1 \leq i \leq n$. Let Q_i^E, $Q_i^E \geq 0$, be the actual number of query predicates whose results are cached at client C_i. The superscript E in Q_i^E denotes *exact*.

We denote by P_{ij}^E the query predicate corresponding to the jth query result cached at client C_i. The superscript E in P_{ij}^E represents the fact that these are the exact forms of cached query predicates, without any approximations applied. Other information related to a query may be associated with its query predicate, e.g., the list of *visible* attributes retained after a projection operation on the selected tuples.

Definition 1: An *exact cache description* ECD_i for the ith client C_i is defined to be the set of exact query predicates P_{ij}^E corresponding to all query results cached at the client:

$$ECD_i = \{P_{ij}^E \mid 1 \leq j \leq Q_i^E\}.$$

Definition 2: A *conservative cache description* CCD_i for the ith client C_i is a collection of zero or more predicates P_{ik}^C such that the union of these predicates is contained in the union of the predicates in the exact cache description ECD_i for the client. Let Q_i^C denote the number of predicates in CCD_i. Formally,

$$CCD_i = \{P_{ik}^C \mid 1 \leq k \leq Q_i^C\},$$

where $\bigcup_{1 \leq k \leq Q_i^C} P_{ik}^C \in CCD_i \implies \bigcup_{1 \leq j \leq Q_i^E} P_{ij}^E \in ECD_i$.

The superscript C in P_{ik}^C and Q_i^C stands for *conservative*. CCD_i may also be thought of as the *client* cache description, since it is used only by client C_i. The symbol $\implies$ denotes the material implication operator, and in the context of query predicates has the following meaning: if $Q \implies P$, then the result of the query corresponding to predicate Q is contained in and is computable from the result of the query corresponding to predicate P.

One example of conservative approximation of a query predicate is its simplification by discarding a disjunctive condition. For other possible differences between ECD_i and CCD_i, consider some cached `BUG` tuples that were fetched through a number of point queries, as well as through a few range queries. CCD_i may consist only of the range query predicates, whereas ECD_i includes all cached predicates, both point and range. CCD_i is thus simpler than ECD_i, having eliminated point query predicates. The effect of this approximation is that cached results of point queries are not taken into consideration when determining cache containment of range queries, likely speeding up the reasoning process. Therefore, if all `BUG` tuples for product number 100 have been fetched and cached through separate point queries, a range query on the `BUG` relation with query predicate `(Product# = 100)` will result in re-fetching these tuples from the server. Such a remote access is only inefficient and not incorrect, and will not recur if the range query predicate gets cached in CCD_i.

It is important to note that the conservative approximation pertains to the cache description only, and *not* to the cache contents. In the above example, individual `BUG` tuples cached through point queries are still locally available at client C_i. Although these tuples cannot be accessed through CCD_i, they are still present in the cache and can be used to answer point queries locally. As long as the server notifies the client of changes to these tuples, their local use cannot result in erroneous operation. A conventional index based on the primary key `Bug#` of the `BUG` relation may be constructed locally at the client to speed up the processing of point queries.

Definition 3: A *liberal cache description* LCD_i for the ith client C_i is a set of zero or more predicates P^L_{ik} such that the union of these predicates contains the union of the predicates in the exact cache description ECD_i for the client. Let Q^L_i denote the number of predicates in LCD_i. Formally,

$$LCD_i = \{P^L_{ik} \mid 1 \leq k \leq Q^L_i\},$$

where $\bigcup_{1 \leq j \leq Q^E_i} P^E_{ij} \in ECD_i \implies \bigcup_{1 \leq k \leq Q^L_i} P^L_{ik} \in LCD_i$.

By the above definitions, incorrect operation can never result as long as all operations on cache descriptions at all clients and at the server always obey the constraint

$$(\forall i, 1 \leq i \leq n)(CCD_i \subseteq ECD_i \subseteq LCD_i).$$

One example of a liberal approximation occurs when a client is notified of a change to a relation attribute that it does not cache. Another example is the case where a tuple that could possibly affect a cached join result is inserted at the central database. The server may be able to eliminate a tuple that is *unconditionally* irrelevant for the join [Blak89] (i.e., irrelevant independent of the database state); however, determining whether the tuple actually affects the join result for the particular database instance requires more work. The client may in this case be informed of the inserted tuple, and can subsequently take actions based on local conditions at the client site (an example of cached join maintenance is given in Section 3.3.4).

If server notification is over-liberal, e.g., at the coarse granularity of relations, it may result in wasted work at client sites; if too detailed, it may have prohibitive overhead at the server. Hence, it is important to control the *degree* of liberal approximation at the server. Ideally, the server should be able to adapt the degree of approximation according to its current workload. Assuming that the cost of precise update screening is greater at the server than at the clients, notification may be more liberal at times of high server load in order to distribute some of the cache maintenance work to the clients. Lighter loads may allow better filtering of irrelevant updates at the server. Similar load-balancing techniques were suggested for maintaining *quasi-copies* [Alon90] that specify coherency conditions.

4.2 Efficient Containment Reasoning

Since the major motivation for A*Cache is effective reuse of local data, it is important that the process of determining cache containment be intelligent. Answering queries using views has been a focus of recent research [Levy95, Sriv96], and these algorithms can be directly used in the reasoning process. In many cases, it is also possible for the reasoning system to consider semantic information that is specific to an application domain, such as user-defined integrity constraints. The utility of using domain-specific constraints in containment reasoning is illustrated in the example below.

Consider a cached result of the join query `(BUG ⋈ PRODUCT)`, with no attributes projected out. If the client encounters a subsequent query for all `BUG` tuples, the general answer to the query containment question is that only a subset of the required tuples are cached locally. However, if it is known that all bugs must have a valid product number, then there are no *dangling* `BUG` tuples with respect to this join, and all `BUG` tuples appear in the join result. Therefore, the join predicate `(BUG.Product# = PRODUCT.Product#)` may be replaced by the simple predicate `True` on the `BUG` relation, indicating that all bugs are available in

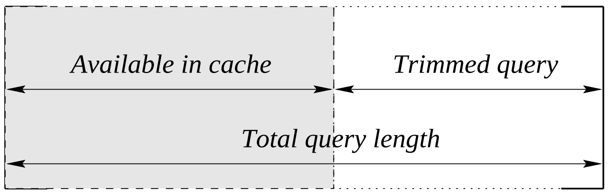


Figure 4.2: *Example of Query Trimming*

the cache. Note however, that all PRODUCT tuples may or may not be available, since not all products have bugs logged against them.

Although using general semantic information may be too complex, simplification of query predicates using such referential integrity and non-NULL constraints on attribute domains can be quite effective. Techniques developed for semantic query optimization [Bert92, King84] are applicable in this regard.

4.3 Query Trimming

Query trimming applies in the case of *partial* cache containment, in which only a part of a query result is available in the cache. In this situation, the server could be requested only for the missing portions of the query result, by pruning the locally available parts from the user query. An example of this scenario is shown in Figure 4.2. Query trimming reduces the data traffic across the network, and can potentially decrease the time required to answer a query.

If the query predicate overlaps multiple predicates in the cache description, then the query may be trimmed in more than one way. It is an optimization decision whether and how to trim the query, involving factors such as cost estimates of evaluating and transmitting the trimmed versus untrimmed result sets, communication costs, and update activity on the cached data. The decision may be left to the server (by annotating as 'optional' parts of a query that are locally available), with the client appropriately skipping or performing the local evaluation step. Some strategies for query trimming have been explored in the context of *BrAID* [Shet91], and for *remainder queries* in semantic caching [Dar96].

4.4 Query Augmentation

Query augmentation is another interesting optimization strategy that can be employed in A*Cache. A query predicate, or the set of attributes projected by a query, can be *augmented* by a client before submission to the server so as to make the query result more suitable for caching.

User-transparent query augmentation to include relation keys is in many cases a viable technique for reducing the long-term costs of maintaining cached query results. One major performance benefit is that data need not be stored in duplicate. Availability of keys implies that query results can be split up into constituent sub-tuples of participating relations, possibly with some non-key attributes projected out, and cached without duplication in local partial copies of original database relations (see Section 3.3.1 for details of the data storage scheme at the client).

Apart from fetching relation keys, query augmentation can be applied to also fetch other attributes that were projected out in the original query, especially if they are not frequently updated. Such augmentation can be useful in data-browsing applications such as a digital library, where queries are often refined successively to display more detailed data. If all attributes of a relation are cached, then a refinement of a previously-posed query can be handled locally without re-executing it at the server.

As another simple example of a situation where query augmentation is appropriate, consider the query predicate `(Product#` $\neq$ `100)` on the relation `PRODUCT`. Suppose that there are 50 tuples in the current instance of the `PRODUCT` relation, and that they are modified infrequently. The query predicate in this case may be augmented to remove the restriction on product number. Thus, the client fetches all 50 `PRODUCT` tuples instead of 49; the cost of transmitting and storing one extra tuple is negligible, as is the maintenance cost in this particular scenario. Savings include simpler containment reasoning at the client and the server. The pre-fetch will also enable the client to locally answer any future queries involving all `PRODUCT` tuples, e.g., the tuples satisfying `(Manager = 'Smith')`.

Possible benefits of query augmentation are: (1) simplified cache descriptions at both the client and the server, (2) reduced costs of storage and maintenance of query results, and (3) local processing of a larger number of future queries, due to pre-fetching of data and the augmented predicate. Overhead due to query augmentation can be significant in some cases. It may cause: (1) an increase in size and response time of the query, and (2) wastage of server and client resources in maintenance and storage of information that might never be referenced by future queries. The cost-benefit analysis of query augmentation involves examining the nature (e.g., selectivity, complexity, and size) of the query predicate, data usage and update patterns, and space availability in the cache.

4.5 Predicate Indexing

The cache description at a client, though conservative, may grow to be quite complex as many query predicates are locally cached. *Predicate indexing* mechanisms [Hans90, Same90] can be used to speed up the examination of predicate descriptions for cache containment reasoning and notification processing. Predicate indexing can also can be employed at the server to facilitate generation of notifications from client subscriptions.

Definition of appropriate predicate indexes is a design decision that depends on the patterns of user queries and updates. R-trees and its extensions are popular indexing schemes [Rous85]. However, n dimensional indexes may incur much overhead, especially when not all attributes are used in query predicates. A simple rule for A*Cache might be to dynamically define a 1-dimensional index on an attribute of a relation whenever cached predicates can be organized efficiently along that dimension.

4.6 Predicate Merging

The problem of handling a large number of predicates in a cache description may arise at both client and server sites. Simplification of cache descriptions through *predicate merging* can help reduce the costs of examining and maintaining cached predicates. In this optimization technique, groups of overlapping or adjacent predicates that form a single contiguous region are merged into a single simplified predicate, thus reducing the total number of predicates in a cache description.

Two or more predicates may be replaced with a single predicate whenever the merged predicate is equal to the union of the spaces covered by the individual predicates. For A*Cache, the cache description at a client can be conservative instead of exact, and therefore

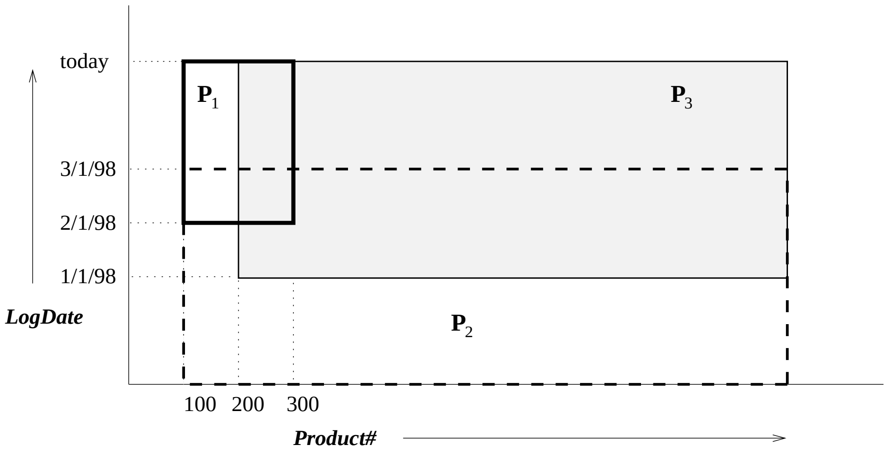


Figure 4.3: *Example of Predicate Merging*

the merged predicate at a client site can be a proper subset of the union of the constituent predicates. On the other hand, the server cache description is allowed to be liberal, and therefore a merged predicate at the server can be a superset of the union of the individual query predicates. The trade-off is that communication between the clients and the server may increase due to conservative or liberal approximations applied to the cache descriptions in the process of merging predicates.

As an example, consider the following cached predicate on the `BUG` relation:

$P1:$ `(100` $\leq$ `Product#` $\leq$ `300) AND (LogDate` $\geq$ `2/1/98).`

If two new predicates

$P2:$ `(Product#` $\geq$ `100) AND (LogDate` $\leq$ `3/1/98),`

and

$P3:$ `(Product#` $\geq$ `200) AND (LogDate` $\geq$ `1/1/98)`

are subsequently added to the cache description, the the union of the three predicates can be reduced to a single equivalent predicate on the `BUG` relation:

$P4 = P_4 = P_1 \cup P_2 \cup P_3 =$ `(Product#` $\geq$ `100).`

Figure 4.3 illustrates the predicate merging scenario in this example.

Well-established algebraic techniques can be applied for merging predicates — two or more predicates can be checked for merge eligibility using distributive, associative, and commutative laws of the boolean and relational operators. However, purely algebraic techniques have exponential complexity, since all possible subsets of the set of cached predicates may have to be considered in determining the applicability of an algebraic rule. To speed up the process of detecting which predicates can be merged, predicate indexing techniques and neighborhood algorithms [Jaro92] in spatial geometry can be employed.

4.6.1 Effect of Merging on Tuple Reference Counts

Let us now consider the effect of predicate merging on tuple reference counts. If there is no overlap among the predicates being merged, no special action is necessary; otherwise, reference counts must be appropriately adjusted for those tuples in the intersection of any two or more predicates being merged. There are two alternative schemes for making this modification:

- Scheme 1. One way of updating the reference counts is to determine for each tuple in the final merged predicate the number of constituent predicates that are satisfied by it. The reference count for such a tuple should be decremented by an amount which is one less than this number, in order to correctly account for predicate overlaps. This scheme involves a one-time overhead at the time of predicate merging, but the reasoning on a per-tuple basis could be expensive when a large number of tuples are involved.

- Scheme 2. The expense incurred in Scheme 1 above may be deemed to be unnecessary in a scenario where predicates are frequently merged but not purged as often, in which case the following alternative scheme may be employed. A *predicate merge history graph* may be maintained for each client cache to record which predicates were merged to produce new predicates. When space needs to be reclaimed by purging a predicate that was previously generated from other predicates, the constituent predicates can be individually deleted, so that each predicate decrements by one the reference counts of those cached tuples that satisfy it. Following our usual rule for reclamation, a tuple can be removed from the cache if its reference count drops to zero during this process. The flushing of a predicate from a *CCD* must also update the predicate merge history graph as necessary.

4.7 Caching Privileges

To control caching of update-intensive shared regions (update 'hotspots'), and to avoid runaway notification costs at the server as the number of clients increases, clients may be granted *caching rights* to a relation or to a part thereof. Denial of caching rights on a region implies that no notification message will be sent to the client when the region is updated, even if the client is caching some tuples in it. The client may reuse such cached data locally with the understanding that the data might be out-of-date. If currency of the data is important, the query should be re-executed at the server.

Caching privileges may be specified statically if the access pattern is known a priori or can be anticipated. If the unit of specification of caching rights is an entire relation, the scheme works in a manner similar to the usual authorization mechanisms for performing a selection or update on a relation. Permission to cache a query result can be checked at the server when a query is submitted by a client, the client being informed of the outcome along with the result set. The right to cache may also be granted on parts of a relation, by defining predicates that specify fixed attribute values or ranges where caching is prohibited for the clients. Processing of such rather detailed caching rights would be approximate, in the sense that the server may denote an entire query result as 'not eligible for caching' whenever there is any intersection of the query space with a region where caching is disallowed for the client.

4.8 Summary of Optimization Strategies

In this chapter, we have discussed several optimizations that can be applied in an A*Cache system. In particular, we have introduced the concept of liberal and conservative cache descriptions for more efficient reasoning with cached predicates. Without compromising the correctness of transaction execution, it is possible for the server to notify liberally and the client to use its cache conservatively. Such approximations are important for performance and scalability when there are many clients or large number of cached predicates in the system. Predicate indexing and merging can be used by both the server and the client to organize and manage cached predicates. Predicate indexing techniques developed in the context of active and spatial databases [Hans90, Sell92, Same90] are also applicable for A*Cache. We suggest query modification techniques such as query trimming and augmentation to alter user queries for greater efficiency of query execution and caching.

Future work includes the investigation and development of further optimization strategies, such as special query optimizers for intelligent query trimming, and sophisticated cache replacement schemes for space management, among other possibilities.

It is important to note that apart from the optimization techniques presented in this chapter, the concurrency control scheme directly affects the performance of the caching system. As discussed in Chapter 3, different levels of optimism may be chosen in the concurrency control protocol to reflect particular data usage and update patterns. For example, in an environment where there are mostly reads and few writes of data, it may be most efficient to choose an optimistic concurrency control scheme, thereby reducing the costs associated with locking. Intensive updates could require a more pessimistic approach, in which the cache is only used for local reads but local writes are not supported, all updates being performed at the server using write locks. In our performance study of A*Cache through simulation (Chapters 7 and 8), we investigate an alternative protocol called A*Cache_Opt, which performs writes at the server for cache misses, and study its performance in comparison to A*Cache.

Chapter 5

Transaction Execution in A*Cache

In this chapter, we describe the execution of transactions in A*Cache, and examine the details of the events that occur in the caching system in response to the different types of requests submitted by client transactions. We also demonstrate that A*Cache transactions that execute in conformance with the given model and that obey the specified concurrency control protocol do not violate serializability constraints. Finally, we consider some consistency issues in cases where the server supports lower non-serializable levels of data isolation.

5.1 Assumptions

The assumptions we make in this thesis about client transactions and update propagation are noted below. Later in this chapter, we discuss the effects of relaxing some of these assumptions.

- **A1.** At most one read-write client transaction T executes at a client site at any given time. We do not consider multiple client transactions utilizing the same A*Cache simultaneously. The A*Cache framework can be extended to accommodate multiple users of the cache, but is not considered here for simplicity.

- **A2.** We assume that the server is lock-based, and that it provides degree 3, i.e., fully serializable, isolation for all transactions on the database. Degree 3 isolation requires that a transaction have no dirty reads of uncommitted data, no lost updates, and additionally that all reads are *repeatable* during the period of its execution [Gray93b].

Read and write locks acquired by a 3^o transaction are two-phase 'long duration' locks, in that they are held until the associated transaction terminates (commits or aborts). Enforcement of repeatable reads at the predicate level requires some variant of *predicate locking*, such as *granular* or *key-range* locks [Gray93b]. A discussion of the effect of reduced isolation levels appears in Section 5.3.4.

- **A3.** A local transaction is started at the client site upon submission of an application program. A remote transaction is also initiated at the server database when the transaction submits a commit request at the client, or earlier if there is a cache miss causing a remote fetch. The remote transaction lasts until the client transaction terminates (commits or aborts), and executes the commands submitted for remote execution.

  Without loss of generality, we assume that the concurrency control algorithm outlined earlier in (Section 3.3.3) is adopted. It is a semi-optimistic scheme that allows local data reads, which may later be found to have seen stale data. Local operations set long duration read and write locks appropriately. Notifications from the server may abort a transaction that read or wrote stale data locally. The commit protocol requires the server to verify with the client that all relevant notifications have been processed before a commit is declared successful (Section 5.2.6). Depending on the outcome of the remote commit, the local transaction at the client either commits or aborts.

- **A4.** For correct operation, the data in the cache must be 'transaction-consistent' at any given instant. That is, it must possess the all-or-nothing effect of data modifications made by transactions. Therefore, all updates committed by a single transaction that are possibly relevant for a cache C must be sent across the network 'batched' by transaction, and must be applied to the cache C such that all of them become visible to a client transaction at the same instant (effectively), or that none of them do. That is, maintenance operations to propagate updates must obey transactional atomic semantics at the client site, and must be appropriately grouped into *maintenance transactions.*

  If the data written by a maintenance transaction conflicts with the read or write sets of a transaction currently executing at the client, then the transaction must be aborted, and possibly restarted.

5.2 Execution of Transactions

Database operations executed by transactions may affect the contents of the central database, as well as the contents and descriptions of local caches. Client-cache contents and predicate descriptions at the client and server sites also change dynamically over time, as query results are cached or purged by the clients. We now consider the steps in executing a transaction in the context of our concurrency control scheme as noted in assumption A3 in Section 5.1 above. The discussion below is with respect to the cache at the ith client C_i, whose cache descriptions at the client and the server are denoted by CCD_i (conservative) and LCD_i (liberal) respectively. The set of client subscriptions at the server is denoted by SCS.

5.2.1 Query Submission

Consider a `SELECT-PROJECT-JOIN` query with predicate Q that is submitted at client C_i. If Q is a point query predicate on a single relation, or can be split up into point queries on several relations, then the tuple(s) satisfying Q may be found using locally defined and maintained indexes on primary keys of the relation(s) cached at C_i. If the tuples are found, and have all selected attributes visible, we use them. If not, we have to determine whether the tuples exist, i.e., whether Q is contained in the scope of the cache, so we treat it as a range query and handle it as described below. Note that it is useful to record that a predicate is cached even when there are no tuples satisfying the predicate, since it can be used to determine locally that the result of a (point or range) query is empty.

If Q is a range or join query predicate, then Q is compared against CCD_i. Three different situations may arise:

- Q is computable from the union of the predicates in CCD_i. In this case, all tuples satisfying Q (if any) are locally accessed in the cache. There is no effect on either CCD_i or SCS.

- Q is not computable from the union of the predicates in CCD_i. The tuples in this case must be remotely fetched from the server. As outlined in the concurrency control protocol above, the request for remote execution is accompanied by any tuples locally read or updated by the transaction since the last communication with the server. The server locks the tuples appropriately and also records the uncommitted updates before executing Q, locking the tuples accessed by it, and returning the result to C_i. The new tuples are placed in the cache at C_i, with CCD_i being *optionally* augmented. If these tuples are cached past the transaction boundary, SCS *must* be updated before the locks on the tuples are released at the server (upon transaction commit or abort). It is not necessary to update SCS at the time the tuples are fetched; locking tuples at the server provides the usual level of isolation from concurrent transactions.

- Q is partially contained in the union of the predicates in CCD_i. As in the preceding case, the query can be executed remotely at the server. One possible optimization is to *trim* the query before submission to eliminate tuples or attributes available locally at the client. Trade-offs involved in this type of optimization are discussed briefly in Section 4.3.

5.2.2 Data Insertion

Insertion at Client C_i

When a tuple is inserted by a transaction running at client C_i, it is placed locally in the cache with an *uncommitted* tag. If the transaction later commits successfully at the server, the new tuple is inserted into the central database, incorporated into SCS, and the *uncommitted* tag is removed at C_i. We have assumed here that the new data is likely to be pertinent for C_i, and hence is cached by it past the boundary of the current transaction. The insertion of this tuple may also affect caches of clients other than C_i (this case is discussed below).

Note that insertion of a tuple may alter the cache description, as perceived by the transaction, for cached join results. For example, if a new `BUG` tuple is inserted with product number 500, and the client is caching a join result (`BUG` $\bowtie$ `PRODUCT`), then the join result after insertion of the new tuple is in general not computable locally, unless all `PRODUCT` tuples are known to be available. The cache containment reasoning must therefore take into account the effect of uncommitted changes made by a transaction on cached join queries; selection and projection query results still remain valid after the insertion or deletion of a tuple.

Insertion at Client C_j

Suppose that a transaction running at client C_j, $i \neq j$, inserts a tuple t. The tuple is inserted into the database when the current transaction at C_j commits. The server checks

SCS to determine which clients other than C_j are affected by the insertion. Client C_i will be notified of the change, along with the new tuple, if t is contained in some P, where $P \in LCD_i$. If C_i is notified, it inserts the new tuple into its cache whenever it satisfies any predicate in CCD_i, and discards it otherwise. No changes are necessary to either CCD_i or SCS. Alternative courses of action are also possible; e.g., the client may choose to extend a nearby existing predicate to include the new tuple, or flush from CCD_i all predicates invalidated by the insertion (e.g., join predicates). The tuples corresponding to the invalidated predicates may or may not be removed immediately from the cache. They may still be used individually for answering point queries locally.

According to the concurrency protocol adopted in this thesis, a transaction running at client C_i must be aborted if the inserted tuple falls within the purview of any cached predicate that has been used to evaluate locally one or more queries within this transaction.

5.2.3 Data Deletion

Deletion at Client C_i

Assume that one or more tuples are deleted at client C_i using query predicate Q. If all tuples satisfying Q are not locally available in the cache, the procedure outlined above for a selection query is followed for Q. All or only the missing tuples in Q are fetched from the server after locking them, and locally cached. CCD_i may be optionally augmented. Tuples satisfying Q are then deleted locally in the cache, but marked as *uncommitted*. The server is informed of the deleted tuples upon transaction commit (or earlier, if a remote query is submitted before the commit). The *uncommitted* tag is removed from the cache if commit is successful. If the predicate Q is cached past the transaction boundary (even though its tuples may have been deleted), SCS must be updated before the locks on the tuples are released.

Retaining predicates in CCD_i whose tuples have been deleted can potentially reduce query response times at the client by allowing local determination of the fact that a subsequent query result is empty and avoiding a trip to the remote server. For example, let all `BUG` tuples satisfying `(Product#` $\geq$ `300)` be cached at C_i. If `BUG` tuples for product number 500 are now deleted by a transaction, then the assertion that the cache holds all `BUG` tuples with the property `(Product#` $\geq$ `300)` is still valid after the deletion. A subsequent query for bugs in product number 500 can be evaluated locally, producing 0 tuples in its result set.

Deletion at Client C_j

Let one or more tuples be deleted using query predicate Q at client C_j, $i \neq j$. Tuples satisfying Q are deleted from the database when the current transaction at C_j commits successfully. The server must again notify clients other than C_j that are affected by the deletion, by comparing Q with SCS. Client C_i will be notified of the deletion if $\exists P \in LCD_i$ such that $(Q \cap P \neq \phi)$.

The notification message may consist of primary keys of deleted tuples, or for large-sized deletions, simply the delete command itself. If client C_i is notified, it must execute the delete command on its cache. No changes are required to CCD_i or LCD_i. A transaction running at C_i must be aborted if any tuples it read locally get deleted due to the notification.

5.2.4 Data Update

Update at Client C_i

If one or more tuples are updated at client C_i using query predicate Q, then the actions taken by client C_i and the effects on CCD_i and SCS are similar to the deletion case above, except that the update may move some tuples from one cached predicate to another predicate (which may or may not already be cached at C_i), depending upon the attributes updated. If such tuples are cached beyond the completion of the transaction, either the individual tuples or a single *modified predicate* [Blak89] describing tuples after the update *must* be inserted in SCS, and *may* optionally be inserted into CCD_i.

Update at Client C_j

If one or more tuples are updated using query predicate Q at client C_j, $i \neq j$, then the updated tuples, or simply the corresponding update command for large-sized updates, are sent to the server when or before the transaction issues a commit request at the client. The server posts the changes to the central database if the commit is successful. The notification procedure is more complex than that for the delete case above, since the values of the changed tuples both *before* and *after* the update must be considered to determine which client caches are affected. The set of clients to be notified by the server depends not only on the query predicate, but also on the updated attributes (exact algorithms appear in [Blak89]).

Note that all updated tuples that no longer satisfy any cached predicate should be discarded by C_i, or else LCD_i at the server must be augmented to include them. Precise screening of each updated tuple with respect to the cache can be done locally at a client site instead of at the central server, thereby distributing some work in maintaining cached results to individual clients. The trade-off is between local computation as opposed to global communication.

5.2.5 Transaction Abort

If a transaction submits an abort request, then the local transaction is aborted, and all uncommitted changes made by the transaction are undone. If the transaction had performed any remote operations, then a request is sent to the server to also abort the corresponding remote transaction. If the transaction had not made any updates, then tuples fetched by it from the server can be placed in the cache, after appropriately extending the client subscription at the server.

5.2.6 Transaction Commit

When a transaction commits at client C_i, the server is informed of all local updates that have not yet been communicated to it. The propagation of updates can either be in the form of updated tuples and corresponding update commands, or for large-sized updates, in the form of update commands only so as to minimize network traffic and message processing costs. C_i is notified of the result of the commit operation. If the commit was successful (according to the message numbering scheme described below), the *uncommitted* tags on tentatively updated data are removed from the cache by C_i; otherwise, the changes are undone. In either case, all locks held by the transaction are released at the server, *after* the server has updated SCS to record all new predicates and tuples cached by the client beyond the transaction.

Commit Verification using Notification Message Numbers

Our commit verification algorithm uses a message numbering scheme, which is similar to the notify locks algorithm presented in [Wilk90] in the context of identity-based caching. The server sends notification messages to clients informing them of potential conflicts. However, due to network delays and asynchronous notification, it is possible that some notification messages sent by the server may not have reached the client by the time a transaction submits a commit request. To ensure serializability of transactions, a special commit verification step is required by the clients and the server, as described below.

Commit Processing Steps at a Client

The flowchart in Figure 5.1 shows the actions taken by a client during processing of a commit request. We make use of three variables: $Client\_Max$, $Server\_Max$, and $LastMsg\#$. These three variables denote the following quantities:

- $Client\_Max$: The sequence number of the last notification message processed at the client.
- $Server\_Max$: The number of the last notification message sent by the server to the client, which is possibly greater than $Client\_Max$ due to network delays.
- $LastMsg\#$: A variable used to denote the sequence number of the last notification that needs to be processed in order to verify a commit request.

The steps in the flowchart are described below in terms of these three variables.

- *Step C1.* The currently running transaction T submits a commit request. This is the initial event that triggers this algorithm. The client responds by executing Step $C2$, with the variable $LastMsg\#$ set to the value $Client\_Max$.
- *Step C2.* Check if there are any notification messages from the server with a sequence number equal to or greater than the specified $LastMsg\#$. If there are no such pending notifications, proceed to Step $C3$; otherwise, go to Step $C4$.
- *Step C3.* Send any local updates made by transaction T to the server, along with the sequence number $Client\_Max$ of the last notification message processed by the client. Go to Step $C6$.

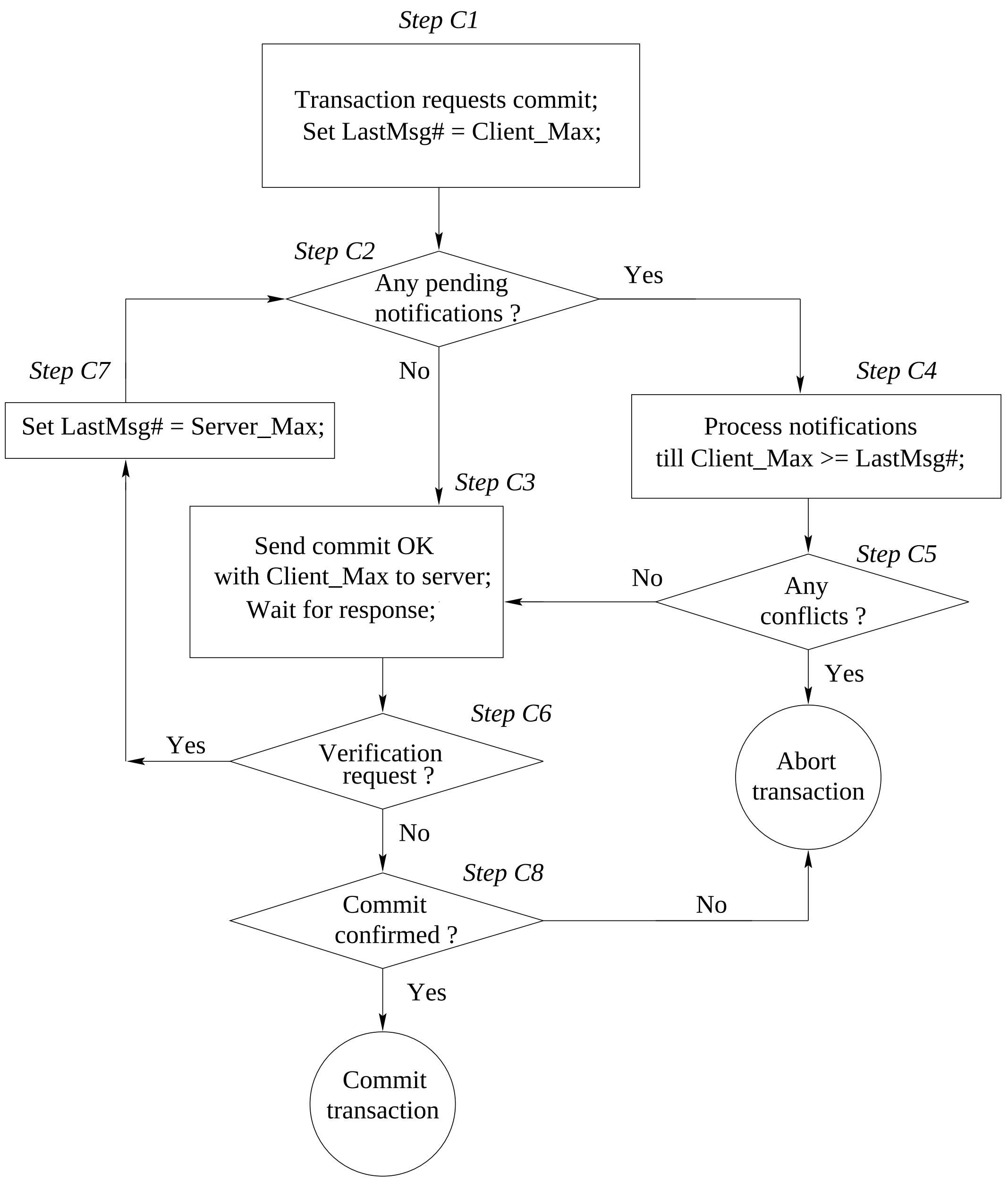


Figure 5.1: *Commit Processing Steps at Client*

- *Step C4.* Process notification messages from the server till the sequence number of the last notification received equals or exceeds the specified $LastMsg\#$. Set the value of $LastMsg\#$ to the sequence number of the last notification ($Client\_Max$) processed at the client, and go to Step $C5$.

- *Step C5.* Updates made by notifications may or may not have conflicted with the updates made by transaction T. If there are any conflicts, then abort transaction T, notify the server of the abort, and terminate commit processing; otherwise, go to Step $C6$.

- *Step C6.* Wait for a response from the server. If the server sends a verification request, go to step C7; otherwise, go to Step $C7$.

- *Step C7.* A verification request was received from the server. It contains the sequence number $Server\_Max$ of the last notification message sent to this client. Set the value of $LastMsg\#$ to $Server\_Max$, and go back to Step $C2$ to process more notifications.

- *Step C8.* If the server accepted the commit request, then commit transaction T and make its updates permanent in the client cache; otherwise, abort the transaction T. Terminate the algorithm.

Commit Processing at the Server

The flowchart in Figure 5.2 shows the server actions taken in confirming a commit request from a transaction running at the client. We use the variables $Client\_Max$ and $Server\_Max$ to denote the same quantities as defined above for commit processing at the client. The steps in the server algorithm are described below.

- *Step S1.* The server receives a commit request from the transaction T currently running at client C. This is the initial event that triggers this algorithm. The commit request contains the sequence number $Client\_Max$ of the notification message last processed by the client. The server starts commit processing by executing Step $S2$.

- *Step S2.* Compare the sequence number $Client\_Max$ sent by the client with the value $Server\_Max$ of the last notification message sent by the server to this client. If $Server\_Max$ is greater than $Client\_Max$, then go to Step $S7$; otherwise, proceed to Step $S3$.

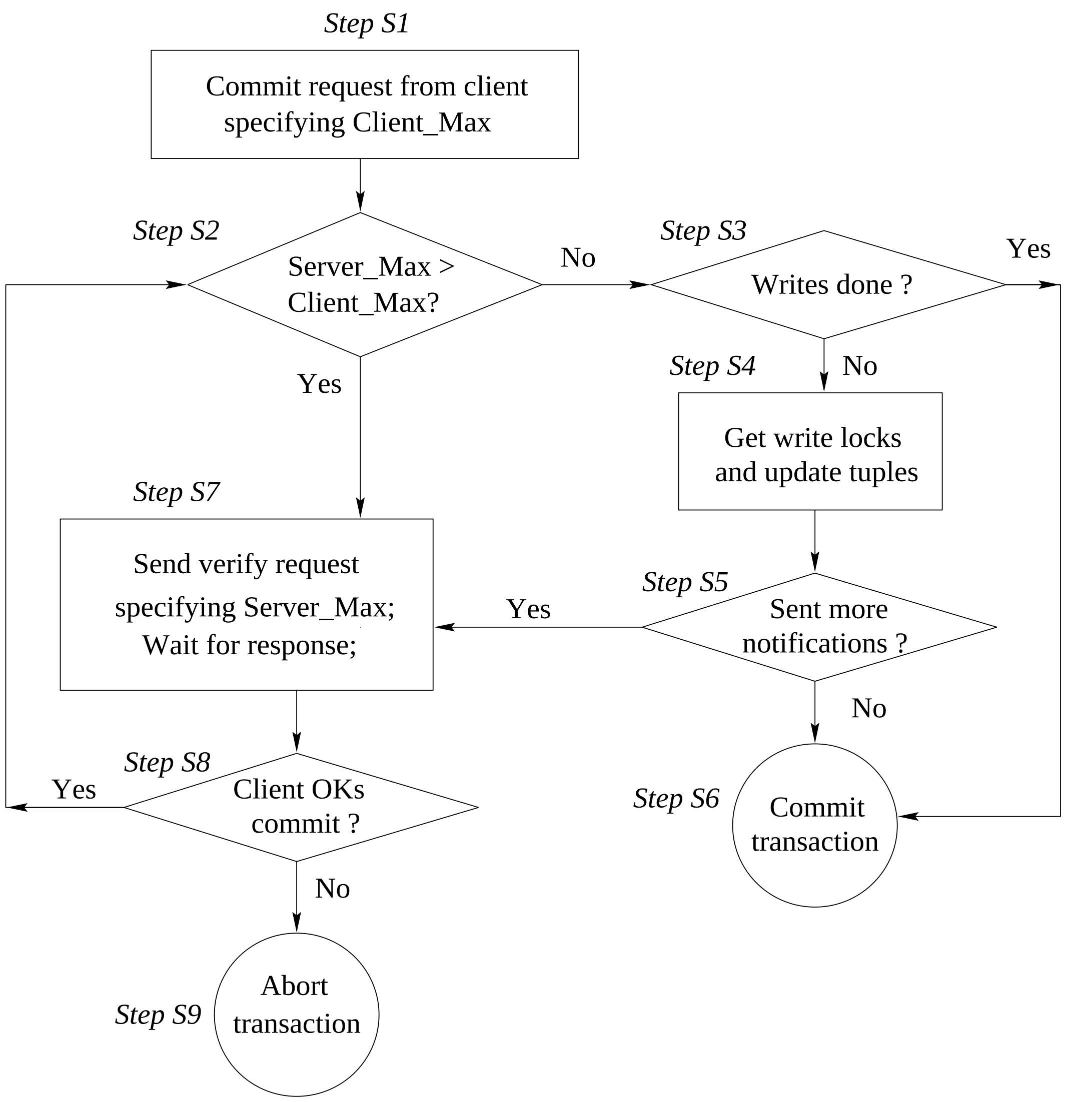


Figure 5.2: *Commit Processing Steps at the Server*

- *Step S3.* Check if the transaction T performed any updates that are yet to be reflected at the server. If yes, proceed to Step $S4$; else go to Step $S6$ to accept the commit request.

- *Step S4.* Obtain write locks for the modified tuples and apply the updates made by transaction T to the locked tuples. After the last lock has been obtained, and while they are being held, another check of message sequence numbers is required. This check is necessary since other transactions may have committed updates in the meantime that invalidated the read/write set of the committing transaction. Proceed to Step $S5$.

- *Step S5.* Clear any pending updates in the notifier queue, and check if any more notifications were sent to client C. If yes, go to Step $S7$; else proceed to Step $S6$ to accept the commit request.

- *Step S6.* Post the updates made by transaction T to the database, and trigger the notifier to check the effect of these committed updates on other clients. Inform client C that the commit is accepted, and terminate commit processing of transaction T.

- *Step S7.* Send a commit verification request to client C, with the sequence number $Server\_Max$ of the last notification message sent to C. Wait for the client to respond, and proceed to Step $S8$ upon receiving a reply.

- *Step S8.* If the client confirms the commit request (specifying its last sequence number $Client\_Max$), then go to Step $S2$. Otherwise, proceed to Step $S9$.

- *Step S9.* Abort transaction T, and inform client C that the commit did not succeed. Undo any (uncommitted) updates made by T, and release any locks held by it. Terminate this algorithm.

The proof of correctness of the above transaction execution scheme can be formalized in terms of preserving the serializability of transactions. This proof is presented in the next section.

5.3 Serializability of Transactions

In this section, we demonstrate that A*Cache transactions executed in conformance with the steps given in this chapter do not violate serializability constraints. In order to be serializable, a transaction must have the following properties [Gray93b]:

- (1) No dirty reads of data changed by other uncommitted transactions,
- (2) No unrepeatable reads of individual tuples as well as of predicate-based queries,
- (3) No lost updates due to multiple uncoordinated writes.

We will now show that all of these three conditions are satisfied by transactions that execute using A*Cache, given that the server supports serializable transactions, and that the assumptions (A1) through (A4) stated in Section 5.1 hold.

5.3.1 No Dirty Reads

This condition is met trivially, since A*Cache always stores committed updates in the cache. Only the updates made by a transaction itself are made visible to it as it executes at the client or at the server, and these changes are undone if the transaction subsequently aborts. Uncommitted changes of other transactions are not present locally, and by assumption (A2) above, are also not perceived at the remote server.

5.3.2 No Unrepeatable Reads

We will show that all reads in the presence of A*Cache are repeatable, both for individual tuples and for predicate-based queries.

Let us first consider an individual tuple X, and assume that there is an unrepeatable read in a transaction $T1$. There are four possibilities: (1) X was read both times at the server, (2) X was first read remotely and then read locally, (3) X was first read locally at the client and then remotely at the server, and (4) both reads of X happened locally at the client.

By our assumption (A2), the server supports long duration read and write locks for all remote reads, and therefore, case (1) is not possible.

Now consider case (2). The server sets a long duration read lock on X upon the remote read operation, say at time t_1, and returns a value X' for X. Say the transaction $T1$ reads a different value X'' of the same tuple later at the client, with the cache read setting a local read lock at the client. For the two reads to be different, updates made by one or more transactions must have changed the older value X'' to X', before the read lock was acquired on it by $T1$ at the server at time t_1. These updates must not have been reflected on the cache at the time of the local read. If the notification message for these updates arrives before $T1$ submits its commit request, then by our assumption (A3) in Section 5.1, $T1$ will be aborted at the client. Otherwise, $T1$ will continue to execute operations either at the client and/or at the server, and will finally submit a request for commit to the server, say at time $t_2 \geq t_1$.

During processing of the commit request, the commit verification algorithm given above (Section 5.2.6) will be invoked to check that all notifications for updates made by other transactions that can affect potentially transaction $T1$ have been sent to the client and examined for conflicts. Since the tuple X was modified at some time before it was locked at t_1, which is less than the commit request time t_2, the update notification for the tuple X must be processed by the client before the commit is accepted. The processing of this notification message will detect a conflict with the local read lock set at the client, and the commit request of transaction $T1$ will therefore be refused.

Cases (3) and (4) can be reasoned using similar logic as above, proving that unrepeatable reads of individual tuples is not possible in A*Cache.

The repeatability of predicate-based queries in A*Cache can also be established similarly. The notification and cache maintenance algorithms at the server and at the client reason at the level of query predicates, and therefore detect any update conflicts on query predicates that were evaluated locally. Our commit verification scheme ensures that such detection occurs before a transaction is allowed to commit. Hence, repeatable reads are guaranteed not only at the level of individual data items, but also at the level of query predicates.

5.3.3 No Lost Updates

Lost updates may result from one transaction over-writing an update made by another transaction, as in the following sequence of operations:

```
Transaction T2:  Read X     /* X has a value 1 */
Transaction T1:  Write X    /* sets X to 2 */
Transaction T2   Write X    /* T2 overwrites the value written by T1 */
```

The above sequence is a bad Read-Write-Write sequence, causing T1's update of X to be lost, since the value written by T2 is based on an earlier read of X by T2. A bad Write-Write-Write can also cause an update to be lost. As discussed in [Gray93b], a lost update can be prevented in a centralized system by setting long duration write locks.

There are two cases to consider for updates in the client-server environment of A*Cache: cache hits and cache misses. In the case of a cache miss in A*Cache, long duration write locks will be set by the server on the set of tuples to be updated. Our commit verification algorithm will detect any conflicting updates that occur after the transaction started and before these locks were obtained, thus precluding the possibility of a lost update.

For cache hits, long duration write locks are only set locally. However, local updates are transmitted to the server "piggy-backed" with the next remote request, and thereafter the same reasoning as in the cache miss case applies. Compared to a cache miss, the window of time between performing a local update and obtaining a write lock at the server is larger in the case of cache hits, raising the chances of a conflict. Other than this increased optimism for cache hits, there is no difference in the conflict detection algorithm for the two cases. It follows therefore that any lost updates will be detected in A*Cache for both cache hits and cache misses, and the transaction will be aborted if necessary.

5.3.4 Effect of Reduced Consistency Levels at the Server

A 3^o transaction has 'complete' isolation from the activities of other concurrently executing transactions, whereas lesser consistency levels may see some effects of concurrently executing transactions. Support for repeatable reads at the level of individual data items provides level of isolation that is slightly lower than 3^o, and is sometimes referred to as 2.99^o [Gray93b]. To be completely serializable, the repeatable read property must also be satisfied by the *sets* of data items returned by the predicate-based reads made by a transaction, such as using `SELECT-FROM-WHERE` SQL queries. This property is also known as *phantom protection* [Gray93b], where the appearance or disappearance of *phantom* records from any predicate-based read is prevented over the duration of the transaction.

Full serializability is often too expensive for practical environments. Most commercial systems do not support predicate locks directly, and transactions desiring full serializability must lock entire tables. This overhead is often too restrictive, and therefore, lower levels of isolation are popular in practice [Bere95, Orac95].

We now consider the effect of lower levels of isolation at the server on the operation of A*Cache. Note that transactions that execute with less than 2^o isolation may read uncommitted data, and therefore data written by them may be invalid and inconsistent. It is therefore not common for 0^o and 1^o transactions to update any data [Gray93b]. In A*Cache, the client cache contains committed data only, and hence, it preserves the same consistency of data as at the server, and supports 2^o isolation by default. Only the uncommitted effects of a locally executing transaction are visible to itself, which is the usual isolation semantics for SQL.

Notice however that alternating local and remote reads could cause a client transaction to switch between past and present states of the database. Although the 2^o read committed property is preserved, the chronology of database states is not guaranteed to be maintained. This behavior could cause a problem in cases where a 'read forward' property is required for the data. Therefore, in the presence of client caching, 2^o isolation may be extended to include the chronology property, as defined in [Hull96].

Besides reading committed data in a chronological order, a transaction might additionally want to specify currency conditions for local reads. For example, a transaction might require that reads of cached data be no more than an hour old. The operation of A*Cache can be easily extended to handle such currency specifications. If the server has generated and sent notifications of all updates committed before an hour, and these notifications have all been processed by the client, then the cached data meets the specified currency constraints. The former condition must be verified by the server, and the latter condition can be checked using the sequential numbering scheme for notification messages. Note however that updates made by such transactions may violate integrity constraints on the database, since they might operate on stale data.

As a special case, consider the isolation level 2^o with no lag. This level of isolation essentially specifies that each local read return the same result as a corresponding remote read. This behavior may be implemented in a variety of ways, including optimistically allowing potentially stale local reads to occur, but aborting the transaction at a later point if the specified lag is found to have been violated. Repeatability of reads at the level of predicates is not required however, since the consistency level is only 2^o.

5.4 Summary of Execution Scheme

In this chapter, we have described in detail the execution mechanism of transactions in an A*Cache system. First, we listed the assumptions that apply to our discussion. Actions taken by the server and the clients for data reads, updates (inserts, deletes, and modifications), and upon termination of transactions (commit or abort), were then specified. An extra verification step is required at commit in order to detect any conflicts with local operations on cached data. Our commit verification algorithm uses numbering of notification messages, and is based on the scheme for notify locks in [Wilk90]. We demonstrated that our transaction execution scheme is correct in that it preserves serializability. Finally, we considered some consistency issues for servers that support lower non-serializable levels of data isolation. Future avenues for work include development of appropriate consistency models for lower levels of data isolation in the presence of client-side caching, and formalizing the operation of A*Cache in such scenarios.

Chapter 6

Simulation Model and Experimental Setup

In this chapter, we develop transaction execution and workload models for our simulation study. The steps involved in executing transactions in A*Cache have been described in detail in the preceding chapters of this dissertation. Below, we specify in pseudo-code the algorithms used in our simulation environment. These algorithms reflect the architecture and operation of A*Cache as described earlier, and additionally, they also specify the transaction execution and caching policies chosen for our simulation. These policies define the algorithms adopted for various system tasks such as space management and cache maintenance. The transaction execution model is supplemented by the workload model, which represents the patterns of data access and update by different clients. Thus, our model of the workload defines the data contention parameters that are of interest in our simulation.

In order to have a relative comparison of our performance results for A*Cache, we also simulate the operation of three other types of client-server databases. Two of these schemes, namely, *No-Cache* and *Tuple-Cache*, are alternatives to A*Cache, and the third, called *A*Cache_Opt*, is an optimized extension of the basic A*Cache model. Procedures for caching and transaction execution in these three schemes are described, and the differences in their query and update processing strategies are noted. These differences will be significant in subsequently interpreting our simulation results.

Next, queueing models of the various logical processes in the operation of the no-caching and caching systems are presented. The physical configurations of the simulated systems are specified in terms of the resources of the clients, the server, and the network. The default settings of the simulation parameters corresponding to these physical resources are also provided. We then describe the workload model adopted for the experiments in our performance study. The basic framework of this model is a subset of the standard Wisconsin benchmark [Gray93a], extended with the data locality and contention models of [Care94] to represent the different data access and update patterns of multiple clients in a database system.

Some implementation notes on our C++/CSIM coding of the simulator appear towards the end of this chapter. Finally, we discuss our experimental methodology and the validation process that was followed to test the correctness of our simulator implementation.

6.1 Assumptions for Simulation

Following are the assumptions that are adopted for transactions in our simulation environment:

- We assume that the client stores data in main memory. 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.

- Update notifications always refresh the cache. As discussed in Section 3.3.4, it is also possible to have invalidation-based cache maintenance schemes. This type of maintenance is not investigated in our simulation study; examining the performance of such schemes is the subject of future work.

- 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 5.2.6.

6.2 Execution Model in A*Cache Simulation

The execution model for A*Cache has been described in detail in the preceding chapters of this dissertation. In this section, we outline using pseudo-code the algorithms used for executing transactions in our simulation environment. For the purpose of the simulation, specific policies must be chosen for various system tasks such as space management and cache maintenance, and these policies define the particular actions taken on an event occurring in the system. For example, an LRU policy is used for cache replacement, which removes data that is the least recently used when the cache is full. Note that although we have generally adopted 'standard' policies such as FIFO queues and LRU replacement, the choice of these policies is specific to our simulation; alternative policies may be chosen in other simulated systems or in real-life implementations, possibly resulting in different performance profiles.

The transaction processing algorithms in an A*Cache system are given below in two parts: first, the actions taken by a client are defined, and then the server-side logic is specified.

6.2.1 Client-Side Logic

At the client, a cache hit causes local evaluation using data in the cache, whereas a cache miss fetches from the server all tuples that are to be read or updated, which are then cached and processed locally. Partial hits are treated as cache misses in this basic A*Cache model. Local updates are flushed to the server upon submission of a remote read request, since an 'in-flight' transaction must be able to view its own updates as it executes remote operations at the server. As noted in Section 3.3.3, this behavior represents the isolation semantics for standard ANSI SQL, and this policy is also adopted in A*Cache.

```
/* *******************************************************
 * Client-side logic for transactions in A*Cache simulation *
 * ********************************************************/
For each request R submitted by a transaction, do
{  If R is a query or an update, then
   {  /** cache hit case **/
      If R is completely contained in the cache, then
          Execute R locally;
      Else
      {  /** Cache miss case.
          ** Note: Basic A*Cache doesn't consider partial cache hits
```

```
        **/
        Send R and any local updates to the server;
        Fetch all tuples accessed by the query predicate of R;
        /* ************************************************
         * Simulation policies for cache space management:  *
         *      store tuples and predicates upon fetch;     *
         *      purge tuples using LRU predicates           *
         * ************************************************/
        Store new tuples in cache, reclaming space if necessary;
        If space was reclaimed, then
        {  Update cache description;
           Send message to server specifying purged predicates;
        }
        If R is an update, then
        { /* **********************************************
           * Simulation policy for updates:               *
           *   update tuples locally if cached;           *
           *   flush local updates with next remote request *
           * **********************************************/
          Update tuples locally;
        }
    }
    Else if R is a commit, then
    {   Send any local updates to server with commit request;
        Verify commit if necessary;
        If commit accepted by server, then
        {  Make updates permanent in cache;
           Commit transaction;
        }
        Else
        {  /* **************************************************
           * Simulation policy for transaction serializability: *
           *   abort on conflict with updates of other clients  *
```

```
            * ***************************************************/
            Undo local updates;
            Abort transaction;
        }
    }
    Else if R is an abort, then
    {   Send abort message to server;
        Undo local updates;
        Abort transaction;
   }
}
```

Observe that in the above algorithm, we have chosen a simple cache loading policy, in which all new results fetched from the server are inserted into the cache, after reclaiming space as necessary. In a practical scenario, this policy could be refined to analyze the long-term benefits of caching the query results (Section 3.3.6), before deciding to store them locally. The client must also implement a cache replacement scheme in order to accommodate newly fetched data. In our simulation, we use an LRU mechanism at the level of predicates along with reference counts for individual tuples, as described in Section 3.3.6. The reference count maintained for each tuple in the cache denotes the number of predicates that refer to a certain tuple. Before the space for a tuple is reclaimed through a predicate, it is first checked that its reference count is not greater than one; otherwise, the tuple is not removed. It is possible to employ more sophisticated algorithms for space management, as in [Dar96], but they are not considered in our study.

In addition to the process that executes transactions at the client, there is a parallel client process which handles notification messages received from the server. This update handling process is interleaved with the process executing transactions, and follows the logic below:

```
/* *****************************************************************
 * Client-side logic for update notifications in A*Cache simulation  *
 * *****************************************************************/
For each notification message N received from the server, do
{  If N conflicts with any reads or writes of the current transaction, then
```

```
  {  /* ***************************************************
      * Simulation policy for transaction serializability:   *
      *   abort on conflict with updates of other clients    *
      * ***************************************************/
      Send abort message to server;
      Undo local updates;
      Abort current transaction;
   }
   If N affects the current cache contents, then
   {  /* ******************************************
       *  Simulation policy for cache maintenance:    *
       *    refresh updated data from notifications   *
       * ******************************************/
      Update cache description if necessary;
      Update modified tuples in the cache;
   }
 }
```

The above notification handling algorithm used in our simulation enforces full serializability of transactions, so that client transactions are aborted when conflicts are detected with data updates made by other clients. Variations of this concurrency control policy are possible in the general A*Cache scheme (see discussion in Section 3.3.3), but are not considered in our simulation. The cache maintenance policy chosen in our simulation study refreshes cached data that has been updated. Thus, notification messages contain the new image of the data after update. In the general A*Cache framework, it is also possible to have invalidation-based maintenance schemes, as discussed in Section 3.3.4. Maintenance based on invalidation is not investigated in our simulation study; examining the performance of such schemes is the subject of future work.

6.2.2 Server-Side Logic

The server processes the requests submitted by client transactions, and generates update notifications for affected clients as updates are committed to the database. Generation of update notifications is done using the predicate descriptions of the cache contents of each client. The server is also responsible for verifying at transaction commit that serializability constraints were not violated.

```
/* ************************************************************
 * Server-side logic for client requests in A*Cache simulation *
 * *************************************************************/
For each request R submitted by client C, do
{  If R is a query or an update, then
   {  /* ************************************************
       *  Server policy for data retrieval:            *
       *    Read-lock tuples                           *
       * ***********************************************/
      Read-lock and retrieve required tuples;
      /* ********************************************************
       *  Server policy for client subscriptions:              *
       *    register new predicate before sending data to client *
       **********************************************************/
      Register access predicate for R in subscription of client C;
      Release read locks;
      Send the result tuples to client C;
   }
   Else if R specifies purged predicates, then
   {  Update subscription of client C;
   }
   Else if R is an abort, then
   {  Undo any updates made by the transaction;
      Abort transaction;
   }
   Else if R is a commit then
   {  Write-lock updated tuples;
      Verify commit with client C using notification message numbers;
      /* ****************************************************
       *  Server policy for transaction serializability:     *
       *    Abort if commit is not verified by client        *
```

```
         * ****************************************************/
        If commit verification fails, then
        {  Deny commit request;
           Undo any updates made by the transaction;
           Abort transaction;
        }
        If commit succeeds, then
        {  Post updates to database;
           Release all locks;
           Send 'commit OK' message to client C;
           For each update, do
           {  For each client C1 != C with registered subscription S1, do
              {  If any predicate in S1 is affected by the update, then
                 {  Mark C1 as 'affected';
                    /* ****************************************
                     * Server policy for notifications:       *
                     *   send updates in notification message  *
                     * ****************************************/
                    Add updated data to notification message for C1;
                 }
              }
           }
           Send update notifications to all clients marked 'affected';
        }
    }
 }
```

As outlined in the above algorithm, 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 for the command is recorded by the server. Observe that the server always registers a new predicate in the client subscription *before* sending the result of a remote query. This behavior is in accordance with our choice of the cache loading policy for our simulation, in which all results fetched from the server are cached locally. Other cache loading policies are possible in A*Cache (see discussion in Section 3.3.6) but are not considered in our performance study. Also note that in our simulation environment, the updated data is sent in the notification message. This scheme supports the cache maintenance policy chosen for our simulation, in which update notifications always refresh the cached data.

Note that the execution model described above represents basic A*Cache operation, with no optimizations. For example, partial cache hits are not considered. This basic model will be extended in the next section (Section 6.3) to incorporate some optimizations aimed at improving its performance.

6.3 Alternate Caching Schemes for Performance Comparison

In order to analyze and interpret the experimental results, one or more schemes must be chosen to serve as the bases for comparison. For our simulation study, we consider the following three types of client-server systems as alternatives to A*Cache:

1. A database with no client-side caching, hereafter called *No-Cache*;
2. A system that we call *Tuple-Cache*, which caches only tuples at clients without any predicate descriptions, and
3. A scheme named *A*Cache_Opt*, which is an optimized extension of the basic A*Cache model.

Query and update processing algorithms for these three schemes are described below, and the differences in their handling of cache hits and misses are noted in particular.

6.3.1 A Client-Server Database with No Caching

A client-server system with no caching at client sites serves as a simple baseline for comparison of our experimental results. In this system, called the *No-Cache* scheme, queries and updates are always executed at the server, and there is no update notification required from the server. Thus, no cost is incurred in cache containment reasoning or maintenance, or in transaction aborts due to update conflicts. As in most commercial databases, central read/write locks acquired at the server are used for concurrency control.

6.3.2 A Client-Server Database with Tuple-Based Caching

In this caching scheme, called the *Tuple-Cache* scheme, it is assumed that the client cache stores only tuples and does not maintain the 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 1.1.2, 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 the client in 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 (possibly using indexes) which tuples belong in the query result, and the set of tuples that are missing from the cache. In other words, the Tuple-Cache scheme recognizes cache hits only at the level of individual tuples and not entire queries, and a round-trip to the server is necessary to process every associative query or update.

In the Tuple-Cache scheme, we assume that the contents of each client cache is represented by a list of unique identifiers for the cached tuples. Such a list is used at the client site for query execution and space management purposes, and is also maintained by the server. The list at the server site serves two purposes: (1) to determine which tuples in a query result are missing from the client cache, and (2) to filter out update notifications that are irrelevant for a client.

The algorithms given below describe in pseudo-code the actions taken by clients and the server to process transactions in the Tuple-Cache scheme.

```
/* ***********************************************************
 * Client-side logic for transactions in Tuple-Cache Simulation *
 * ***********************************************************/
For each request R submitted by a transaction, do
{  If R is a query or an update then
   {  Send R and any local updates to the server;
      Fetch from the server any missing tuples for R;
      /* ************************************************
       * Simulation policy for Tuple-Cache space management: *
       *      store tuples upon fetch;                      *
       *      purge LRU tuples                              *
       * ************************************************/
      Store new tuples in cache, reclaiming space if necessary;
      If R is an update, then
         Update tuples locally;
   }
   Else if R is a commit then
   {  Send any local updates to server with commit request;
      Verify commit if necessary;
      If commit accepted by server, then
      {  Make updates permanent in cache;
         Commit transaction;
      }
      Else
      {  /* ************************************************
          * Simulation policy for transaction serializability: *
          *   abort on conflict with updates of other clients  *
          * ************************************************/
          Undo local updates;
          Abort transaction;
      }
   }
   Else if R is an abort then
```

```
    {  Send abort message to server;
       Undo local updates;
       Abort transaction;
    }
}
```

In the above algorithm, note that updates are made locally at a client after fetching the tuples missing from its cache. This update scheme is chosen for Tuple-Cache, since it is comparable to the update-handling mechanism in the basic A*Cache model. In the latter scheme, a cache miss causes a remote fetch of all tuples to be updated, which are then updated locally; cache hits in A*Cache also produce local updates. In contrast, for Tuple-Cache, cache miss is the automatic default for each query or update, the tuples being processed locally after the initial step of fetching any missing tuples from the server.

The client must also implement a space management policy. In our simulation of Tuple-Cache, all newly fetched tuples from the server are cached locally, and space is reclaimed using an LRU scheme at the level of tuples.

In addition to the query and update processing tasks, 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 an A*Cache client, since old copies of the modified tuples are directly replaced with their new images. Unlike A*Cache, there are no cached predicates to be analyzed and maintained by the client, since only tuples are stored in this type of cache. The notification processing algorithm in Tuple-Cache is therefore a simpler version of the A*Cache algorithm given in Section 6.2.1, the simplification being that cached predicates are not involved in the process.

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. This requirement is true for all the caching systems we consider here, including Tuple-Cache, and can be enforced in a variety of ways. In our simulation of Tuple-Cache, we have adopted the following semi-optimistic concurrency control scheme with a commit verification step to detect conflicts, which is similar to the policy chosen for A*Cache:

```
/* ****************************************************************
 * Server-side logic for client requests in Tuple-Cache simulation *
 * ****************************************************************/
For each request R submitted by a client C, do
{  If R is a query or an update then
      {
         Determine the set of tuples S required to execute R;
         Check list of tuples T cached by C;
         /* ******************************************
          *  Server policy for data retrieval:       *
          *    Read-lock tuples                      *
          * ******************************************/
         Read-lock and retrieve tuples in the set (S-T);
         Record the access predicate for R;
         Release read locks;
         Send tuples in the set (S-T) to client C;
      }
      Else if R is an abort, then
      {  Undo any updates made by the transaction;
         Abort transaction;
      }
      Else if R is a commit, then
      {  Write-lock updated tuples;
         Verify commit with client C using message numbers;
         /* *****************************************************
          *  Server policy for transaction serializability:     *
          *    Abort if commit is not verified by client        *
          * *****************************************************/
         If commit verification fails, then
         {  Deny commit request;
            Undo any updates made by the transaction;
            Abort transaction;
         }
```

```
          If commit succeeds, then
          {  Post updates to database;
             Release locks;
             Send commit OK message to client C;
             For each updated tuple t, do
             {
               For each client C1 != C, do
               {  If tuple t is cached by C1, or
                        it affects the current transaction at C1, then
                  {  Mark C1 as 'affected';
                     /* ****************************************
                      * Server policy for notifications:        *
                      *   send updates in notification message   *
                      * ****************************************/
                      Add tuple t to notification message for C1;
                   }
                }
             }
             Send update notifications to all clients marked 'affected';
          }
      }
  }
```

As outlined in the above algorithm, 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 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.

For associative queries, the Tuple-Cache scheme can be effective only if the queries use indexed attributes, which is the case in our simulation workload (Section 6.6). For such queries, the server can use its index pages to determine the set of tuples in a query result, thereby avoiding full-fledged query execution on the database. In other words, tuples in the result set that are missing from the client cache can be detected using the indexed attribute, and only these tuples are retrieved from the database. In this sense, the Tuple-Cache scheme is related to the index-based associative caching systems discussed in Section 2.2.1, 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.

6.3.3 An Optimized Extension of A*Cache

This scheme, which we call *A*Cache_Opt*, introduces two optimizations in the basic A*Cache model described earlier:

- 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 on a cache miss for update operations. 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, even if some are already present in the cache, and performs the requested update locally, subsequently flushing these updates to the server.

  This extended update policy for a cache miss 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. The A*Cache_Opt scheme is therefore less optimistic about data writes than the basic A*Cache and Tuple-Cache schemes, and can result in fewer transaction aborts.

The A*Cache_Opt scheme thus utilizes a cache miss as an opportunity to optimize network traffic and also to reduce data update conflicts among different clients.

In our simulation of the A*Cache_Opt scheme, for each remote query or update, the server sends all tuples that were either updated or were missing from the cache back to the client, which caches them locally. As in A*Cache, the access predicate for the operation is registered in the client subscription at the server before transmission of the results to the client, and the cache description at the client is updated prior to the tuples being stored locally. These operations are consistent with the cache loading policy chosen for our simulation, in which new tuples are always stored after they are fetched from the server.

For updates executed remotely by the client, we assume that other predicates previously cached by the client are not modified. This assumption is correct for the workloads chosen for our simulation (described below in Section 6.6), since the updated attributes are distinct from those attributes that are used to access the data for a query or an update. Thus, for a remote update in our simulation, storing a new access predicate and the updated tuples is sufficient for correct operation — a previously-cached tuple might possibly be modified by such an update, but other predicates cached earlier at the client are not affected.

Notification generation and processing in the A*Cache_Opt scheme is the same as in basic A*Cache. Processing of update notifications is predicate-based, so that query results and tuples cached at the client can be correctly refreshed as updates are committed at the database.

6.4 Simulation Models

In this section, we present simulation models for the No-Cache and A*Cache systems using queueing networks. First, we state the execution policies chosen for our simulation environment. We then describe the No-Cache model with data caching at clients, and then extend this model to A*Cache. Our models are based on the database simulation model originally presented in [Agra87]. This model was later extended to the client-server scenario in studies such as [Fran93, Wilk90, Wang91]. To represent the architecture and operation of A*Cache, we extend the basic client-server model of [Wilk90] with caching sub-systems at the clients and the server. Our extensions take into account the client-side caching facilities, data retrieval and maintenance costs for the cache, as well as the costs of processing update notifications at the server and clients.

6.4.1 Logical Queueing Model of a No-Cache System

Figure 6.1 shows the logical queueing model for a No-Cache system. For brevity, only one client is shown; however, multiple client modules are used in our simulation experiments. The various functional modules represent the logical processes that cooperate to execute transactions, and the queue associated with a functional module holds pending requests for service.

The operation of the system shown in Figure 6.1 is explained below in terms of its logical modules. Module names have been abbreviated in the figure for clarity, and each abbreviation is noted in parentheses in the description of the corresponding module.

Each client is modeled as follows. It has an application module (APP) that processes data downloaded from the database. Queries and updates to the database are generated by the Workload Generator (WLG), which creates a new database transaction upon request by the client application. All queries and updates are sent to the server through the *Send Queue* of the Network module (NET), and the results received from the server are directed from the *Receive Queue* of the network to the application module.

The network model adopted in our simulation is a simple one. It consists of a single service (shown split into two queues, *Send* and *Receive*, in the figure) with First-Come-First-Served discipline that handles data and messages sent from the clients to the server, and vice-versa. A simple model is adequate for the purposes of our simulation study, since we focus on the relative performances of the various caching schemes, and not on the details of network operation. The network bandwidth and the CPU costs of message processing are parameters in our simulation, and for the purposes of this thesis, they provide sufficient control over the behavior of the network resource.

At the server, a data read or write request arriving through the network is inserted into the *Ready Queue*, in which it awaits service. A multi-programming level is defined at the server to limit the maximum number of client requests that can be active at a time, and be competing for system resources simultaneously. This limit is enforced by the MPL module.

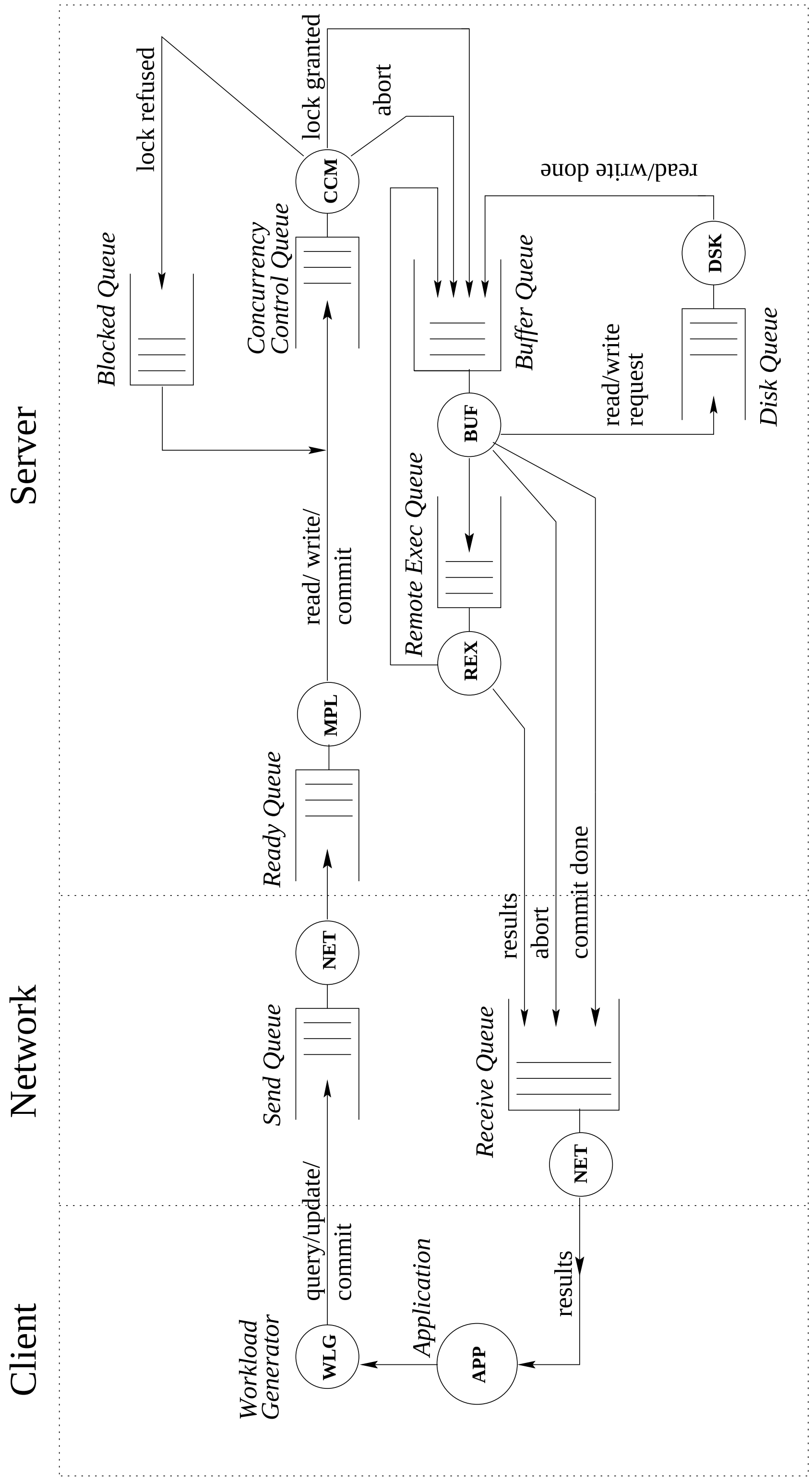


Figure 6.1: *Logical Queueing Model of a No-Cache System*

Thus, a job remains waiting in the Ready Queue until the number of active requests become less than that specified by the multi-programming level. The MPL module helps control the amount of concurrent data access at the server.

Active jobs that need to acquire locks are queued for service at the Concurrency Control Manager (CCM); this module processes all lock requests using the standard lock-compatibility matrix (details of this locking algorithm can be found in [Gray93b]). A query or an update first tries to get all locks that are necessary for its execution. If a lock request is denied, then the job waits in the blocked queue until the lock is available. The Concurrency Control Manager also implements a deadlock detection mechanism that looks for cycles in a *Waits-For* graph maintained for active jobs, using the algorithm described in [Gray93b]. In case a cycle is found among blocked jobs, the job that consumed the least amount of processing time is aborted, and all its locks released. In this case, the Concurrency Control Manager sends an *Abort* message to Server Buffer process (BUF), which performs the necessary clean-up for the writes made by the transaction, and then forwards the *Abort* message through the Network module to the client application.

When all locks have been successfully obtained for a query or an update, the request is passed through the Buffer module to the Remote Execution Unit (REX) at the server. This Execution Unit processes the request by sending tuple read or write requests, as appropriate, to the Buffer module. The Buffer process checks its list of pages cached in main memory, and in case of a miss, submits a disk read request to the Disk module (DSK). When the read is complete, the disk page is loaded into the main memory of the server buffer. The Execution module then takes over further processing, and either applies a query predicate to the tuple or performs an update, as appropriate, storing any modified tuples in the buffer.

After the last tuple in a query or an update has been processed, the results are sent back to the client via the Network module. The client application module processes the returned results, and finally submits a commit request. A successful commit causes the update log records of the transaction to be written out to the server disk, and then releases all locks acquired by the transaction. Pages updated by the client remain in the server buffer, to be written out to disk by a DBWriter process (not shown in the figure), that runs in the background and flushes pages periodically or on demand when the buffer is full. At the client, the completion of a commit request causes a new transaction to be generated by the Workload Generator, and the cycle repeats.

6.4.2 Logical Queueing Model of A*Cache

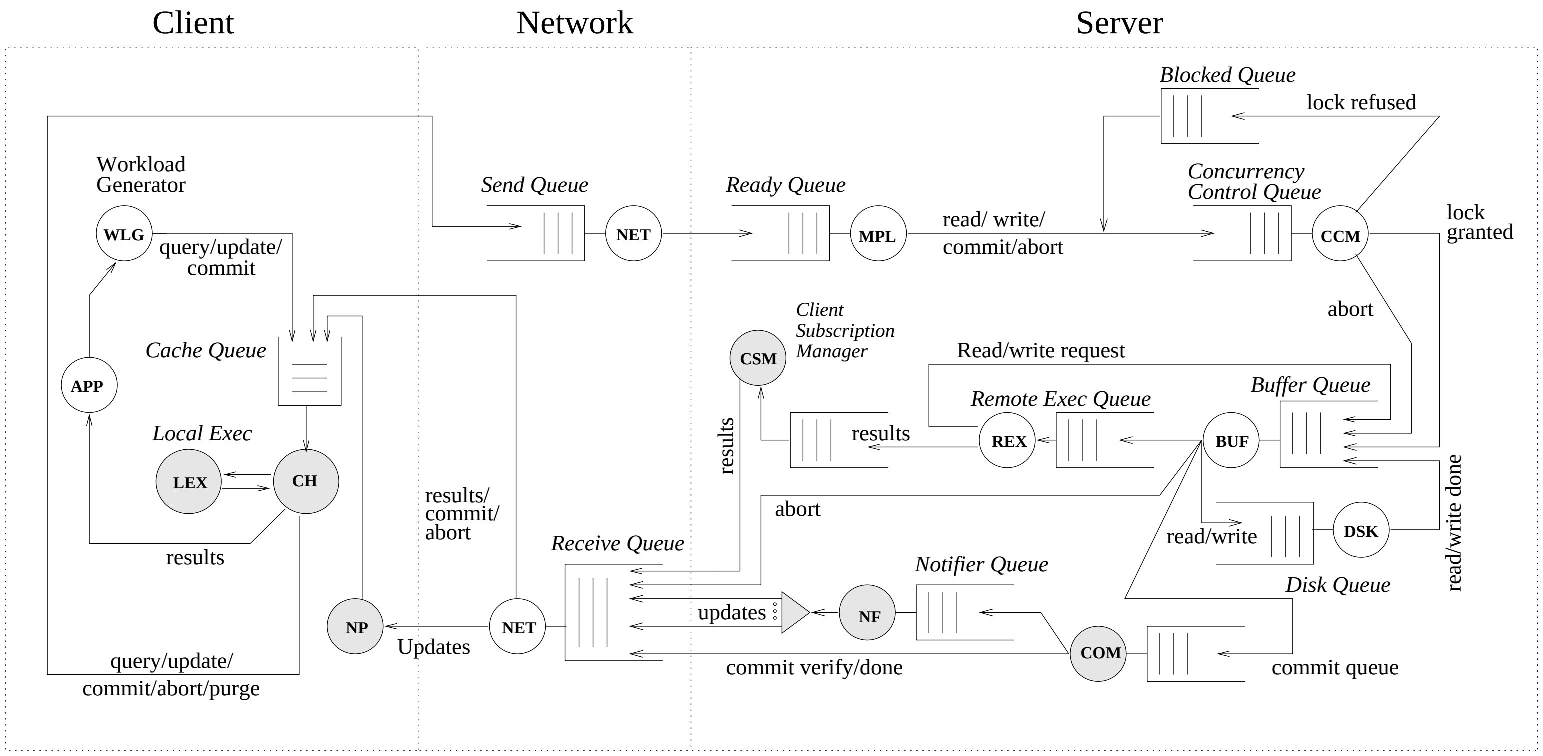
A queueing model that represents the logical processes in A*Cache is depicted in Figure 6.2. The model for this system is an extension of the No-Cache one, with additional modules at the client and the server to support data caching at the client. These extra modules are shown shaded in the figure, and are described below.

Queries and updates are generated by the Workload Generator (WLG) and submitted to the Cache module (CH), which checks its cache description to determine whether the required data is available locally in the cache. In the case of a cache miss where the data is not found locally, a request is sent to the server through the *Send Queue* of the Network module (NET), and the results received from the server are placed in the cache.

The cached predicates and tuples are managed by the local Cache module at the client, which also maintains the cache description and tuple reference counts for space reclamation. Space may need to be recovered when a new predicate and its associated tuples are inserted into the cache, and previously cached predicates must be flushed as necessary. In our simulation, we use an LRU policy at the level of predicates for cache replacement. When a predicate is aged out of the cache, each associated tuple that has a reference count of 0 (i.e., has no references by any other cached predicate) is also flushed (as described in Section 3.3.6). Removing predicates from the cache causes the Cache module to submit a *Purge* message through the *Send Queue* of the Network module to the server, indicating that the client subscription at the server is to be updated to remove the specified predicate.

Local data availability implies execution of the query or update at the client, and a request for execution on the local store is forwarded to the Local Execution Module (LEX) at the client. This Execution Module submits data read and write requests to the cache, sending the results of the operation to the Application module for processing. After the client application has processed all the data, it sends a commit request to the server via a *Commit* message submitted to the *Send Queue* of the Network module. Transaction commits are verified by a Commit Manager (COM) at the server, which communicates with the client if necessary, to ensure that serializability constraints were not violated for the transaction. As described in Section 3.3.3, sequence numbers of notification messages are used for this purpose.

During remote execution at the server on a cache miss, the job is entered into the queue



Shaded modules denote special components introduced for the A*Cache scheme.

Figure 6.2: *Logical Queueing Model of an A*Cache System*

of the Client Subscription Manager (*CSM*) module after the last tuple in a query or an update has been processed, and before the results have been sent to the client. The *CSM* module is responsible for registering and removing query predicates in client subscriptions. After a new subscription has been registered, the results of the operation are routed from the server to the client by queueing them at the *Receive Queue* of the Network module. The client Application module processes the returned results, and finally submits a commit request, provided it was not already aborted by notification of conflicting updates.

Additionally, there is a Notifier process (*NF*) at the server that is activated at each commit to generate notifications by checking the updates made by the transaction against the client subscriptions registered by the Client Subscription Manager. The Notifier process searches for overlap of client subscriptions with updated predicates in order to generate the appropriate notifications. Client caches that are found to be affected by an update are sent all tuples updated by the committed transaction, so that the cached data can be refreshed by the client.

The client also has a Notification Processor (NP) module that receives update messages sent by the server through the Network. The Notification Processor uses containment reasoning in the Cache module to determine if the update affects the cache contents, and to update the cached results if necessary. A transaction that is currently executing at the client could also be aborted by notification of a conflicting update in order to maintain the serializability property. In this case, the Cache module generates an *Abort* message that is sent to the server through the Network module. This message is also sent to the Application module, which rolls back and restarts the transaction.

Logical queueing model for the Tuple-Cache system is similar to the A*Cache model described above — the functional modules are the same, but as discussed in Section 6.3, there are differences in query and update processing. For example, cache containment reasoning in Tuple-Cache is performed at the server using central indexes, and notification and space management are based on individual tuples and not query predicates. Likewise, the model of A*Cache_Opt has the same functional modules as A*Cache, with different a query processing policy for partial cache hits and remote execution of updates on a cache miss, as described in Section 6.3 above.

6.5 Physical Resource Model

In this section, we describe the various physical resources in the system, such as the CPU, the disk, and the network, and the associated simulation parameters. Most of these resources are common to all of the four caching schemes that we investigate, the major exception being the cache module and associated containment checking and query execution costs, which are relevant only for the A*Cache scheme and its extension A*Cache_Opt.

In the following description, the simulation parameter corresponding to each resource is noted beside it within parentheses. We also specify the default values of the input parameters used in our experimental setup; these values have been carefully chosen to represent a typical environment of current client-server databases. Some of the parameters are varied in our experiments, and these variable parameters are noted as applicable. An experiment may vary a certain parameter around its default value while other parameters are held fixed, in order to investigate the system sensitivity to this parameter. Unless otherwise noted, an experiment in which a given parameter is held invariant uses its default setting specified in the tables below.

6.5.1 Client Configuration

There are a variable number, *NumClients*, of clients in the system. Each client has a single CPU with *ClientCPU* MIPS, which is modeled as a First-Come-First-Served service, and a main-memory cache of variable size (*CacheSize* bytes). The unit of data storage in the cache is a tuple of size *TupleSize* bytes. The instruction cost of comparing two predicates for overlap or containment is represented by the parameter *CompInst*, which comes into play during cache containment reasoning and notification processing. In Table 6.1 below we list the client parameters and their default settings.

The costs of caching and transaction processing at the client are modeled in terms of the parameters noted in Table 6.1. For example, the worst-case cost of detecting cache containment in the A*Cache and A*Cache_Opt schemes is represented by the following formula:

$$ContainCost = \frac{(Number\ of\ cached\ predicates) * CompInst}{ClientCPU}\ instructions$$

Thus, the cost of containment reasoning varies depending on the number of queries

Parameter	**Description**	**Usage**	**Default Value**
NumClients	Number of clients in the system	Variable	10
ClientCPU	Instruction rate of client CPU	Variable	25 MIPS
CacheSize	Size of client cache	Variable	5% of database
TupleSize	Size of each tuple	Fixed	200 bytes
AccessInst	Cache access cost per tuple	Fixed	50 instructions
SelectInst	Per-tuple cost of applying a predicate	Fixed	100 instructions
ApplInst	Tuple-processing cost in application	Fixed	1000 instructions
CompInst	Compare two predicates for overlap or containment	Variable	1000 instructions

Table 6.1: *Client Parameters and their Default Settings*

already stored at the client; in the worst-case, all predicates have to be checked, with no containment or overlap being found with respect to the current query.

6.5.2 Network Configuration

The parameters of the network resource are shown in Table 6.2. The network packet, which is the unit of data transfer across the network, has a size of *PacketSize* bytes. The network is a simple First-Come-First-Served service that models traffic in both directions, with a delay that depends on the *NetBandwidth* parameter that specifies the speed of the network in MBits/sec. The CPU costs of sending and receiving messages across the network are also taken into consideration. The network costs of message transmission include the actual wire time, as represented by the *NetBandwidth* parameter, a fixed CPU cost of *MsgInst*, plus a per-packet CPU cost *PacketInst*.

Parameter	**Description**	**Usage**	**Default Value**
NetBandwidth	Bandwidth of the network	Variable	10 MBits/sec.
PacketSize	Size of a message packet	Fixed	4096 bytes
MsgInst	Per-message send/receive cost	Fixed	20,000 instructions
PacketInst	Per-packet send/receive cost	Fixed	12,000 instructions

Table 6.2: *Network Parameters and their Default Settings*

6.5.3 Server Configuration

The resources on the server side consist of a single CPU with *ServerCPU* MIPS, and a single disk having an average page access time of *DiskSpeed* milliseconds. A main-memory buffer of *BufferSize* bytes is utilized for avoiding disk traffic. Pages of fixed size 4 Kbytes each are read from the disk into the buffer pool following an LRU page replacement policy. The CPU and disk are each modeled as a First-Come-First-Served single-server single-queue service. Each disk access involves a CPU cost of $DiskInst$ instructions. Costs are assigned for lock and unlock actions on tuples (*LockInst*), and for applying a selection predicate to a tuple ($SelectInst$). The server parameters and their default settings are shown in Table 6.3.

Parameter	**Description**	**Usage**	**Default Value**
$ServerCPU$	Instruction rate of Server CPU	Variable	50 MIPS
$BufferSize$	Size of the server buffer	Variable	25% of database
$PageSize$	Size of a data page	Fixed	4096 bytes
$DiskSpeed$	Average time to access a disk page	Fixed	6.5 msecs.
$DiskInst$	CPU cost for a disk I/O	Fixed	5000 instructions
$LockInst$	Per-tuple cost of lock/unlock	Fixed	400 instructions
$SelectInst$	Per-tuple cost of applying a predicate	Fixed	100 instructions
$RegInst$	Subscribe/unsubscribe a predicate	Fixed	100 instructions
$CompInst$	Compare two predicates for overlap or containment	Variable	1000 instructions

Table 6.3: *Server Parameters and their Default Settings*

The server in the A*Cache and A*Cache_Opt systems notifies clients of database updates based on their registered subscriptions. The cost to register or de-register a predicate in the subscription is $RegInst$. During notification of updates by the server, costs are incurred in comparing each update predicate with the predicates in client subscriptions for inclusion or overlap; this cost is represented by the $CompInst$ parameter, which is also used at the client to model cache containment and notification processing costs.

In our simulation, if an update predicate is found to overlap the registered subscription for a client, then all updated tuples for that update command are sent to that client; individual tuples or other predicates in the subscription of that client are not checked. Thus, some updated tuples may not actually be cached by the client, and are ignored by it upon receipt. This behavior is an example of the liberal notification mechanism discussed earlier in Section 4.1. In the worst case, an update in the A*Cache and A*Cache_Opt schemes causes inspection of the subscriptions of all clients but the client i making the update, incurring a cost of:

$$NotifyCost = \frac{\sum_{c \neq i}(\#\ of\ predicates\ cached\ by\ c) * CompInst}{ServerCPU}\ instructions$$

6.6 Workload Model

In this section, we describe the workload model adopted for our simulation experiments. The choice of the workload is critical in a performance study, since it determines the results to a large extent. We have carefully selected a workload profile that allows us to examine the costs and benefits of our caching schemes in a client-server scenario representative of real systems.

6.6.1 Database Benchmarks

Database benchmarks were developed in order to provide common environments for performance evaluation studies, in terms of well-defined system configurations and controlled patterns of workload. A benchmark usually consists of a synthetic workload that models the characteristics of a particular application domain. A synthetic benchmark, such as the one used in our study, has a close resemblance to a real environment, but is easy to create, and defines a relatively simple yet representative scenario for interpreting the simulation results meaningfully.

In our simulation study, we base our workload model on the Wisconsin benchmark [Gray93a]. This benchmark is a well-known standard one, and is suitable for our study, since it is directed towards relational databases, and uses associative queries. The benchmark covers many different query types, including projections and joins. We focus on a specific subset of this benchmark, namely, on clustered and unclustered selections that access data through associative predicates, and extend it to include associative updates. Although join and projection queries can be supported in A*Cache (see earlier discussion in Sections 3.3.2 and 3.3.4), we have omitted these two types of queries in our simulation for simplicity, and for easier interpretation of the simulation results. Examples of our queries and updates are provided below.

6.6.2 A Wisconsin-Style Database Setup

We assume a single relation with 10,000 tuples of 200 bytes each in a Wisconsin-style setting [Gray93a]. The relation has two uniquely indexed attributes *Unique1* and *Unique2* that are unclustered and clustered respectively. These attributes are unique for each tuple in the relation. Each query is a linear range selection either on *Unique1* or on *Unique2*, its length in terms of the number of tuples retrieved being a variable quantity. A sample query is shown below:

```
SELECT * FROM table
WHERE attribute BETWEEN lower AND upper;
```

In the above query, *attribute* is either $Unique1$ (indexed and unclustered) or $Unique2$ (indexed and clustered). The *lower* and *upper* quantities are boundaries that define the length of the query in terms of the number of retrieved tuples.

Updates in the Wisconsin benchmark are based on unique records, and hence are not suitable for our associative caching scenario. We have instead incorporated associative updates in our model, using the associative queries defined in the benchmark as a basis for our extension. A sample update command appears below:

```
UPDATE table
SET update_attribute = value
WHERE access_attribute BETWEEN lower AND upper;
```

In the above update, *access_attribute* is either $Unique1$ or $Unique2$, depending on whether the case is unclustered or clustered. For the updates in our workload, we assume that the *update_attribute* is different from both $Unique1$ and $Unique2$, so that index updates are not involved (i.e., values of $Unique1$ and $Unique2$ attributes are not modified by the update).

6.6.3 Representation of Locality and Contention

We extend the Wisconsin benchmark to integrate the locality models of [Fran93, Care94], which represent locality of data reference and update contention on data shared among different clients.

The database is logically split into a *hot* region and a *cold* one. Each client *owns* a section of the hot region. With respect to the BUG example introduced in Section 1.1.2, the hot region can be considered to be the set of active bugs that have been logged within the last three months. The rest of the BUG database consists of bugs that are either closed or obsoleted, and thus constitutes the cold region which is not accessed as frequently.

Data access probabilities *HotAccess* and *ColdAccess* are defined for the hot and cold regions of the database respectively. Update probabilities *HotTransWrite* and *ColdTransWrite* are also associated with transactions that execute in the hot and cold regions respectively. These quantities are defined on a per-transaction basis, and are set to the same value for every client. The parameter $HotTransWrite$ denotes the probability that a transaction in the client-specific hot region is a read-write transaction, and $ColdTransWrite$ represents the same quantity for the shared cold region. Additionally, the parameter $HotTupleUpdate$ denotes the probability that a read-write transaction in the hot region updates a tuple read by it, while $ColdTupleUpdate$ denotes the same quantity for a read-write transaction in the shared cold region. Contention of shared data can be controlled either by increasing the value of *ColdAccess*, or by increasing the update probabilities appropriately.

Table 6.4 shows our workload parameters. 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. As with all caching studies, not the absolute values but only the relative ratios of these different sizes are relevant for performance.

The Private Data Access Model

In the *Private* access model [Fran93, Care94], the hot range of a client i is defined by the following linear interval of tuples:

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

The cold region is the same for all clients, and is defined by:

$$[HotRatio * DBsize, DBsize) .$$

The access pattern of this model is illustrated in Figure 6.3. The hot region of a client

Parameter	**Description**	**Usage**	**Default Value**
DBsize	Database size	Fixed	10,000 tuples
TupleSize	Tuple size	Fixed	200 bytes
QueryLength	Query length as % of DBsize	Variable	0.1%
HotRatio	Hot region as % of DBsize	Fixed	50%
HotAccess	Access frequency of client-specific hot region	Variable	80%
ColdAccess	Access frequency of client-specific cold region (=1 − $HotAccess$)	Variable	20%
HotTransWrite	Probability that a transaction in the hot region writes data	Variable	20%
ColdTransWrite	Probability that a transaction in the cold region writes data	Variable	20%
HotTupleUpdate	Probability that a read-write transaction in the hot region updates a tuple	Fixed	50%
ColdTupleUpdate	Probability that a read-write transaction in the cold region updates a tuple	Fixed	50%

Table 6.4: *Workload Parameters and their Default Settings*

is private to it, in that it does not overlap with the cold region of any other client. The per-client cold region is equal to the cold region of the entire database. Thus, each client has a non-shared hot region, the same cold region as all other clients, and a portion of the database that it does not use (which is the union of the hot regions of all the other clients).

In terms of the BUG example introduced in Chapter 1, the Private access model fits the scenario where a client has its own 'internal' set of bugs that is not readable or writable by anyone else. The Private model also reflects the case of organizational specialization in an enterprise; for instance, each separate department in a school may create its own special region of data that, though residing in a common database, is either not available for use by any other department, or not of interest to them.

The HotCold Data Access Model

In the *HotCold* access model [Fran93, Care94], the hot region of client i is defined exactly as in the Private case above, but the remainder of the table represents the cold region of client i. Thus, the hot region of a client is part of the cold region of all other clients, but

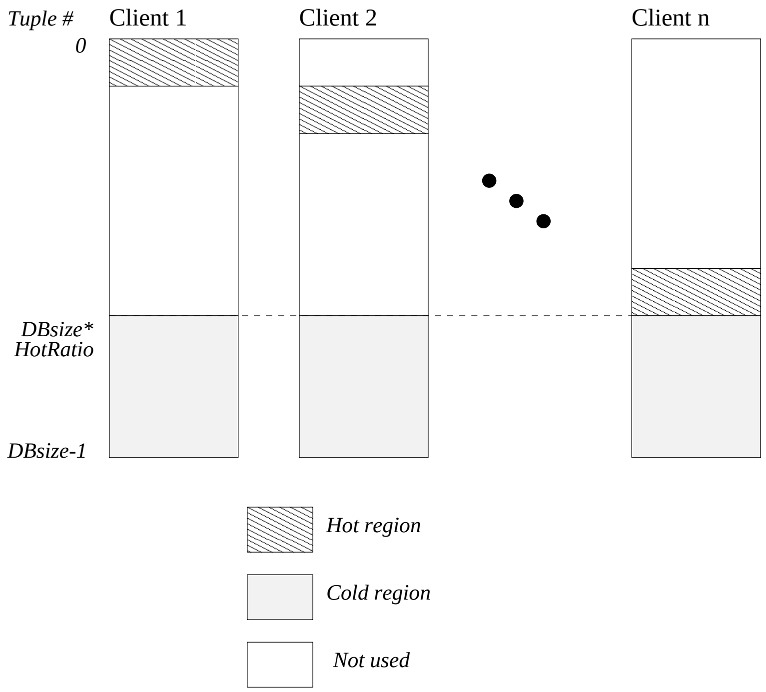


Figure 6.3: *Data Access Pattern for the Private Model*

the hot regions of the different clients are mutually disjoint. This access model is depicted in Figure 6.4.

The HotCold model is quite appropriate for the BUG example described in Section 1.1.2. Users may have their own area of interest, depending on the product they deal with, but they might also be interested in bugs logged for other products in the database.

In addition to the Private and HotCold models, we consider two other data access patterns, namely, the Hicon and Uniform access models. These two access patterns are not representative of common real-life scenarios, but are included here as bases of comparison for the results of our performance study. These access models are described below.

The HiCon Data Access Model

The *HiCon* access model [Fran93, Care94] defines a single hot region of the database that is common to all clients. The rest of the database constitutes the cold region of each client. This access pattern is shown in Figure 6.5. The hot region is defined by the following linear

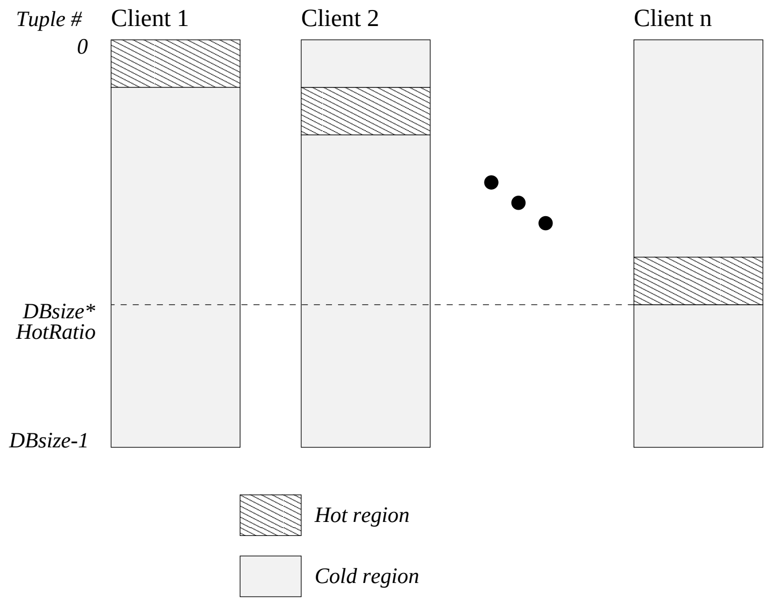


Figure 6.4: *Data Access Pattern for the HotCold Model*

interval of tuples:

$$[0, HotRatio * DBsize) .$$

For our BUG database example, the HiCon model represents a scenario where some common bugs are heavily updated by multiple concurrent users. Although this situation is unlikely to occur in practice, it reflects high update contention on shared data. Another scenario where the HiCon model applies is the case of index updates. Indexes are normally the heavily-used 'hot spots' in a database; if many new tuples are inserted in the database concurrently by different users, or the indexed attributes are updated, then the indexes would be subject to high contention.

The Uniform Data Access Model

In the *Uniform* access model, there are no specific hot or cold regions. The queries of each client are directed uniformly to all attribute values. Figure 6.6 shows the Uniform access pattern.

In Table 6.5, we summarize the hot and cold ranges of the clients in the four different access models described above.

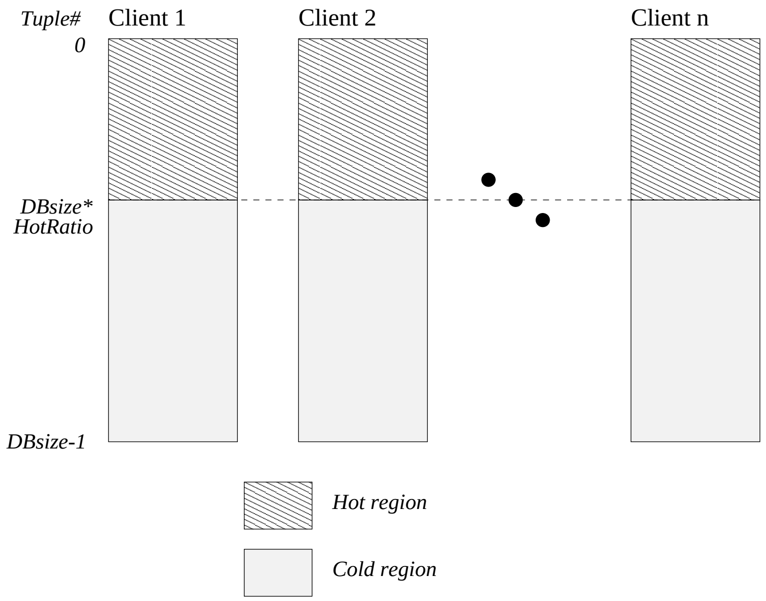


Figure 6.5: *Data Access Pattern for the HiCon Model*

6.7 Implementation Notes for the Simulator

The simulator is implemented in C++/CSim [Mesq94], and consists of about 5000 lines of code. C++/CSIM is a discrete event-driven simulation language based on C++. Below we provide an overview of the relevant CSIM features as used in our implementation of the simulator.

- A CSIM program models a simulation system as a set of *processes* that interact with each other using *events* or *mailboxes*. The CSIM system simulates the parallel operation of multiple active processes, even though they are in fact executing sequentially on a single processor. For example, each client is modeled as a separate process, operating in parallel with other client processes.

- Resources in the simulation system that provide certain services, such as a CPU or a disk, are modeled directly as single-server-single-queue *facilities* in CSIM. A CSIM facility has built-in support for management of its service queues in which processes wait while the server is busy. A process can *reserve*, *hold*, or *use* a facility for a specific amount of time, with the order of service depending on the policy defined for the facility. For our simulation, the disk and CPUs use First-Come-First-Served service discipline.

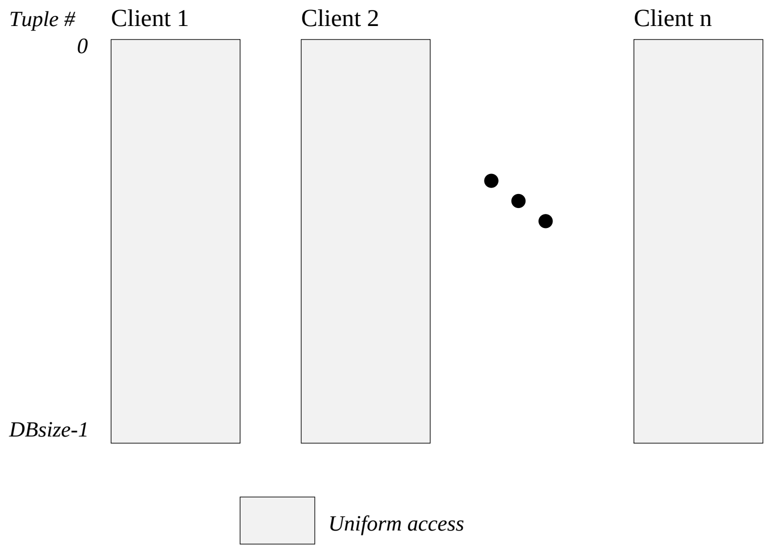


Figure 6.6: *Data Access Pattern for the Uniform Model*

- Completion of a service request is indicated by a CSIM *event*. Thus, events are used to synchronize the operations of CSIM processes. For example, if a page is not found in the buffer, a request to read the page is submitted to the Disk facility. When the read is completed (depending on the jobs in the disk queue), a 'ReadComplete' event occurs. This event indicates to the buffer that the page can now be placed in the main memory of the buffer pool.

- Notification in our caching systems is triggered by a transaction commit. This scenario is modeled using CSIM *mailboxes*. A mailbox is a container for sending and receiving CSIM messages that allow processes to communicate asynchronously. In our simulation, the Notifier is a server process that 'wakes up' upon receiving a message in its mailbox from a client transaction that has successfully committed its updates. Once the message has been deposited in the notifier mailbox, the client transaction continues on, while the Notifier starts generating notification for all other clients based on their registered subscriptions.

The cache containment reasoning logic in a client process is implemented as follows. For our workloads, the query predicates are linear integer intervals on either the attribute *Unique1* or on *Unique2*, corresponding to range queries on these attributes. Thus, in our simulation of the A*Cache and A*Cache_Opt schemes, the predicate of each query stored in

Access Model	**Hot Range for Client i**	**Cold Range for Client i**
Private	$[r*(i-1), r*i)$, $r = \frac{HotRatio*DBsize}{NumClients}$	$[HotRatio*DBsize, DBsize)$
HotCold	$[r*(i-1), r*i)$, $r = \frac{HotRatio*DBsize}{NumClients}$	Rest of the table
HiCon	$[0, HotRatio*DBsize)$	Rest of the table
Uniform	$[0, DBsize)$	—

Table 6.5: *Hot and Cold Ranges for Different Access Models*

the cache description is in the form of an integer interval on the attribute constrained by the query. The predicates are organized by their query attribute, either $Unique1$ or $Unique2$. Within each attribute category, the set of predicates is maintained as a doubly-linked list in reverse order of entry. When examining the cache description for query containment and update notifications, the list of cached predicates for the matching attribute is scanned in sequence. If the interval ranges of one or more cached predicates together contain the new predicate, then cache containment reasoning indicates a complete hit. If there is an overlap of the query interval with cached predicates, a partial hit is detected in the case of A*Cache_Opt scheme.

Predicates that represent a new query to be inserted in the cache description at a client are placed at the head of an LRU list called the *ageList*. Thus, the predicates that are used less often migrate towards the tail of this list. When the cache is full, as many predicates as necessary are purged from the tail of the $ageList$, and the new predicate is inserted at the head of the list.

Further details on the implementation of the simulator can be found in [Poes97].

6.8 Experimental Methodology

There are three basic ways of investigating the behavior of a complex system — using analytical means, empirically through experiments, or by simulation. Analytical examination can provide fast and accurate results, provided that the underlying model is small and tractable. The A*Cache system, unfortunately, is too complex to be modeled analytically. In contrast, empirical investigation requires observation of system behavior through carefully-designed experiments, which are run either on a real implementation of the system, or on a simulated one. Developing a real A*Cache system was beyond the scope of this thesis, so we had to investigate system behavior using detailed simulation.

Simulation is a powerful yet complex technique for performance evaluation. As is evident from the discussion in the preceding sections, there are many parameters in the model that can be varied experimentally. In order that the simulation results are meaningful, careful consideration must be given to several factors:

- The system model must be a sufficiently accurate representation of the real system, and yet be tractable. The process of simulation modeling inevitably involves some simplifications and approximations over the real system; however, these simplifications should be chosen such that they do not substantially impact the relative performance of the schemes being studied.

  For example, consider the assumption we make in our simulation that the client stores data in main memory only. In a real system, disks might also be employed at client sites. However, none of our caching algorithms depend on the specific nature of the client-side data store. The performance figures would certainly be different in the presence of client-side disks; however, the *relative* performance of the different caching schemes is not expected to be affected by this assumption.

- Workloads are another important consideration. Ideally, a trace of a real system should be used as input. However, this approach is inflexible, and the required data can be hard to obtain. We have used a workload based on a real database benchmark, and extended it with the well-defined locality and contention models of [Care94] for multiple clients.

- Input parameter settings and their scope of variation must be chosen with care, since they directly influence the experimental results and conclusions. In our case, we have based the parameter values on realistic systems, as well as on prior performance studies of client-server systems, e.g. [Fran93, Wilk90].

- Output parameters must be defined appropriately, their behavior corroborated with the help of other output measures, and checked for statistical validity. For example, the behavior observed in an experiment with respect to the response time must be supported by other output parameters, such as server costs, network traffic, and disk accesses. Therefore, for all our experiments, we examine multiple output parameters to ensure consistency of the results. Also, all experiments were performed for a large number (500 or more) queries at each client, so as to minimize transient effects of the warmup of client caches and the server buffer. The server buffer was also populated randomly with pages at start-up, to reduce the time required to reach steady state.

- Finally, the implementation of the simulator must be carefully debugged and validated, to avoid producing invalid results and false conclusions. We have analyzed in detail the execution traces of 'small' experiments, in order to verify the correctness of simulator operation. Additionally, the operation of the No-Cache simulator was validated against a commercial database system, as described in Section 6.9 below.

6.9 Validation of the Simulator

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. The goal of the validation procedure is to verify that our model of the system with no client-side caching behaves reasonably close (within $\pm 20\%$) of a real database system. Below, we provide a brief description of the validation environment and results — full details can be found in [Poes97]. [1]

6.9.1 System Configuration

The database system used for the validation is the Oracle 7.3.2 database, in a single-client single-server configuration. Both the client and server processes were on the same machine,

[1] This part of the research is joint work with Meikel Poess and Kurt Shoens — Meikel ran validation experiments on the simulator developed jointly by the two of us, and Kurt Shoens performed the experiments on Oracle.

so that network costs were not involved. The Oracle 7.3.2 server ran on a 50 MIPS HP 735 machine with a 125MHz clock and the OS THP 10.0 operating system. It had a 50K bytes main-memory buffer for buffering disk pages of size 4KBytes each. The simulator parameters were tuned to match the characteristics of the real database system as closely as possible.

6.9.2 Queries

A single database relation with two indexed attributes, one clustered and the other unclustered, was created. Two types of queries were considered in our validation experiments — clustered and unclustered selections. Since only one client was present, the multi-client data locality and contention model (i.e., Private, HotCold, or HiCon access) was ignored, queries being targeted uniformly over all the tuples in entire database relation.

The database relation had 300,000 tuples, and the result returned by each type of query was 1% of the relation (i.e., 30,000 tuples). Note that the size of the database relation was scaled up 30 times in the validation experiments, compared to the database size (10,000 tuples) used in the subsequent experiments for performance evaluation. The size of each tuple was the same in both the validation and performance experiments, i.e., 200 bytes. Thus, the size of the database for the validation experiments was 300,000*200, i.e., 60 MBytes, and 2 MBytes for the simulation experiments. This scaling up was necessary to minimize the effect of file buffering by the underlying operating system in the HP machine running the database server.

6.9.3 Validation Methodology

Query traces were generated for the validation experiments using a Workload Generator. The generated queries were run on the Oracle database, and our simulator was run in a trace-driven mode. For the validation experiments, two output parameters were measured, namely, the query response time and the number of disk I/O operations. The experimental results in terms of these two parameters are given below.

6.9.4 Validation Results

Query Response Times

The query response times of the simulated system and for the Oracle database are shown in columns 2 and 3 of Table 6.6. The difference in the two numbers appears in column 4, while column 5 shows the difference as a percentage of the response time of the simulated system. As can be seen from the table, the percentage difference is about 13%, being well within our target of 20%.

Query Type	**Query Response Time**		**Difference (Simulated - Oracle)**	
	Simulated	Oracle	Total	Relative to simulation
Clustered Selection	2.92 secs.	3.04 secs.	-0.12 secs.	4.11%
Unclustered Selection	116.20 secs.	101.14 secs.	15.06 secs.	12.96%

Table 6.6: *Query Response Times in Validation Experiments*

Number of Disk Reads

The number of disk reads that occurred in the simulated and Oracle systems are reported in Table 6.7. A significant difference (about 19%) is observed in the case of the clustered selection query. The total difference in terms of the number of disk reads is 761, but the relative deviation is high because there are only about 3,200 reads needed in the simulation to compute the entire query from the clustered data.

Further details on the validation experiments and results can be found in [Poes97].

6.10 Summary of Chapter

In this chapter, we have described the simulation models and experimental setup used in our performance study. Our simulation model for A*Cache is an extension of the client-server model used in [Care94]. We defined three alternate caching schemes for comparison of our results, namely, No-Cache, Tuple-Cache, and A*Cache_Opt. Differences in the query and

Query Type	**Number of Disk Reads**		**Difference (Simulated - Oracle)**	
	Simulated	Oracle	Absolute	Relative to simulation
Clustered selection	3211	3972	-761	19.16%
Unclustered selection	39892	37392	2500	6.6%

Table 6.7: *Disk Reads in Validation Experiments*

update handling policies of these schemes with A*Cache were noted. Queueing models for the logical processes in the simulated systems were presented. Characteristics of the physical resources in the system were specified in terms of simulation parameters. We then defined a workload model for performance evaluation. Our model of the workload is based on the single-user Wisconsin benchmark [Gray93a], extended with the locality and contention models proposed for multiple clients in [Care94]. Four different types of workloads, Private, HotCold, HiCon, and Uniform, are chosen to represent different contention profiles among the clients. Some implementation issues for our simulator were also discussed, including the validation process that was followed to test the correctness of our simulator by comparing it against a commercial database system.

Chapter 7

Performance Analysis of Read-Only Workloads

In order to study the performance of A*Cache, we conducted a variety of simulation experiments under different data access and update conditions. In this chapter, we report the results of read-only experiments in which transactions do not modify data, and hence there are no costs for update notification and cache maintenance. The read-only results are interesting in their own right, because they allow us to observe and analyze the system behavior with fewer factors in play, and provide a basis for interpreting the results of our subsequent experiments with read-write workloads. In many practical settings, read activity greatly exceeds update loads, and the read-only results are of interest in these situations.

The material in this chapter is organized as follows. First, we briefly review the major differences in the query processing strategies of our four caching schemes. Next, we specify the performance metrics that are used to evaluate and compare the behavior of the different simulated systems. We then focus on the *Private* access model defined in Chapter 6, and explore in depth the effects of various system and workload parameters. Results are reported graphically for several different output parameters, and the behavior observed in each experiment is analyzed. Many experiments were also performed with the other data access models, namely the *HotCold*, *HiCon*, and *Uniform* models; some interesting results from these experiments are presented. Finally, we summarize the conclusions of our performance analysis for workloads in the read-only environment.

7.1 Comparison of Query Processing Strategies

Each of the four schemes, No-Cache, Tuple-Cache, A*Cache, and A*Cache_Opt, processes queries quite differently. Below, we provide a summary of our discussion on this issue in the previous chapter.

- No-Cache always requests the server for data, since it has no client-side caching.
- 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. Thus, 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 re-using cached tuples more effectively.

The distinguishing characteristics of query processing methods in the four types of simulated systems are summarized in Table 7.1 below:

7.2 Read-Only Performance Metrics

Our primary performance metric is the response time, measured in terms of the time that elapses between the start of a transaction at the client and the time at which it commits

Query Processing Characteristic	**No-Cache**	**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 7.1: *Comparison of Query Processing Methods*

successfully. In order to understand the details of system behavior, we also track several other important simulation output, such as the hit ratios of the client caches and the server buffer, utilizations of client and server CPUs, the server disk and the network, the number of disk reads and writes, overhead for predicate comparisons, etc. Table 7.2 lists some output parameters that were monitored in our simulation experiments with read-only workloads. One or more of these output metrics will be used to analyze and explain our experimental results.

Output Parameter	**Description**	**Unit**
ResponseTime	Transaction start time – end time	Secs. per transaction
DiskReads	Number of disk reads	Number
DiskUtil	Utilization (busy time) of disk	% of elapsed time
CacheHitRatio	Ratio of queries evaluated in cache	% of total queries
BufferHitRatio	Ratio of tuples found in server buffer	% of buffer accesses
NetworkVolume	Data and message traffic on network	Kbytes per transaction
NetworkUtil	Utilization (busy time) of network	% of elapsed time
ServerUtil	Utilization (busy time) of server CPU	% of elapsed time
ClientUtil	Utilization (busy time) of client CPU	% of elapsed time

Table 7.2: *Output Parameters for Read-Only Workloads*

7.3 Effect of Client and Workload Parameters

In this section, we report the results of experiments done by varying several client and workload parameters in the Private access model, such as the size of the client cache, clustered and unclustered selection of data, hot data access probability, speed of the client CPU, and the cost of predicate comparison.

7.3.1 Cache Size

In this set of experiments, the size of the client cache is varied from 0% up to 40% of the database size, while the rest of the input parameters have their default settings shown in the Tables 6.1, 6.2, 6.3, and 6.4. For example, the frequency $HotAccess$ with which the client-specific hot region is accessed is set to 80%. In this experimental environment, the size of the hot region of each of the 10 clients is equal to 5% of the database size, and the size of the shared cold portion of the database is 50%. Thus, the hot regions of all 10 clients taken together is 50% of the database. The size of the server buffer is fixed at 25% of the database, and is not varied in these experiments.

Varying Cache Size for Clustered Data Selection

In the experiments with clustered data access, queries use the clustered attribute $Unique1$. Figure 7.1 shows the change in $ResponseTime$ and $DiskReads$ as the size of the client cache is varied from 0 to 40% of the database size. For the same experiment, the behavior of two other output parameters, namely $NetVolume$ and $ServerUtil$, are also shown in the figure.

For all the three caching schemes, a large drop in the output parameters shown here occurs in the 0% to 5% region of the cache size, a smaller drop being observed in the 5% — 10% region. The improvement in performance essentially 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 the environment of these experiments, a cache size of 5% is large enough to hold the entire hot region of a client, to which 80% of all accesses by the client are directed. Therefore, cache sizes smaller than 5% cannot hold all the tuples in the hot region of a client, causing remote trips and significant deterioration in the response time. Increasing the cache size to 5% produces rapid improvement due to higher cache hit

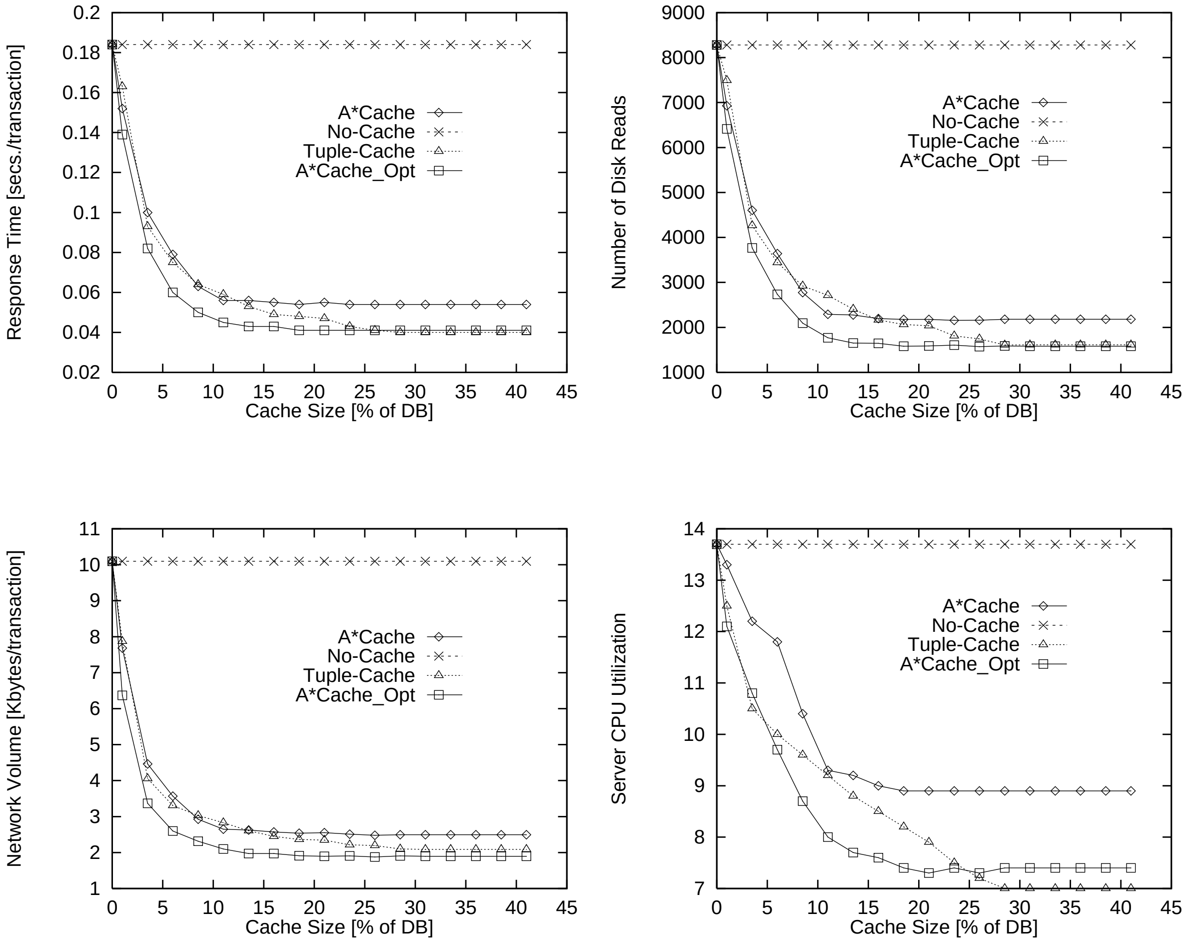


Figure 7.1: *Varying CacheSize, Private Clustered Access*

ratios. A cache size larger than 5% causes the client to store cold data in addition to its hot data, further reducing the number of remote trips, disk accesses, network volume, and response times.

In terms of the other output parameters, 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 more cache hits for larger caches, 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.

However, beyond a cache size of 10%, the response time saturates for all three caching schemes, showing no significant improvement with larger caches (upto 40% of the database size). This asymptotic behavior is a result of our experimental environment, in which cold data is spread uniformly over 95% of the database and is accessed 20% of the time; even a cache size of 40% is not large enough to significantly improve the cache hit ratio for the cold region. Additionally, the commit in all our schemes happens at the server, and hence in every scheme, a certain number of messages and server cycles are consumed for each transaction, even when all data is found locally in the cache. The response time therefore always has a remote component, and is never determined completely by local processing costs at the client.

Now, 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 the basic A*Cache scheme 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 the help of the other output parameters, $DiskReads$, $NetworkVolume$, and $ServerUtil$ shown in the figure. 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 resulting from fetching missing tuples only.

In terms of network volume, Tuple-Cache must send each query to the server for cache containment check, thereby sending more messages than both the A*Cache schemes. In contrast, basic A*Cache fetches all tuples in a query even when there is a partial cache hit, thereby incurring extra network cost in data transmission compared to Tuple-Cache. The relative difference in response times between Tuple-Cache and basic A*Cache is prominent for 15% and larger cache sizes, since the cache hit ratio is large enough to cause the savings in data volume for Tuple-Cache to offset its larger number of network trips and yield significant benefits. A*Cache_Opt performs the best, since it supports both local cache containment reasoning like A*Cache, and like Tuple-Cache, also utilizes partial cache hits to reduce the data volume across the network.

With respect to the utilization of the server CPU, No-Cache has the highest CPU usage compared to the response time, while values for all the caching schemes are at least 30% lower for cache sizes 15% and larger. Tuple-Cache uses the CPU less heavily than basic A*Cache, because it does not read tuples in the result that are already present at the client. In fact, the CPU utilization in Tuple-Cache is even lower than A*Cache_Opt for cache sizes 25% and larger. This behavior is due to the server cost of maintaining a large number of predicates for large-sized caches in the A*Cache schemes. As an example, consider a 30% cache size in our simulation. For this set of experiments, each query is .1% of the database size, with records 200 byte long, producing query results of 1 Kbyte each. Thus, when the cache size is 30% (600 KBytes), each of the 10 clients holds 600 query predicates. In our simulation, predicates are simple integer intervals but they are not indexed, so that the server has to examine a non-trivial number of predicates for a client when it is informed that a particular query result has been purged from the cache. This cost increases the server CPU load for bigger cache sizes in the A*Cache schemes, since our simulation does not use any optimization techniques for indexing or merging the predicates in a cache description. The cost of maintaining cache descriptions in the A*Cache schemes are expected to decrease with such optimizations, improving their performance for large cache sizes.

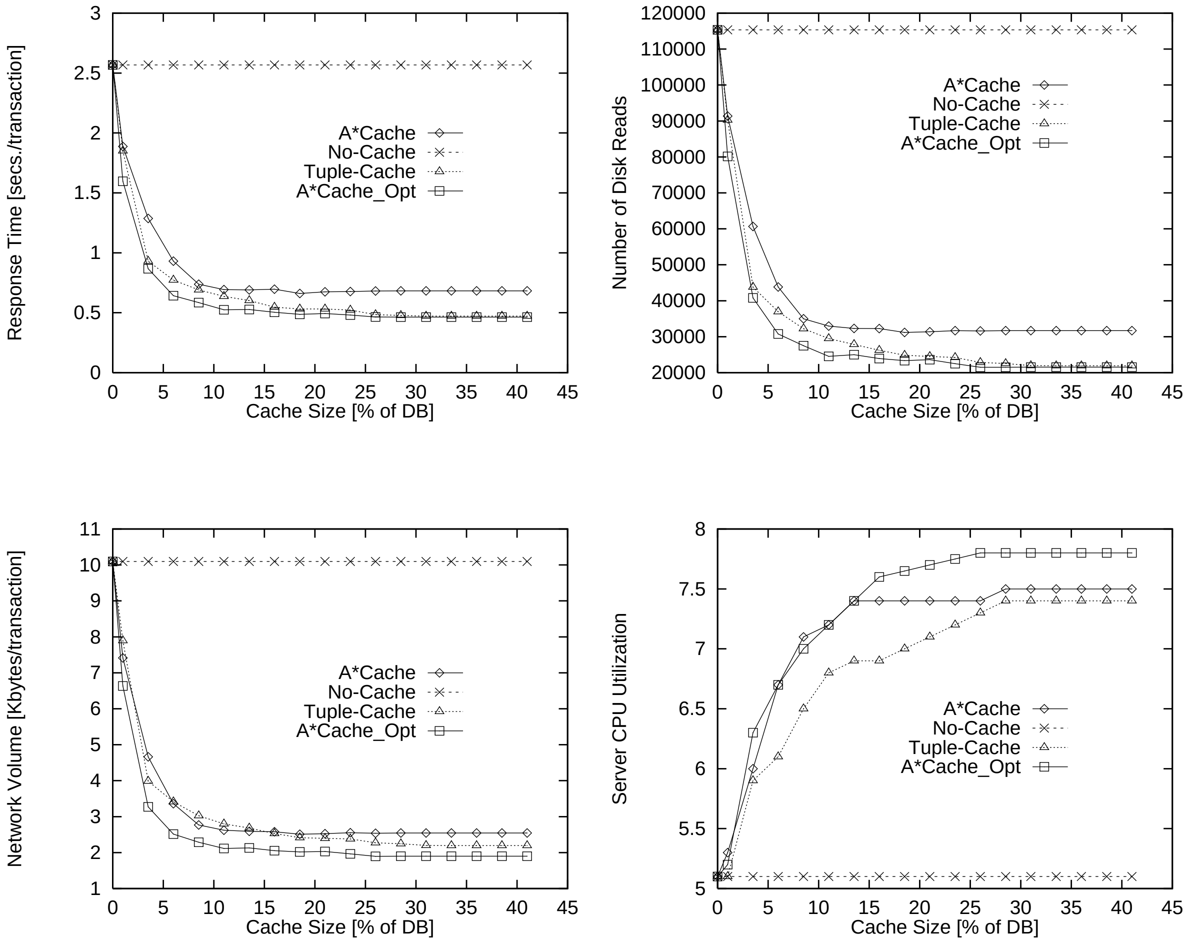


Figure 7.2: *Varying CacheSize, Private Unclustered Access*

Varying Cache Size for Unclustered Data Access

In these experiments, queries use the unclustered attribute $Unique2$ to select tuples. Unclustered data 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. Therefore, the benefits of client-side caching are more prominent compared to clustered access.

Figure 7.2 shows the effects of varying $CacheSize$ for unclustered reads, using default settings for all other parameters. A comparison with Figure 7.1 for the clustered results verifies that the number of disk reads are 10 times greater for unclustered access, and the response time is correspondingly an order of magnitude larger. A*Cache_Opt is again found to have the best response time, followed by Tuple-Cache and then basic A*Cache. Note that for cache sizes below 15%, the difference in response times between Tuple-Cache and basic A*Cache is more pronounced than in the clustered case. This interesting difference in response time characteristics is explained as follows. Consider the basic A*Cache scheme first. According to the execution model described in Section 6.2, on a cache partial cache hit, basic A*Cache fetches all tuples in the query result from the server, irrespective of whether some of these tuples are already in the client cache. The larger number of tuples accessed causes more disk reads, especially since reuse of pages in the server buffer is reduced for unclustered page access patterns. Thus, the cost of a cache miss is higher in the unclustered case, and reduces the relative performance of basic A*cache. The A*Cache_Opt scheme does not have this problem because of its partial cache hit optimization. In the scenario of this unclustered set of read-only experiments, A*Cache_Opt yields better response times compared to the Tuple-Cache scheme for cache sizes between 5 and 20%, and is nearly the same as Tuple-Cache for caches of other sizes.

Another interesting issue is the observed difference in server CPU utilization depending on whether data access is clustered or unclustered. For the clustered case (Figure 7.1), the server CPU utilization decreases steadily with increasing cache sizes, as more of the query processing costs are distributed to the clients. For unclustered access (Figure 7.2), CPU utilization at the server *increases* as cache size is increased. This increase is due to the unclustered pattern of page reads causing many more disk accesses than in the clustered pattern, so that processing at the server is more disk-bound than CPU-bound for small cache sizes. As the client cache size is increased, more tuples are found in the cache for the hot regions of clients, resulting in fewer server disk accesses and comparative increase in the server CPU time with respect to the total elapsed time for a query.

7.3.2 Hot Access Probability

In this set of experiments, the probability HotAccess of reading data in the hot region is varied for all the clients from 0% to 100%. Access frequency of the cold region changes accordingly. The rest of the input parameters have their default settings shown in the Tables 6.1, 6.2, 6.3, and 6.4. Each client cache is thus 5% of the database size. Results for both clustered and unclustered access modes are reported below.

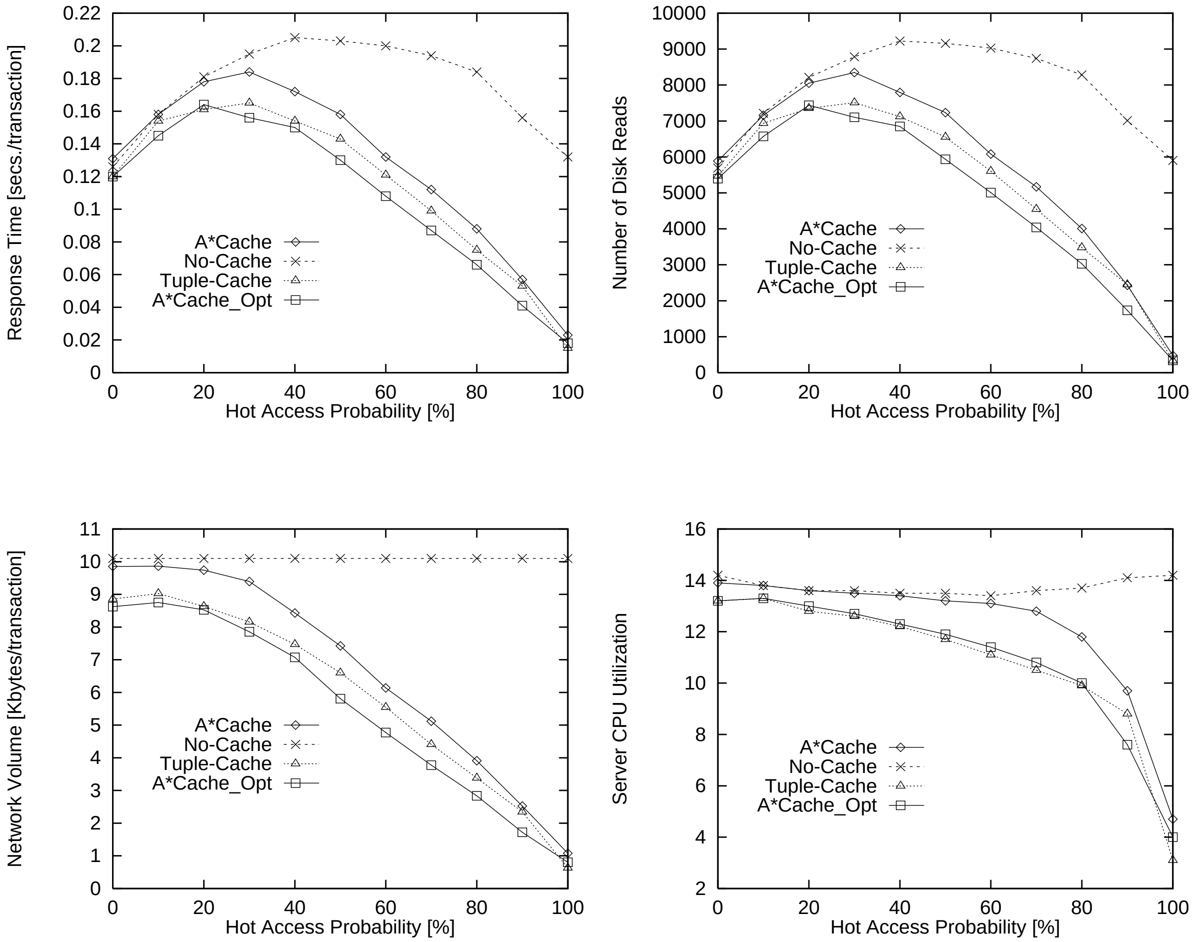


Figure 7.3: *Varying HotAccess, 5% CacheSize, Private Clustered Access*

Varying Hot Access for Clustered Selection

Figure 7.3 presents the results of varying HotAccess in the case of clustered data retrieval. For all the four schemes, the query response times schemes are found to first increase, reach a peak around 30% HotAccess, 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 Private workload model, and can be explained as follows.

For HotAccess values below 10%, most data accesses occur in the shared cold region of the database, which is 50% of the database. For these experiments, the server buffer is 25% of the database, i.e., nearly half of the cold region. Therefore, the buffer hit ratio is high small values of HotAccess, the buffered pages being effectively shared by all clients. The cold 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 HotAccess values upto 20%. The caching schemes start to perform better than No-Cache beyond 20% HotAccess, since data cached locally is better reused.

As HotAccess 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 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.

Varying HotAccess for Unclustered Selection

Figure 7.3 presents the results of varying the hot access probability from 0% to 100% for unclustered reads. The behavior of the ResponseTime, DiskReads, NetworkVolume, and ServerUtil output parameters are shown here.

Comparison with the results for the clustered case 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.

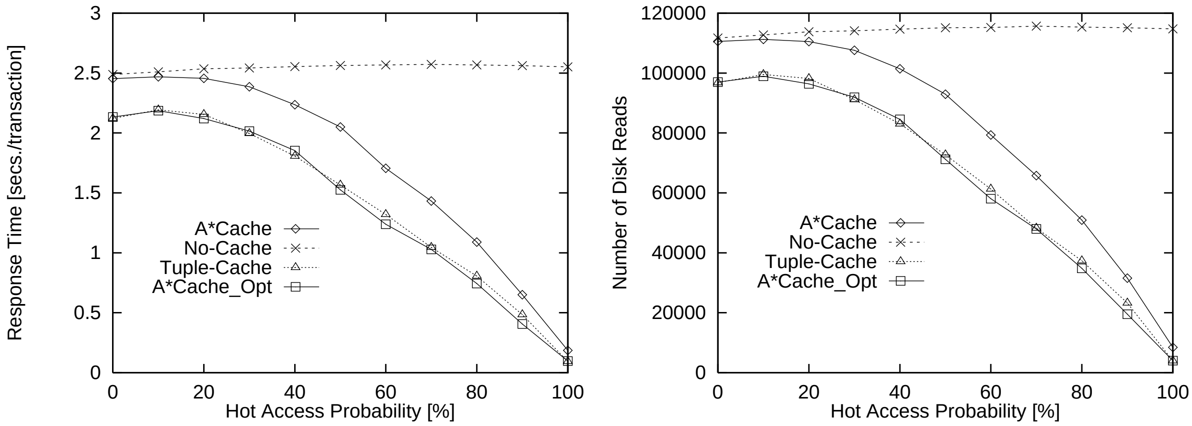


Figure 7.4: *Varying HotAccess, 5% CacheSize, Private Unclustered Access*

- The response of No-Cache is almost constant over the entire range 0 — 100% of HotAccess 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 HotAccess. For example, for 75% HotAccess 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 in Figure 7.3 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 cold 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 HotAccess 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.

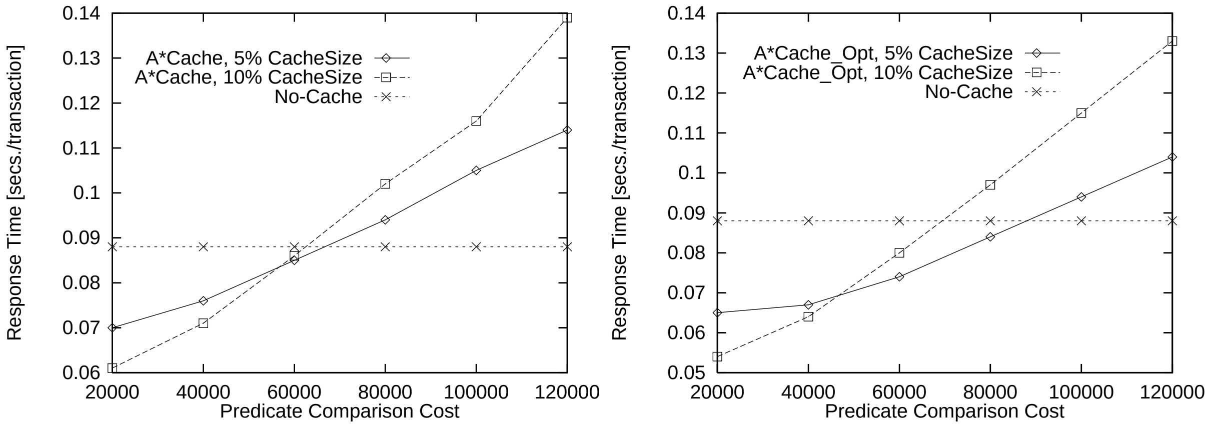


Figure 7.5: *Varying Predicate Comparison Cost, Private Clustered Access*

7.3.3 Predicate Comparison Cost

In Figure 7.5, we show the effects of varying the cost of predicate comparison from 20,000 to 120,000 instructions in the basic A*Cache and A*Cache_Opt schemes, with two different settings, 5% and 10%, for the cache size. The measured CacheHitRatio in the first case was constant at about 50%, and in the second case, at about 60%. From the graph for A*Cache, we see that as the predicate comparison cost increases beyond 70,000 instructions, both configurations of A*Cache perform worse than the No-Cache scheme. This deterioration in performance is due to the larger cost of cache containment reasoning, which after about 70,000 instructions, outweighs the cost of remote trips to the server. The performance can be improved with suitable predicate indexing and merging techniques (Chapter 4), which are not used in our simulation.

Another observation that can be made from the results in Figure 7.5 is that the larger cache size of 10% performs worse than the smaller cache for a predicate comparison cost of about 60,000 instructions and above. This result is expected, since a larger cache causes more predicates to be cached and compared during cache containment reasoning, thus increasing the cost of local processing.

The corresponding results for A*Cache_Opt also appear in Figure 7.5. The response time for A*Cache_Opt is better than that of A*Cache, and the cross-over with No-Cache therefore occurs at higher values of the predicate comparison cost.

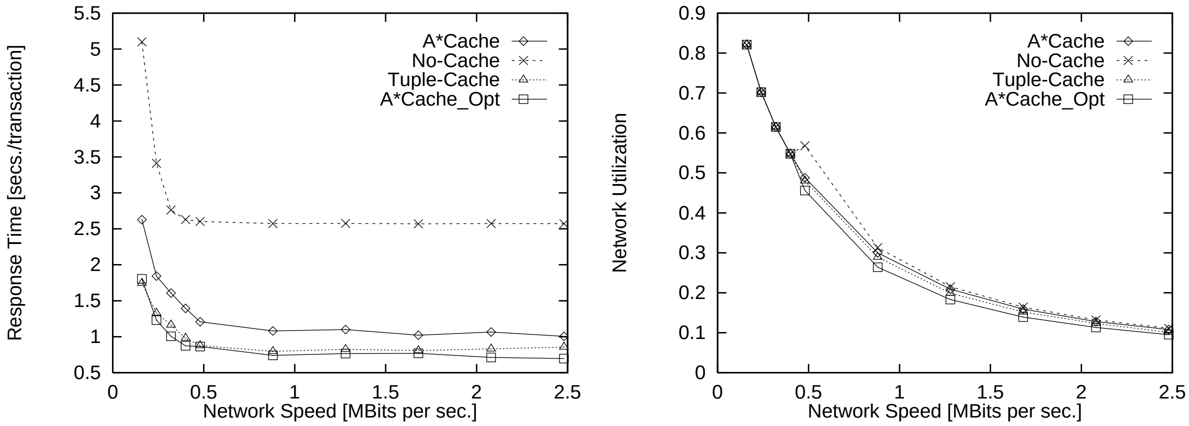


Figure 7.6: *Varying Network Speed, 5% CacheSize, Private Unclustered Access*

7.3.4 Client CPU

A number of additional experiments were performed to detect the sensitivity of the system response to other parameters, such as the ClientCPU, HotRatio, and RecordSize. Decreasing the MIPS of the client CPU was found to have little impact on the system response, unless it fell very significantly (below 5 MIPS). Client CPU utilization was low in all cases, showing that the client was not overloaded with local caching tasks. The performance effect of varying server CPU was much more pronounced, which is expected since the server is a shared resource.

7.4 Effect of Network Speed

Figure 7.6 shows the effect of varying the network speed on the ResponseTime and on NetworkUtil outputs for a 5% cache size. The NetworkSpeed parameter is varied from 0.1 MBits/sec to 2.5 MBits/sec. Because all of the caching schemes have less network traffic than No-Cache, they perform better for increasing values of network speed. The performance gains saturate at a network speed of about 0.5MBits/sec for the environment chosen for this set of experiments. As discussed in Section 7.1, A*Cache transfers more data from the server than the other two caching schemes, and its response time is accordingly higher.

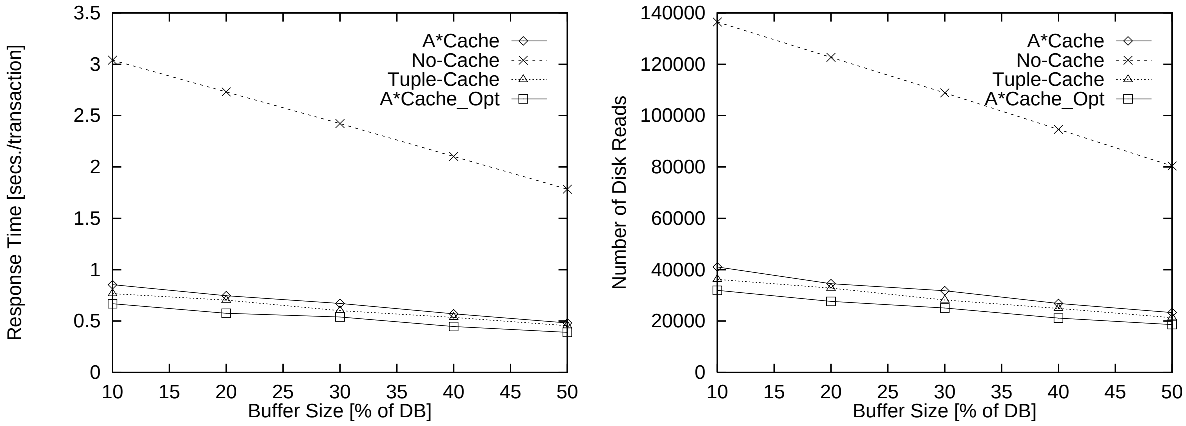


Figure 7.7: *Varying BufferSize, Private Unclustered Access*

7.5 Effect of Server Parameters

Server resources are shared among many clients, and therefore affect the system performance to a large extent. In this section, we examine the effect of varying the parameters of the resources at the server site, such as the buffer size, and disk speed, and the CPU MIPS.

7.5.1 Buffer Size

Figure 7.7 shows the effect of varying the size of the server buffer for unclustered data access, i.e., for queries that use the attribute $Unique2$ to access data. The cache size is set to 200 Kbytes, which is 10% of the database, and the buffer size is varied from 10% to 50% of the database. Rest of the input parameters have their default settings. The response time of all the schemes improves with increasing buffer size. However, as is evident from the figure, all three client-side caching schemes are much less sensitive to the server buffer size compared to No-Cache. This behavior is desirable, since it reduces the dependency of the system performance on shared resources at the central server. Similar benefits are observed in the clustered case, although the absolute values of the query response time are much smaller for clustered access.

7.5.2 Disk Speed

Figure 7.8 shows the effect on system performance of varying the speed of the disk at the server from 2 milliseconds to 16 milliseconds for unclustered data access. In this experiment,

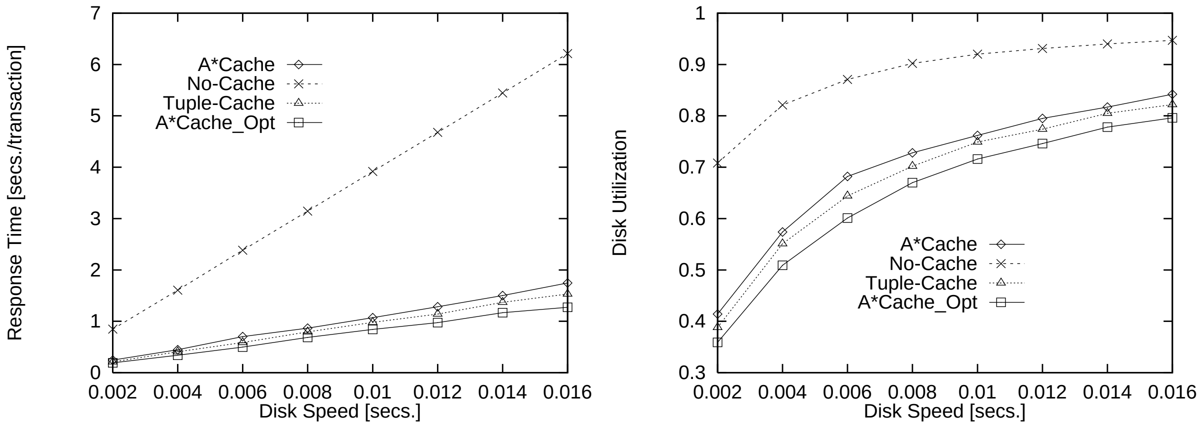


Figure 7.8: *Varying Disk Speed, Private Unclustered Access*

the cache size is set to 200 Kbytes, which is 10% of the database, and all other parameters have their default settings. Response times for all the schemes increases as the disk gets slower. However, all the schemes with client-side caching are less sensitive to the disk speed than No-Cache, since the server disk is not accessed whenever data is read locally in the cache. Similar results are obtained for the clustered case. These results are analogous to those obtained for the server buffer, essentially reinforcing the conclusion that client-side caching schemes are less sensitive to the availability of central resources.

7.5.3 Server CPU

Figure 7.9 shows the effect of varying the speed of the server CPU on the query response time and server CPU utilization. In this set of experiments, the server CPU speed is varied from 5 MIPS to 75 MIPS, with all other input parameters having their default settings.

From the graphs, we see that No-Cache has the slowest response time, and is also the most sensitive among all the schemes to smaller values of the server MIPS. Tuple-Cache performs better than No-Cache, and about the same as A*Cache for CPU speeds 25 MIPS and higher. For lower values of the CPU speed, A*Cache_Opt and Tuple-Cache are comparable. A*Cache_Opt performs the best for values of server speed greater than 25 MIPS, since it utilizes its cache most effectively, making fewer remote requests compared to both A*Cache and Tuple-Cache.

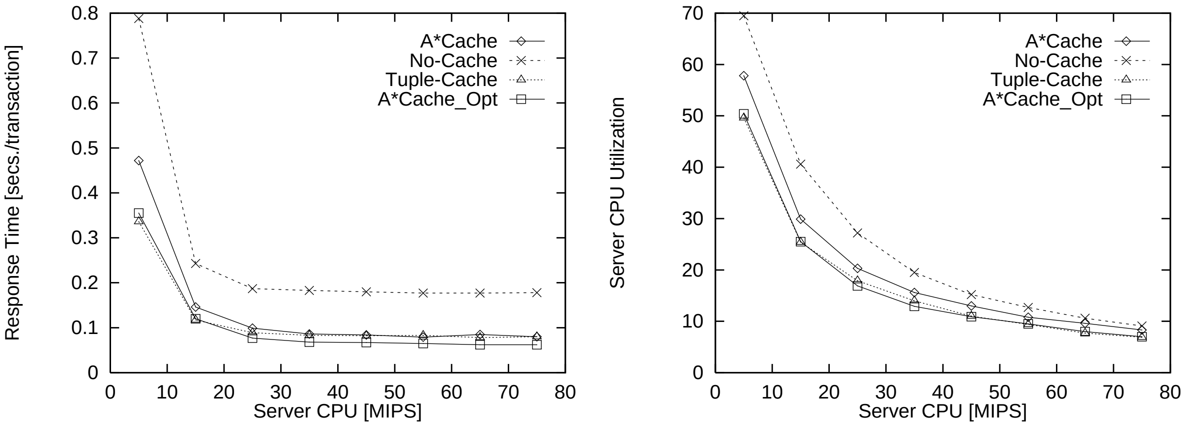


Figure 7.9: *Varying Server CPU, 5% CacheSize, Private Clustered Access*

7.6 Scalability Experiments

For a database system to perform well, it must not only provide good response to varying system loads, but it must also scale with respect to the additional clients and queries. In our simulation, we examined the scalability of the four types of client-server systems for increasing values of query length and number of clients. These results are presented below.

7.6.1 Query Length

Figure 7.10 shows the behavior of the systems when the query length is varied from 0.1% to 1.3% of the database, with CacheSize being at its default setting of 5% of the database size. All other input parameters are also set to their default values. Data is accessed using the clustered attribute $Unique2$, the number of tuples in a result set increasing with larger query lengths.

It can be observed from the graphs in Figure 7.10 that the output parameter CacheHitRatio for both A*Cache schemes rises with increasing query lengths from about 30% at 0.1% query length, to 60% at 1% query length. Larger queries have larger predicate intervals that are more likely to overlap with each other, resulting in a greater number of cache hits. This behavior causes a slower growth in the query response time for increasing query lengths in the two A*Cache schemes, as seen in the figure. In contrast, the response time for the No-Cache scheme increases rapidly with larger query length, as does the number of disk reads it performs. Tuple-Cache has approximately the same performance characteristics as

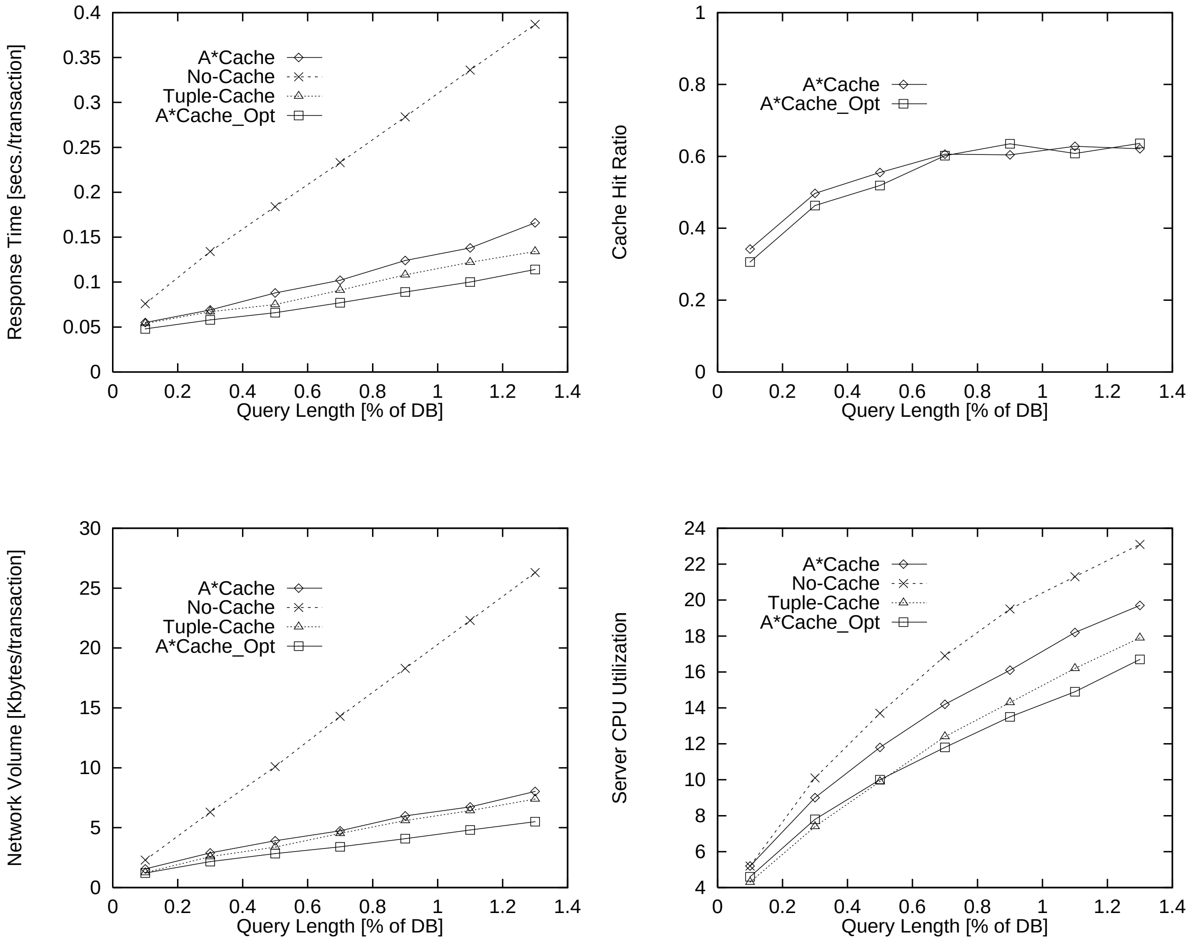


Figure 7.10: *Varying Query Length, Private Clustered Access*

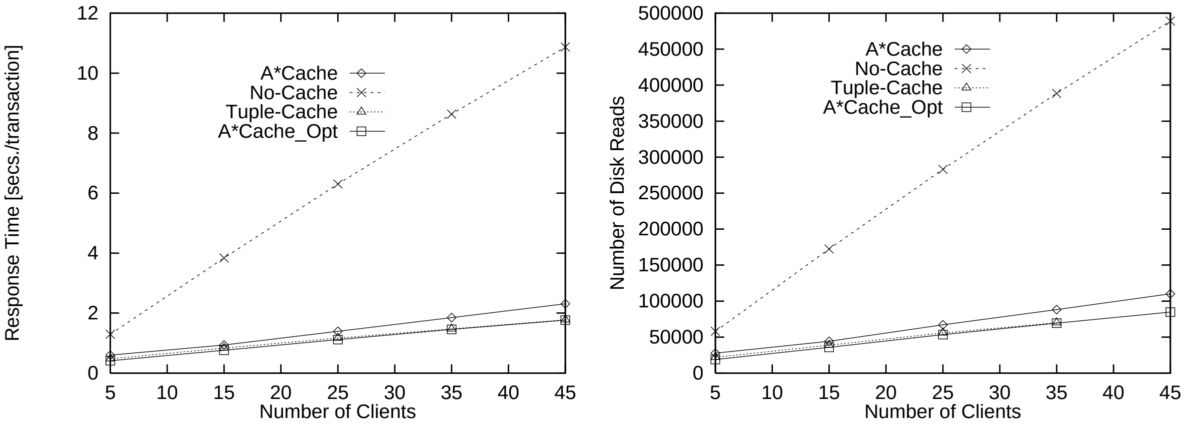


Figure 7.11: *Varying Number of Clients, Private Unclustered Access*

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.

7.6.2 Number of Clients

The demands on the shared resources increase directly with more clients, and therefore, the effect on system performance of increasing clients is an important scalability issue. In the following set of experiments, we vary the number of clients in the system from 5 to 45 with 10% CacheSize, keeping all other parameters fixed at their default values. Figure 7.11 presents the results of these experiments, showing the effects on response time and server CPU utilization.

As can be seen from these graphs, the query response times for all four systems increases with larger number of clients. However, the systems with client-side caching scale much better with increasing load. 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. This result is analogous to that obtained for other shared resources such the server buffer and the disk; the performance characteristics in all three sets of experiments establish the better scalability of the client-side caching schemes, as well as their reduced sensitivity to critical central resources, compared to systems with no

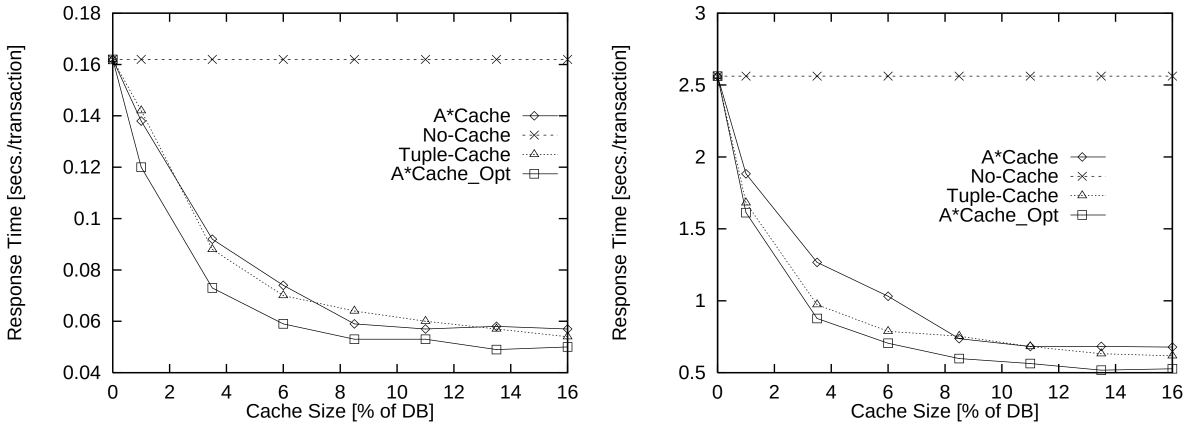


Figure 7.12: *Varying CacheSize, HotCold Access, Clustered and Unclustered*

caching.

7.7 Experiments with the HotCold Workload

In this section, we describe the some of the results of our experiments with the HotCold workload type. Unlike the Private access model, hot regions are not exclusive to clients in the HotCold scenario, and are therefore subject to shared access. In general, the HotCold read-only results are quite similar to the Private model, since the only difference between the two is in the nature of cold access. Below we report the results of two sets of experiments, one varying the cache size and the other varying the hot access probability, in the HotCold access model.

7.7.1 Varying Cache Size

Figure 7.12 shows the response time changes with varying cache size, for both clustered and unclustered access. All other input parameters are set to their default values.

Comparing these results against the Private results from Figures 7.1 and 7.2, we find that all the four schemes behave very much the same as in Private. No-Cache has the worst performance, with the benefits of caching more prominent for unclustered access. As in the Private scenario, A*Cache_Opt performs the best in both clustered and unclustered access, reusing its cached data most effectively.

Notice that for smaller cache sizes, 6% or less, A*Cache performance is comparable

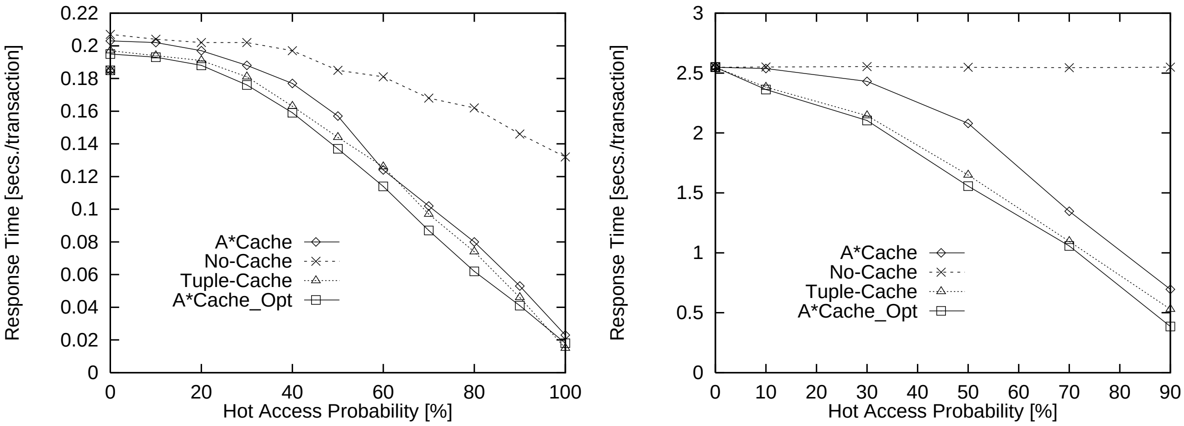


Figure 7.13: *Varying HotAccess, 5% CacheSize, HotCold Clustered and Unclustered*

to that of Tuple-Cache for clustered access, while it is worse than Tuple-Cache for the same range of cache sizes in the unclustered case. This behavior is also manifested for the Private workload, and can be explained as follows. In the clustered case, there are fewer disk accesses for a query, since tuples in a result set are grouped together in a small number of pages. For unclustered data, there are many more disk accesses per query compared to clustered. Recall from our discussion of query processing strategies in Section 7.1 that A*Cache does not take into account partial cache hits, thus fetching all tuples in the result set. This feature of A*Cache causes many more disk reads for small cache sizes in the unclustered case, slowing down its performance with respect to Tuple-Cache. On the other hand, the difference in the number of disk reads is much smaller in the clustered case, making A*Cache performance about the same as Tuple-Cache for cache sizes 6% or lower.

7.7.2 Varying HotAccess

In this set of experiments, the hot access probability is varied from 0% to 100%, with the cache size being at its default setting of 5%. All other input parameters have their default values. The results are shown in Figure 7.13, for both clustered and unclustered access, the behavior being similar to that obtained for Private access.

Notice that the 'peak' effect observed in the Private clustered case (Figure 7.3) is missing in the HotCold clustered results. The reason for the difference in HotCold behavior is precisely that the hot region of a client in the HotCold model is shared as part of the cold region of other clients. Recall that the peak in the Private clustered scenario is caused

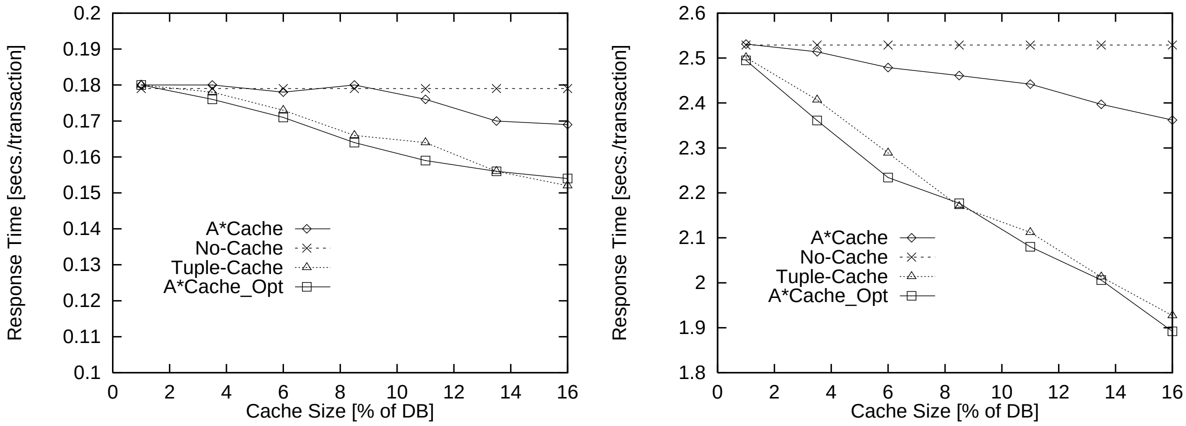


Figure 7.14: *Varying CacheSize, HiCon Clustered and Unclustered Access*

by reduced sharing in the server buffer as client-private hot regions are accessed more frequently. For HotCold however, increasing HotAccess from low values does not cause the server buffer reuse to deteriorate as much as in the Private case, thus providing a steady improvement in the response time.

7.8 Experiments with the HiCon Workload

This section presents the results of some of our experiments with the HiCon workload. Recall that this workload has a common shared hot region, and is primarily intended to model shared high contention scenarios for data writes. We include some read-only results for this workload below, in order to provide a reference for the subsequent read-write experiments.

7.8.1 Varying CacheSize

Figure 7.14 shows the effect of varying the cache size for both clustered and unclustered data access in the HiCon workload model. In this set of experiments, all other input parameters have their usual default values, as noted in the Tables 6.1, 6.2, 6.3, and 6.4. For this environment, the HotRatio is 50%, splitting the database equally between hot and cold regions, with 80% of accesses being directed by all clients to the common hot region. A*Cache_Opt and Tuple-Cache schemes perform the best in this scenario, causing far fewer disk reads than the No-Cache and also the basic A*Cache schemes.

Observe that the benefits of A*Cache_Opt and Tuple-Cache as compared to basic

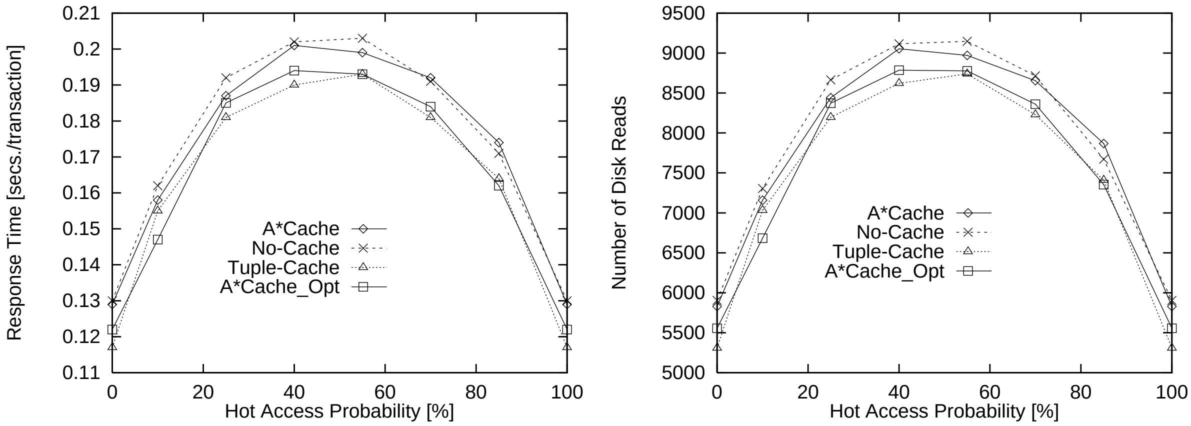


Figure 7.15: *Varying HotAccess, HiCon Clustered Access*

A*Cache are more significant in the HiCon unclustered case than in clustered. In the environment of these HiCon experiments, the common hot region is 50% of the database, which is large compared to the size of the client cache (5%). Therefore, the cache miss rate is higher than for the Private or HotCold experiments with 5% hot regions. Additionally, the cost of a cache miss in the unclustered case is higher for A*Cache, since it fetches all tuples in a result set, causing more disk reads than A*Cache_Opt and Tuple-cache. Due to these reasons, a degradation in A*Cache performance occurs for the unclustered environment, and even for the clustered one, in which A*Cache performs about the same as No-Cache for cache sizes about 10% and smaller.

7.8.2 Varying HotAccess

In this set of experiments for the HiCon model, the hot access probability is varied from 0% to 100%, with the cache size being at its default setting of 5%. All other input parameters have their default values. The results for clustered and unclustered cases are shown in Figures 7.15 and 7.16 respectively.

The response time and disk reads in Figures 7.15 and 7.16 show interesting peak effects, the maximum occurring at 50% value of the HotAccess probability. This behavior is a result of the environment of these HiCon experiments, in which the HotRatio is 50%, splitting the database equally between hot and cold regions. For low values of HotAccess, the server buffer mostly holds pages from the common cold region. As access to the cold region is decreased, pages are accesses more uniformly from the entire database, causing a decrease

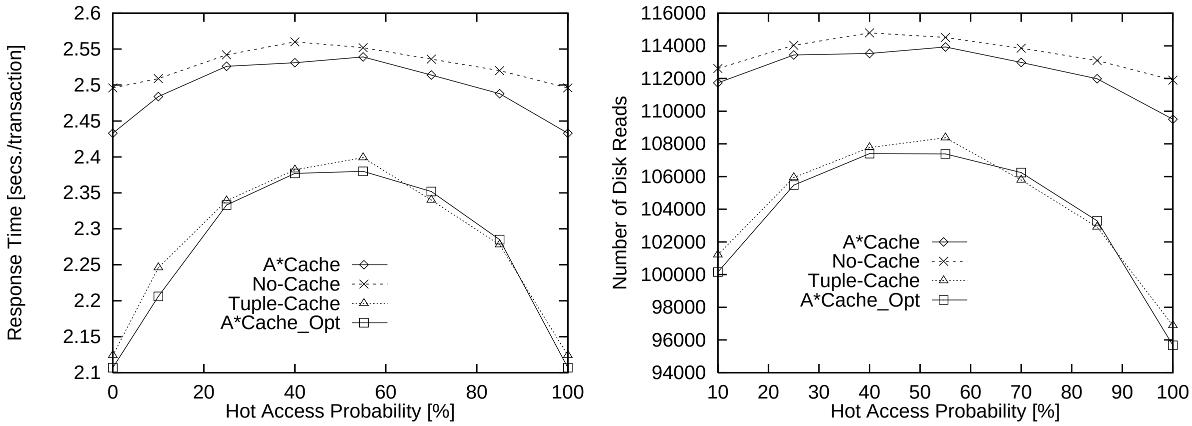


Figure 7.16: *Varying HotAccess, HiCon Unclustered Access*

in the effectiveness of the server buffer. For a HotAccess value of 50%, half of the queries use the cold region and the other half use the hot, so that the entire database is accessed equally and the hit ratio of the server buffer (25% of the database) is the lowest. As access switches over to the hot region, reuse of pages in the buffer rises again, reaching the same values with 100% hot access as with 0%.

Another interesting observation from Figures 7.15 and 7.16 is that the benefits of A*Cache_Opt and Tuple-Cache over basic A*Cache and No-Cache are much larger in the unclustered case than in the clustered. Again, the reason for this behavior is that the cost of a cache miss is much higher for basic A*Cache than Tuple-Cache or A*Cache_Opt, since it fetches entire query results and not just the missing tuples, and this cost is much larger for unclustered disk access than clustered, due to poor utilization of the server buffer for unclustered use.

7.9 Experiments with the Uniform Workload

The Uniform workload portrays uniformly random access to the entire region of the database. Although this workload is rare in practical situations, we examine some results here in order to compare its behavior against the other workload types.

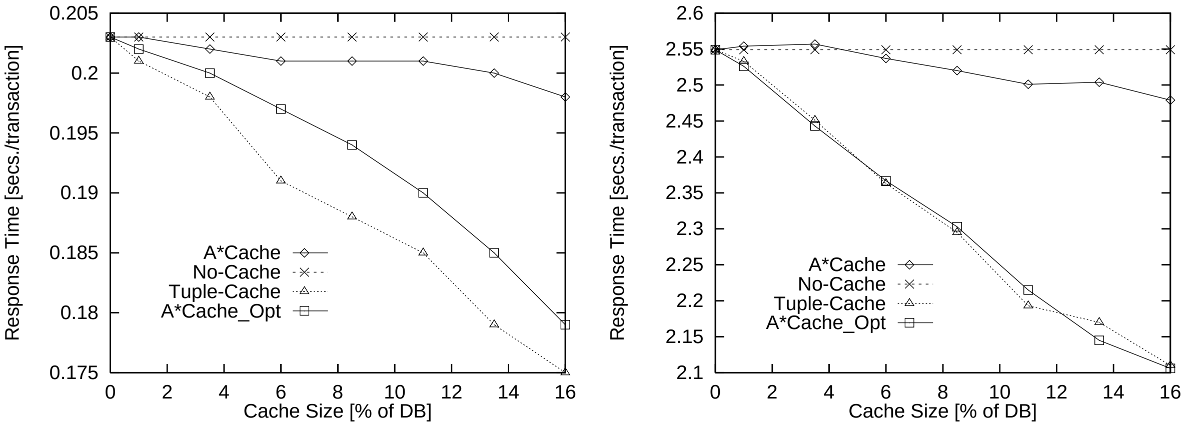


Figure 7.17: *Varying CacheSize, Uniform Clustered and Unclustered*

7.9.1 Varying CacheSize

In this set of experiments, the cache size is varied for both clustered and unclustered uniform access to the entire database. As in the results presented above for the other workloads, basic A*Cache is much less effective than Tuple-Cache or A*Cache_Opt, due to its higher cost of cache miss. This difference is more significant for the unclustered case, since the server buffer utilization is poor for queries using an unclustered attribute. It is also seen that for the clustered case, Tuple-Cache performs the best, even better than A*Cache_Opt. This result can be attributed to the fact that for uniform access spread over the database, predicate-level hits to the cache are rare, thereby decreasing the cache hit ratio for the A*Cache schemes. The overhead of examining and maintaining the predicate-based cache description becomes significant, causing Tuple-Cache to perform better in this scenario.

7.9.2 Varying BufferSize

The effect of the server buffer size is quite pronounced for the Uniform scheme, as shown in the Figure 7.18 below. This is because the page access pattern for the database is random, and the buffer size has a direct relationship with the number of disk reads. Note that all the caching schemes are less sensitive to buffer size than No-Cache, with A*Cache_Opt providing the best response time in the environment of these experiments.

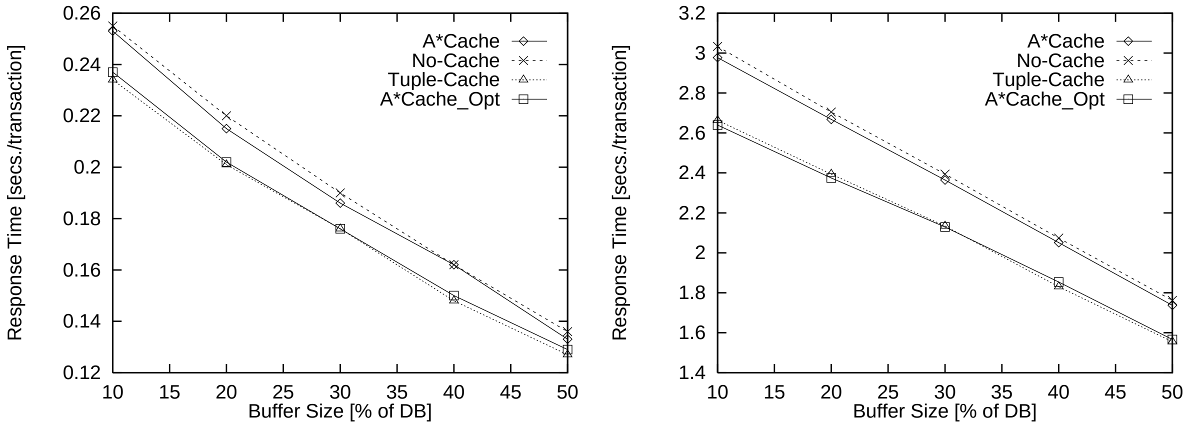


Figure 7.18: *Varying BufferSize, Uniform Clustered and Unclustered*

7.10 Summary of Read-Only Experiments

In this chapter, we have examined the behavior of the four simulated systems, No-Cache, Tuple-Cache, A*Cache, and A*Cache_Opt, for read-only transactions in the Private, Hot-Cold, HiCon, and Uniform access models. The read-only scenario inherently favors the caching schemes over the No-Cache architecture, since there are no maintenance costs incurred at the shared server and on the network. Our experimental results verified that A*Cache_Opt performs the best for most read-only environments, followed by Tuple-Cache, and then basic A*Cache. Benefits for all the three caching schemes are substantial when shared resources are at premium, or when data access on the disk is unclustered, since the local client cache is most effective in these cases. However, when the predicate comparison cost is increased, the cost of cache containment reasoning also rises, and it can exceed the savings from local containment checking and query execution in the A*Cache schemes. For our experimental environment with a cache size 5% of the database, the cross-over of A*Cache performance with No-Cache happened for a cost of about 70,000 instructions per predicate comparison. Such values of the predicate comparison cost are extreme. Furthermore, this limit can be reduced by suitable indexing and merging of the query predicates in a cache description; such optimizations were not used in our simulation.

Chapter 8

Performance Analysis of Read-Write Workloads

In this chapter, we consider client transactions that both read and update data. First, we briefly review the important differences in update-handling procedures of our four caching schemes. We then specify our performance metrics, and present our experimental results, focusing primarily on the update characteristics. We explore and compare the behavior of the four caching schemes, considering particularly the costs of cache maintenance and its effect on response times. We conclude this chapter with a summary of our results for read-write workloads.

8.1 Comparison of Data Update Strategies

Each of the four caching schemes, No-cache, Tuple-Cache, A*Cache, and A*Cache_Opt, handles data writes quite differently. Below, we provide a summary of our discussion on this issue in earlier chapters.

With respect to the steps taken by a client on a request for data update, the following differences exist among the four systems:

- No-Cache always performs its updates at the server, just as it does its queries.
- 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 remote trip made by the transaction. For our simulation, a transaction consists of a single query or update, and hence updates are sent to the server along with the commit request.

- As described in Section 6.3.2, 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 on commit. 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.

- The extended A*Cache_Opt scheme defined in Section 6.3.3 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. On the other hand, 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 satisfied the access predicate but are missing from the client cache, are returned as the result.

  A characteristic of the associative queries used in our simulation environment is that the updated attributes do not occur in the predicates used to access data (Section 6.6.2). Thus, the remote updates in A*Cache_Opt do not cause predicates previously cached by the client to be changed. Only the new access predicate is inserted in the client and server cache descriptions as the client application continues its data processing, and finally submits a commit request if not aborted by notification of conflicts.

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.

- No-Cache has no client-side caching, and consequently, no generation or processing of update notifications.

- Both A*Cache schemes, the basic 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. Note that this filtering is at the level of *predicates*, and 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. For the updates in our simulation environment, attributes that are modified do not affect the predicates used to access the data. Hence, only those tuples that are affected by the notification are updated in the cache; previously-cached predicates do not need to be altered.

- 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 8.1 below:

Update Processing Characteristic	**No-Cache**	**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	$\surd$			$\surd$
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 8.1: *Comparison of Update Handling Methods*

8.2 Read-Write Performance Metrics

For the read-write scenario, update-related simulation parameters are introduced in addition to those in the read-only environment. As described in Section 6.6, these parameters — namely, *HotTransWrite*, *ColdTransWrite*, *HotTupleUpdate*, and *ColdTupleUpdate* — deal with write probabilities of transactions and tuples. These input parameters have the default settings noted in Table 6.4 in Section 6.6. In our read-write experiments, the values of these parameters are varied around their default settings and the effects are observed.

Table 8.2 lists the output parameters that are monitored in our simulation experiments for read-write workloads. As in the case of read-only experiments, our primary performance metric is the response time. Additional simulation output related to data writes, such as the number and network cost of notifications, number of disk writes, etc., are measured for the read-write transactions. These parameters are summarized in Table 8.2, which also includes the output parameters described earlier in Section 7.2 for our experiments in the read-only environment.

Output Parameter	**Description**	**Unit**
ResponseTime	Transaction start time — end time	Secs. per transaction
DiskReads	Number of disk reads	Number
DiskWrites	Number of disk writes	Number
DiskUtil	Utilization (busy time) of the disk	% of elapsed time
CacheHitRatio	Ratio of queries evaluated in cache	% of total queries
BufferHitRatio	Ratio of tuples found in server buffer	% of tuples accessed
NumNotify	Number of notifications	Number
NumAborts	Number of transaction aborts	Number
NetworkVolume	Data and message traffic on network	Kbytes per transaction
NetworkNotify	Notification traffic on network	Kbytes per transaction
NetworkUtil	Utilization (busy time) of the network	% of elapsed time
ServerUtil	Utilization (busy time) of server CPU	% of elapsed time
ClientUtil	Utilization (busy time) of client CPU	% of elapsed time

Table 8.2: *Output Parameters for Read-Write Workloads*

8.3 Experiments with the Private Workload

In this section, we report the results of experiments done by varying client and workload parameters in the Private access model, such as the size of the client cache, clustered and unclustered selection of data, hot data access probability, and the cost of predicate comparison. Transactions both read and write data, triggering cache maintenance activity at the clients and the server.

8.3.1 Varying Cache Size

In this set of experiments, the size of the client cache is varied from 0% through 20% of the database size. The read-only results of Chapter 7 demonstrate that performance does not improve much for caches larger than 10%, and hence we choose to examine 20% or smaller cache sizes. The rest of the input parameters have their default settings shown in the Tables 6.1, 6.2, 6.3, and 6.4. For example, the frequency *HotAccess* with which the client-specific hot region is accessed is 80%, and the size of the server buffer is fixed at 25% of the database. In this experimental environment, the size of the private hot region of each of the 10 clients is equal to 5% of the database size, and the size of the shared cold portion of the database is 50%.

Varying Cache Size for Updates via Clustered Index

Figure 8.1 shows the response time, network volume, number of notifications, and server CPU utilization with varying cache size. The transaction write probability is set to its default value of 20% for access in both hot and cold regions, the tuple update probability of a read-write transaction being 50%. From the figure, it is seen that for cache sizes larger than 5%, A*Cache_Opt performs better than all the other schemes in terms of the response time, network volume, number of notifications, and server CPU utilization. As the cache size is increased, more cold data is stored at a client, which is only accessed 20% of the time for this experimental environment. This fact, coupled with the increased cost of update notification for cold 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 8.1 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 6.3.2, 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 but at the same time, locking the tuples at the server. Acquiring 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 this difference 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.

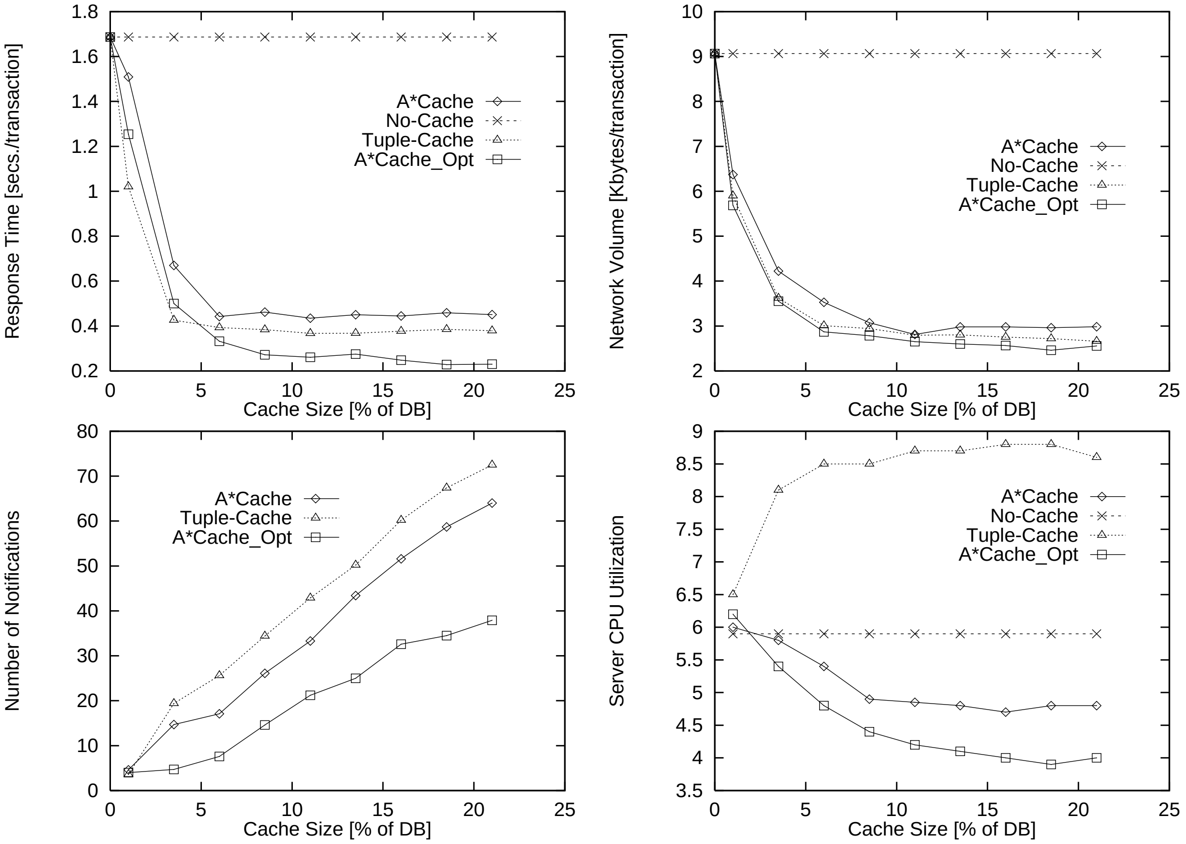


Figure 8.1: *Varying Cache Size, 20% Update Transactions, Private Clustered Access*

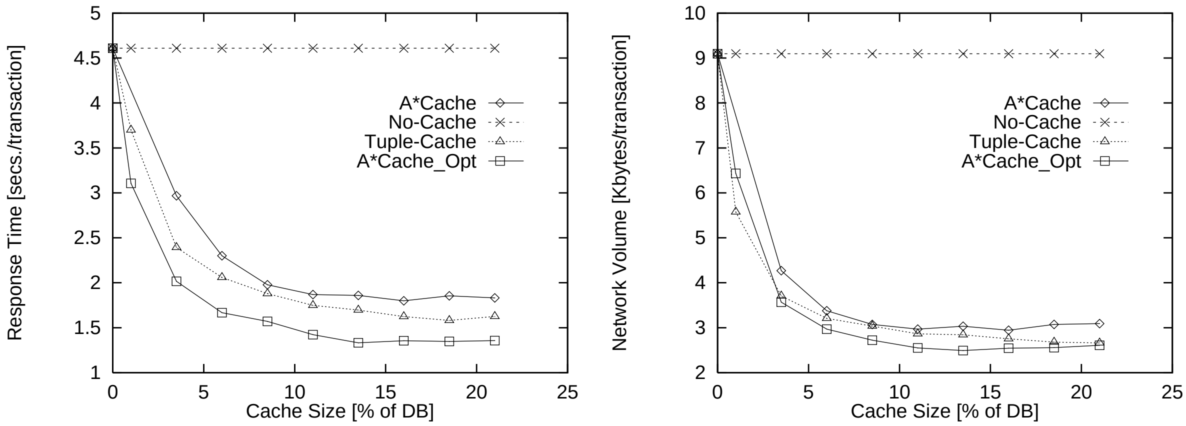


Figure 8.2: *Varying Cache Size, 20% Update Transactions, Private Unclustered Access*

Varying Cache Size for Updates via Unclustered Index

Figure 8.2 shows the effects of data updates made using the unclustered attribute *Unique2* for a 20% ratio of update transactions. Again, all the three caching schemes perform better than No-Cache, with A*Cache_Opt being the best, followed by Tuple-Cache and A*Cache. As in the read-only case, the response times are an order of magnitude larger for the unclustered updates compared to clustered, since many more disk accesses are made for an unclustered query, reducing the effectiveness of the server buffer. The performance improvement saturates after a cache size of about 15% is reached, for the same reasons as noted in the clustered case above.

8.3.2 Varying Write Probability

In the Private model, the hot region of a client is not shared, and update conflicts happen only on the shared cold portion of the database. Thus, varying the amount of hot writes does not change the number of notifications generated for other clients. In this section, we report the results of experiments that vary the *ColdTransWrite* probability for Private workloads. This parameter specifies the percentage of transactions that update data in the cold region, and therefore determines the contention encountered by different clients on the shared cold region of the Private access model.

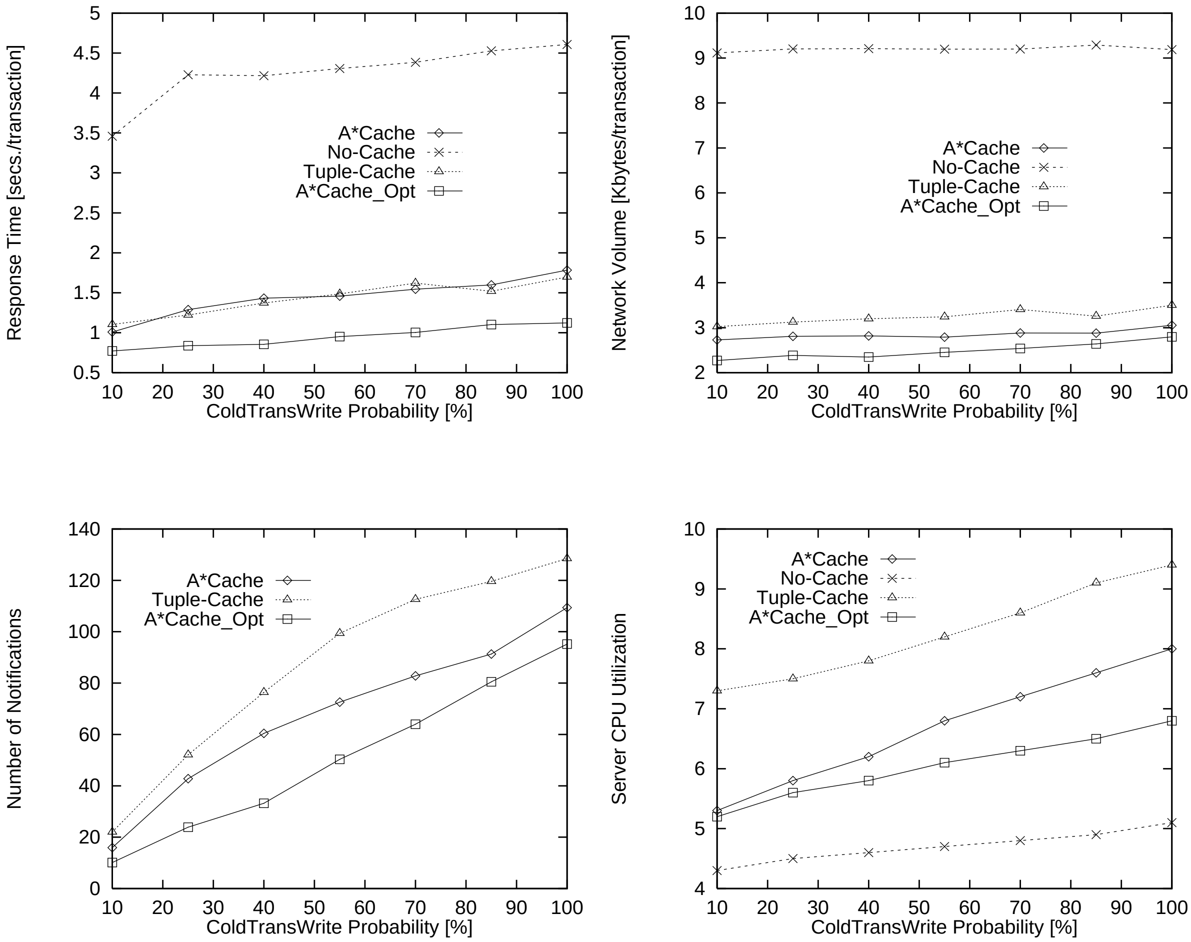


Figure 8.3: *Varying Cold Write Probability, Private Unclustered Access*

Varying ColdTransWrite

Figure 8.3 shows the effects of varying the *ColdTransWrite* probability. The value of the *HotTransWrite* parameter is set to 0, indicating that hot data is only read and not updated, while all other input parameters have their usual default settings from Tables 6.1, 6.2, 6.3, and 6.4.

As the ratio of update transactions in the cold region is increased, the response times of all the four schemes increase, and also the number of notifications and server CPU utilization. A*Cache_Opt has the best performance, while Tuple-Cache and A*Cache are comparable. Due to its update policy of locking and fetching missing tuples only, Tuple-Cache has the highest number of notifications, which balances its savings in terms of reduced

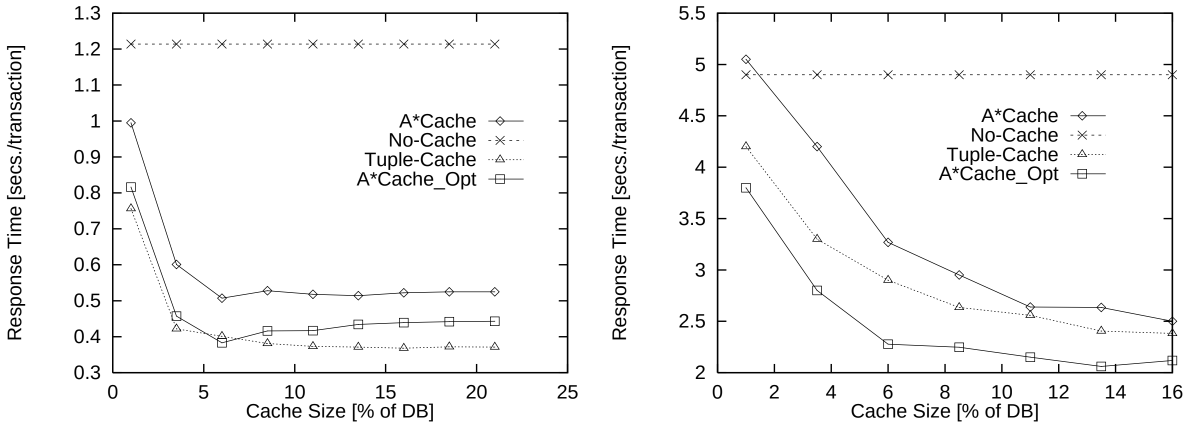


Figure 8.4: *Varying CacheSize, 40% Update Transactions, Private Clustered and Unclustered*

data volume on the network on a remote fetch. A*Cache fetches all tuples in a result set, but has fewer notifications precisely because the tuples are locked at the server before the fetch. Notice that the server CPU utilization for all the caching schemes is greater than that of No-Cache. There are two reasons for this behavior: (1) because unclustered access makes a job more disk-bound than CPU-bound for No-Cache, and (2) because the server processor generates notifications for updates in A*Cache, A*Cache_Opt, and Tuple-Cache.

8.3.3 Varying CacheSize with 40% Update Transactions

In this set of experiments, 40% of all transactions perform updates, irrespective of whether they access the cold region or the hot region. That is, the values of both the *ColdTransWrite* and *HotTransWrite* parameters are set to 40%, the per-tuple update probability of a read-write transaction being at its usual setting of 50%.

For the clustered case, it can be observed that Tuple-Cache actually performs the best for cache sizes larger than about 7%, 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 is relatively more expensive for the clustered case.

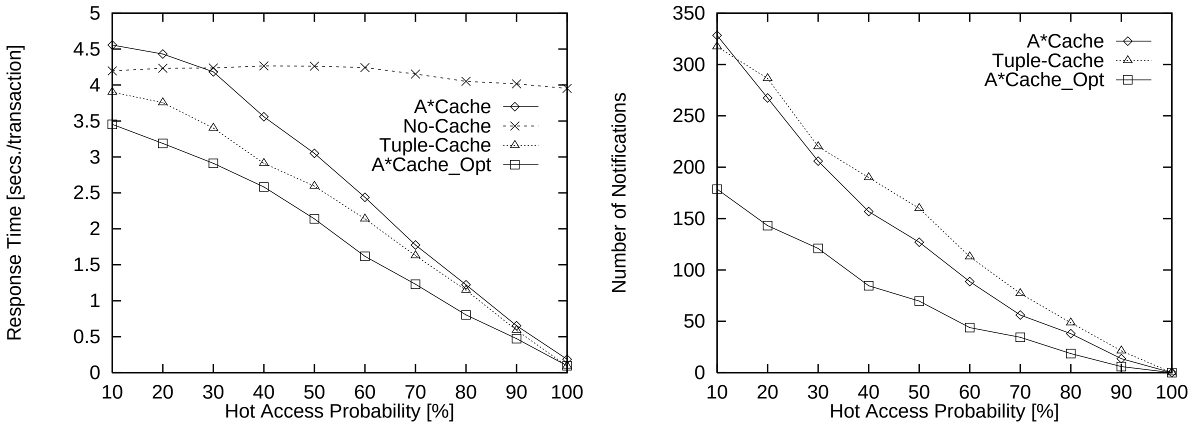


Figure 8.5: *Varying HotAccess, Private Unclustered Access*

8.3.4 Varying Hot Access Probability

This set of experiments varies the probability HotAccess of using data in the hot region of the client, with a *HotTransWrite* value of 0 and a *ColdTransWrite* of 20%. Thus, no transactions update hot data, but 20% of transactions in the shared cold region perform updates. The cache size is set to 10% of the database. Figure 8.5 shows the response time and number of notifications generated for this experimental environment.

Response time of all the caching schemes is seen to decrease nearly linearly with increasing hot access, while the improvement for No-Cache is minimal. The Private workload has a non-shared hot region for each client which is 5% of the database size, and it fits entirely into the client cache (10% of the database in these experiments). Thus, for the caching schemes, the shared server buffer is accessed only for the shared cold region. In contrast, the non-shared hot data in the No-Cache scheme also resides in the server buffer, thereby 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 hot access.

8.3.5 Effect of Other Parameters

Many other experiments were performed to examine the effects of varying the network speed, server buffer size, speed of the central disk, and server CPU. Analogous to the read-only environment, performance of all the caching schemes is less sensitive to availability of shared resources than the No-Cache configuration. Scalability experiments varying the query length and the number of clients were also performed. 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.

8.4 Experiments with the HotCold Workload

We now examine the behavior of transactions in the HotCold access model. Results are expected to be similar to that of Private, since the two access models are closely related. A difference between the two models is that for HotCold, each client's hot region is part of the cold region of other clients, so that unlike the Private access model, update contention exists also on the hot regions.

8.4.1 Varying CacheSize

Figure 8.6 show the effects of varying the cache size in the HotCold access model for unclustered data access. The experiment uses 20% values for *HotTransWrite* and *ColdTransWrite* parameters, so that 20% of the transactions in both the hot and cold regions perform data updates.

The results are similar to that obtained for Private workloads, although the number of notifications is larger in HotCold. All the caching schemes perform better than the No-Cache, and A*Cache_Opt provides the best response in both the 20% and 40% update scenarios with unclustered access.

8.4.2 Varying HotTransWrite

Figure 8.7 shows the effects of varying the probability of transactions that write data in the hot region, with unclustered access of data. It is seen that the response time increases for

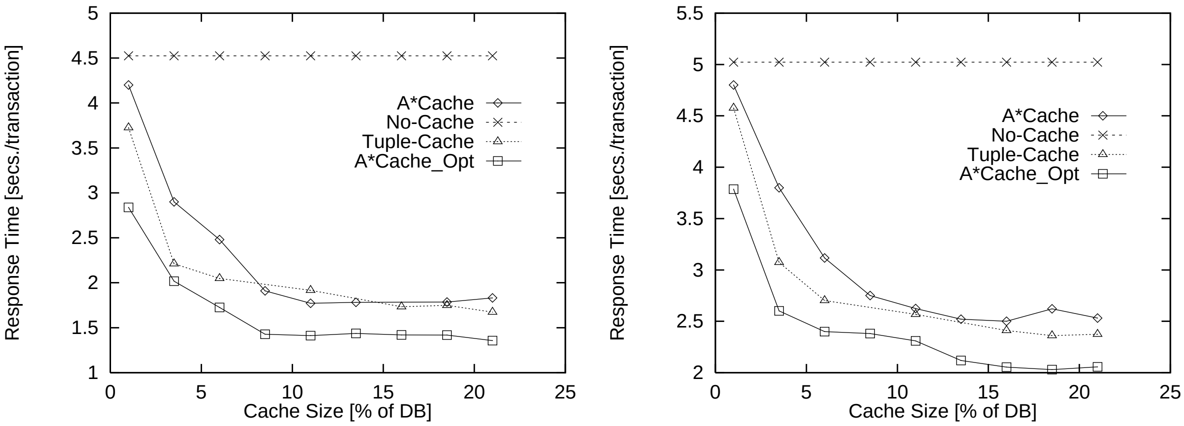


Figure 8.6: *Varying CacheSize, 20% and 40% Update Transactions, HotCold Unclustered Access*

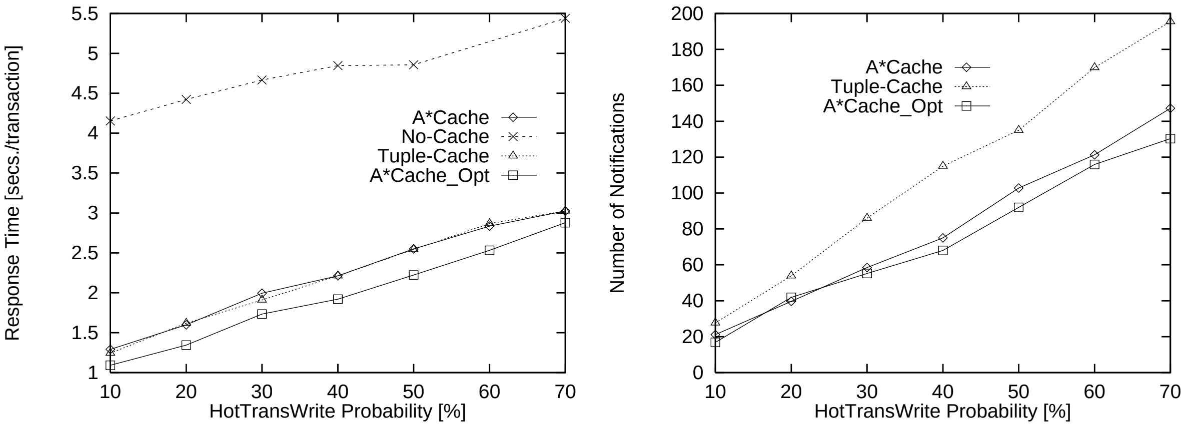


Figure 8.7: *Varying HotTransWrite, HotCold Unclustered Access*

all the schemes, although the caching schemes do much better than No-Cache. As expected, the number of notifications of the three schemes is also seen to increase for higher data write probabilities.

8.5 Experiments with the HiCon Workload

This section presents the results of some of our experiments with the HiCon workload. This workload has a common shared hot region, and represents high contention for data writes. For our experimental environment, the HotRatio is 50%, splitting the database equally between hot and cold regions.

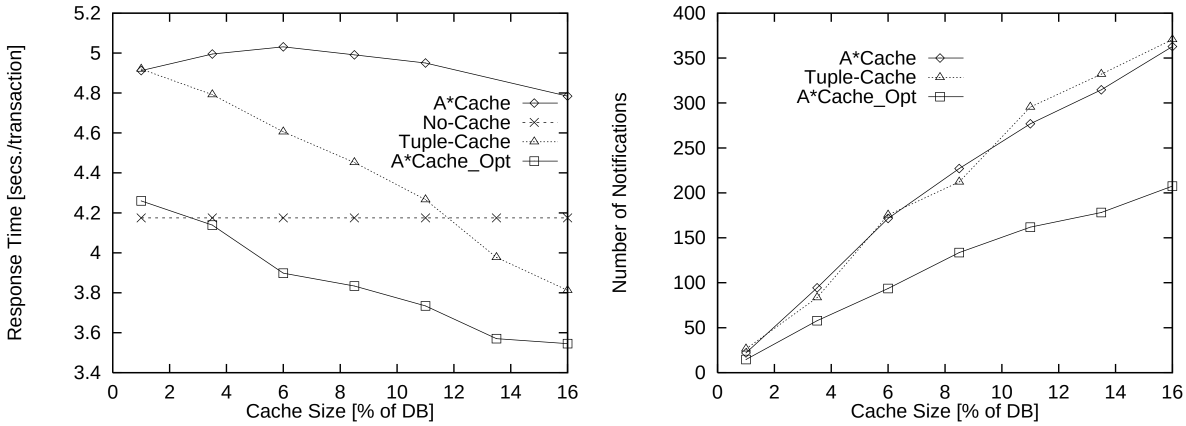


Figure 8.8: *Varying CacheSize, 20% HotTransWrite, HiCon Unclustered Access*

8.5.1 Varying CacheSize

Figure 8.8 presents the results of varying the cache size in the HiCon workload model with unclustered access. The experiment uses 20% values for *HotTransWrite* and *ColdTransWrite*, so that 20% of the transactions in both the hot and cold regions perform data updates. The probability *HotAccess* of accessing the hot region is set to its default value of 80%, and all other input parameters have their usual default settings.

It can be observed that A*Cache actually performs the worst in the above high-contention scenario for cache sizes up to 15% of the database size. Tuple-Cache does better with increasing cache sizes, surpassing the No-Cache scheme at around 12% cache size. A*Cache_Opt provides the best response time, and also the least number of notifications. This result can be attributed to the fact that A*Cache_Opt performs updates at the server on cache misses, while Tuple-Cache and A*Cache fetch the required tuples and update them locally. The cache hit ratio is low in this experimental setting, and hence the remote update strategy of A*Cache_Opt proves to be beneficial for this high-contention environment, acquiring central locks and reducing data conflicts.

8.5.2 Varying Hot Access

Figure 8.9 shows the effect of varying the hot access probability from 10% to 100%, with the cache size being at its default setting of 5%. All other input parameters have their default values, with 20% update transactions in the hot and cold regions of the database.

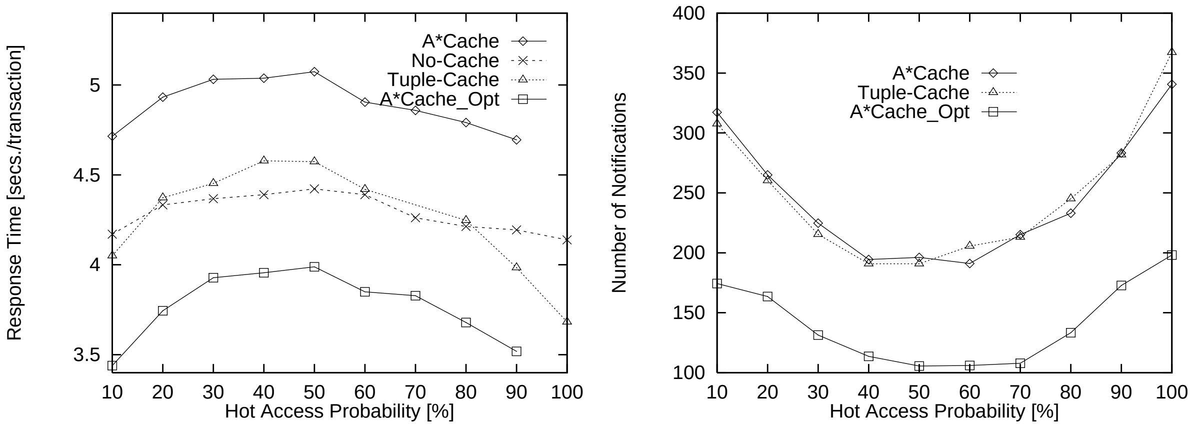


Figure 8.9: *Varying HotAccess, HiCon Unclustered Access*

A*Cache is again observed to perform the worst, with No-Cache and Tuple-Cache having comparable response times. A*Cache_Opt has the best response, and also the least number of notifications, because its policy of remote updates on a cache miss pays off in this experimental environment. The 'peak' effect in the read-only scenario (Figure 7.16) is also present in the read-write case. Recall that this effect is caused by the nature of the HiCon access and pattern of server buffer reuse in our experimental environment. It is interesting to observe the reverse 'trough' effect in terms of the number of notifications. The minimum number of notifications occurs around 50% hot access, when access is split uniformly across the whole database, producing the least number of data update conflicts among the different clients. Because of its remote update policy on cache miss, A*Cache_Opt has the fewest update conflicts of the three caching schemes in the setting of these experiments.

8.6 Summary of Read-Write Results

In this chapter, we have presented the results of simulation experiments with read-write workloads. The differences in the update handling strategies of the four caching schemes were summarized at the beginning of the chapter. These differences help us interpret the behavior observed in our simulation experiments. The effect of various parameters have been analyzed for all four workload types, Private, HotCold, HiCon, and Uniform. For moderately high amount of updates (about 20%) in the shared regions of the Private and HotCold workloads, the caching schemes still provide good performance benefits. Non-shared writes among the clients yield better improvements for all the caching schemes, since notification messages are not sent. In general, utilization of the server CPU is higher for the caching schemes because of the cost of generating notifications, while the number of disk accesses is lower due to caching. The trade-off is between update notifications and local data processing, with larges caches generating more notification traffic. The HiCon scheme represents high data contention on a common shared region, and its performance is therefore the most affected by the amount of shared data writes.

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. We use simple range queries in our workloads, the query predicates being integer intervals. For more complex queries, increased cost of predicate comparison deteriorates the performance of the A*Cache schemes, since notification generation and processing happens at the level of cached predicates. In contrast, the Tuple-Cache scheme is dependent on the use of server indexes, and also requires tuple-level filtering of update notifications at the server. 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. The results obtained in our simulation experiments demonstrate that substantial benefits are obtained for all the caching schemes even when data update probabilities are moderately high, upto 40% in some scenarios (Figure 8.6). However, performance deteriorates for high write contention on a shared region, such as the environment of Figure 8.8.

Chapter 9

Conclusions and Future Work

Client-server configuration is a popular architecture for modern databases. For such environments, local caching of data potentially offers substantial performance benefits by utilizing the computing power and storage capacity of today's powerful clients. In this dissertation, we have introduced the concept of associative client-side caching based on query predicates. Associative access implies that the data can be retrieved using predicates on the attribute values of cached items, and not merely through their unique identifiers. The A*Cache scheme studied in this thesis uses client memory and CPU to process associative queries locally, effectively exploiting the locality of data reference.

In A*Cache, query results are loaded dynamically into a local store, and cache descriptions derived from query predicates are used to describe the cache contents. In contrast to identifier-based caching, the predicate descriptions in A*Cache allow associative queries from subsequent transactions to be examined locally for containment, and executed at the client if the data is available. Central indexes defined at the server are therefore not required for local containment reasoning, and the clients can independently define local access paths depending on their specific data usage patterns. The goal is to reduce network traffic and response times, and to distribute the server load to the clients, thereby increasing system performance and scalability.

In the first half of the dissertation, we described in detail the framework for our A*Cache system. First, we introduced the notion of associative caching through a motivating example. We then presented the architecture of an A*Cache system in terms of the various functional modules at the clients and at the server. An execution model for transactions was defined with respect to a concurrency control scheme that supports serializable transactions in A*Cache. In order to support local execution of queries, the usual lock-based concurrency control protocol for centralized client-server systems is extended with update notification messages from the server, and an extra commit verification step is included to detect violations of serializability constraints. Thus, the concurrency control scheme is semi-optimistic in nature, allowing local query execution but possibly causing a transaction to abort if commit verification fails. We have shown the correctness of our transaction execution scheme by demonstrating that it preserves serializability.

We have also examined efficiency and design issues for good dynamic behavior in A*Cache, and identified new opportunities for optimization. For instance, the client can be conservative in reasoning about its cache contents, while the server must be liberal in notifying clients of cache updates. Such an optimization can potentially reduce the cost of cache containment reasoning and maintenance, without compromising the correctness of transaction execution. We introduce query trimming and query augmentation as techniques that can be used to rewrite the user query to better utilize the cache. Query trimming eliminates from a remote query submission those portions of the query for which data is locally available, while query augmentation fetches essential data such as relation keys for easier maintenance of the cache. Other optimizations such as merging and indexing of the query predicates in a cache description can also be employed to reduce the complexity of reasoning with predicates.

In the latter half of the dissertation, we studied the performance of A*Cache through simulation in order to determine the viability and characteristics of our proposed scheme in a dynamic caching environment. In order to compare our performance results, we introduced three other client-server database systems, namely, No-Cache, Tuple-Cache, and A*Cache_Opt. The No-Cache system has a traditional configuration, with server-side processing of all database operations and no data caching or query execution at clients. Tuple-Cache is a caching system that stores tuples only without any predicate-based cache descriptions, causing query containment checks to be performed at the server with the help of central indexes. In Tuple-Cache, only the tuples missing from a result set are fetched back from the server. A*Cache_Opt is an extended version of the basic A*Cache scheme. Like A*Cache, it performs local containment reasoning using the predicate description of a cache. However, unlike A*Cache, A*Cache_Opt also takes into account partial cache hits, executing a trimmed query at the server to fetch only those tuples that are not available at the client. A*Cache_Opt uses an additional optimization for updates, in which data writes are performed remotely at the server in the case of a cache miss. These differences in the query and update processing strategies of the different caching schemes are discussed in detail in Chapters 7 and 8, since they are important for interpreting our simulation results.

We developed simulation models for the three types of caching systems by extending an existing client-server model with client-side data storage and appropriate logical modules for cache management. A suitable workload model was also designed for our associative caching scenario. Our model of the workload is based on a standard database benchmark, namely, the single-user Wisconsin benchmark [Gray93a], extended with the multi-user data locality and contention models of [Fran93, Care94]. The system parameters and data usage profiles were chosen carefully, so as to be representative of real systems. The discrete-event simulation language C++/CSIM was used to implement the simulator. Our implementation of the simulator was validated against a commercial database system, by running generated query traces against the database and on the simulator. The validation process was performed for read-only queries with a single client, and was used to establish that the simulator behavior was reasonably close to that of a real database system.

Finally, numerous experiments were run on the simulator in order to study the effects of various parameters. Our goal was to verify through simulation the effectiveness of our associative caching scheme for workloads that represent typical multi-client environments, and to compare its performance against the No-Cache and Tuple-Cache systems. Below, we summarize our conclusions in this regard.

For all read-only scenarios, the performance of all the three caching systems is found to be better with respect to No-Cache, since there is no cost involved in update notifications. The benefits are more pronounced for large caches, slow networks, and data access through an unclustered index, because large caches produce less data traffic on the network, and local query processing is independent of the data clustering on server disk. The caching systems also provide significant benefits when the private hot region of a client fits entirely in its cache, since the shared server buffer is bypassed for this access to data in this portion. Additionally, all three caching schemes are less sensitive to changes in shared system resources, such as the server buffer, central disk, server CPU, and the network speed. The caching schemes therefore scale much better with larger number of clients compared to the No-Cache case.

Now, comparing the relative performance of the three caching schemes, we find that the basic un-optimized version of A*Cache performs worse than Tuple-Cache in general. As discussed in Section 7.1, 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. 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 result is not surprising, since 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, thereby yielding the best results.

In a few cases, such as for the large cache sizes in Figures 7.1 and 7.2, and the scenario shown in Figure 7.4, Tuple-Cache performance is comparable to that of A*Cache_Opt. For large caches, there are many predicates in the client cache description, so that local containment checking and predicate management at the server consumes more CPU cycles, offsetting the cost of an extra network trip in Tuple-Cache. Also, when the cost of predicate comparison is increased, then local containment checks and maintenance of cache descriptions are more expensive, causing the response time of the A*Cache schemes to rise accordingly. Beyond a certain limit (75,000 — 80,000 instructions in our experimental setting of Figure 7.5) of the predicate comparison cost, the A*Cache schemes can perform worse than both No-Cache and Tuple-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. 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.

With respect to the different types of workloads, Private and HotCold access models have very similar results. Performance characteristics for Private access are slightly better than HotCold when the cache is large enough to hold the hot region for a client, since for the Private workload, the hot region is used exclusively by a single client, and the cold region is also smaller compared to HotCold. Gains from caching are poor in the Uniform environment, because data is accessed randomly over the entire database, and the effectiveness of the client cache and server buffer are significantly reduced.

Next, consider read-write environments in which transactions can modify data. Low to moderate values of write probabilities provide similar benefits as in the read-only scenario, while a larger volume of updates causes an increase in cache maintenance costs, and a corresponding reduction in the benefits of local caching. The trade-off is between update notifications and local data processing, with larger caches causing more cache hits but also more notification traffic. As shown in Figures 8.1 and 8.6 for Private and HotCold workloads, all the three caching schemes perform better than No-Cache for 20% update transactions and 50% tuple update probability. Similar results are observed even with 40% update transactions (Figures 8.4 and 8.6) in our experimental setting. Although scenarios certainly exist where the costs of caching outweigh its benefits, our simulation study has established that it is beneficial to have associative caching on the clients even for moderately high update contention.

The relative differences in the performance characteristics of the three caching schemes in the read-write scenario are a result of complex interplay between the various factors involved in their handling of data updates and notifications. As discussed in Section 8.1, the three caching schemes handle data updates quite differently, and accordingly exhibit significant differences in their performance. In comparison to server-based containment checks in the Tuple-Cache scheme that incurs extra network round-trips, the predicate-based notifications in A*Cache and A*Cache_Opt are CPU-intensive at the server, and can also have larger data volume for a single notification. The latter condition can occur in the two A*Cache schemes, since filtering of relevant updates is at the level of cached predicates, and some updated tuples that are irrelevant for a cache can be sent over to a client. In contrast, notification in Tuple-Cache is based on tuples, and not on query predicates, thus being less expensive in certain scenarios, as for example in Figure 8.4. In the A*Cache schemes, the costs of maintaining the local data and predicates increase with the predicate comparison cost, as well as with other factors such as the size of the cache and the number of data updates, causing deterioration in performance. However, as noted in the discussion on read-only results above, Tuple-Cache assumes data access based on server indexes, and is therefore applicable for a smaller class of queries than the predicate-based A*Cache and A*Cache_Opt schemes.

There is another subtle difference between the Tuple-cache and A*Cache schemes, which depends on our choice of the concurrency control protocol, and affects the performance in the presence of update transactions. Recall from Section 5.2.6 that Tuple-Cache fetches only those tuples from the server that are missing from its cache. In terms of 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 volume but at the same time, locking the tuples at the server. In a read-only scenario, the extra network cost of fetching unnecessary tuples dominates in A*Cache, and its performance is generally worse than Tuple-cache. However, when updates are being performed, acquiring the locks on tuples serves to reduce conflicting updates by clients, which would otherwise have to be resolved by aborting transactions 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 A*Cache schemes, with A*Cache_Opt having the least number of notifications of the three.

The contention characteristics are also different for each of our access models, Private, HotCold, HiCon and Uniform, producing different performance profiles. For example, in the Private access model, the hot region of a client is not shared, and update conflicts happen only on the shared cold portion of the database. In contrast, the HotCold model permits cold access to the hot region of a client by others, thus generating notifications also for the hot data stored in a client cache. Therefore, the number of notifications produced in the HotCold scenario is larger than in Private, for the same distribution of hot data among clients. The HiCon access model represents high contention on a shared hot region. As expected, the number of notifications generated in this scenario can be large, with a corresponding decrease in the response time (Figure 8.8). However, A*Cache_Opt can still yield benefits in this high-conflict scenario (see Figure 8.9), since it performs updates at the server on a cache miss, thus acquiring central locks and reducing the chances of concurrent writes to the same data.

In conclusion, this dissertation has proposed and analyzed an associative caching scheme based on query predicates. The execution model for transactions in this caching scheme has been defined, and its preservation of the serializability property demonstrated. We have examined, both qualitatively and through simulation, the costs and benefits of our proposed scheme in a dynamic caching environment, and for workloads that are representative of data usage in client-server environments. Our simulation results clearly demonstrate the effectiveness of associative caching for read-only environments, and also for read-write scenarios with moderately high data update probabilities.

9.1 Future Work

Many important issues related to associative caching remain unexplored in this dissertation. Interesting questions can arise in the design and implementation of an associative caching system based the techniques presented here. Below, we discuss a few such issues, and outline directions for future research.

- **Management of cache space**

  In Chapter 3, we briefly discussed effective management of space by a client. For our simulation, we adopted a cache replacement policy based on purging of least recently used predicates. More advanced space management techniques, such as the ones studied in [Dar96, Sche96], can be employed and their effects on A*Cache performance investigated. Additionally, analytical models can be developed for simple but representative scenarios. For environments in which data usage patterns are not static but change dynamically, the client could maintain local data usage statistics and use intelligent 'learning' techniques to achieve effective space utilization.

- **Maintenance of cached data**

  For our simulation study, we assumed that the cache is always refreshed using update notifications. Alternative policies for cache maintenance are discussed briefly in Section 3.3.4. Further work remains to be done in comparing different cache maintenance strategies. For instance, instead of refreshing the cache, the client may choose to invalidate updated data. Possible refinements of the update-always and invalidate-always schemes include invalidating the cached data in a 'cold' region and updating only the 'hot' data for the client. Other intelligent techniques for cache maintenance, and their adaptation to dynamic usage patterns, can also be devised.

Mobile computing and the World Wide Web are other environments where the associative caching framework of A*Cache is applicable. For mobile clients, local caches should be capable of disconnected operation, updating the stale data upon re-connect. Likewise, Web clients might desire update notifications upon demand, instead of automatic transmission of updates from the server. Investigating the performance of A*Cache in these new environments is an open area of research.

- **Local query execution**

  For our simulation, we assumed that the local store uses main memory only. If the cache size is large, clients could store data on local disks, thereby altering the performance characteristics. The techniques presented in this thesis can be extended directly to apply to this scenario. When the results are large, or involve joins between multiple relations, the creation of indexes on data cached at the client can also improve the turn-around time of local query processing. Dynamic tailoring of indexes based on data usage patterns at a client can be based on the adaptive optimization techniques presented in [Derr92].

- **Optimizations in reasoning with predicates**

  In Chapter 4, we presented some optimization techniques that can be applied for reasoning with predicates in cache descriptions. For example, the client can be conservative in its cache containment checks, while the server must be liberal in generating notifications. Heuristics for such approximation techniques, and efficient query containment algorithms for determining cache containment are topics for future efforts. Predicate indexing and merging can also be used by both the server and the client to organize and manage cached predicates. Predicate indexing techniques developed in the context of active and spatial databases [Hans90, Same90] are applicable for A*Cache.

- **Weak consistency models for client caching**

  For the purposes of this dissertation, we adopted a semi-optimistic concurrency control protocol that performs an extra commit verification step to detect any violations of transaction serializability. Future avenues of research include the development of appropriate consistency models for lower levels of data isolation in the presence of client-side caching, and formalizing the execution model of A*Cache in such scenarios. Work done in the context of distributed databases [Davi85, Wied90], and on view consistency [Hull96, Zhug95] can be adapted to the client caching environment.

- **Development of a prototype**

  Clients with large memories and fast CPU's are a technical reality today. Based on the work reported in this dissertation, it is feasible to develop a prototype system with associative client-side caching and server notifications. In fact, our colleagues Marie-Anne Neimat and Kurt Shoens at Times Ten are interested in developing such a prototype, using a main memory database as the client-side cache and a commercial database server as the central data store. In the absence of direct support for client caching at the server, replication facilities commonly provided by databases could be used to generate the required notification messages for cache maintenance.

Bibliography

[Adya95] A. Adya, R. Gruber, B. Liskov, and U. Maheswari, "Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks"; *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, CA, May 1995.

[Agar88] A. Agarawal, R. Simoni, J. Hennessey, and M. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence"; *Proceedings of the 15th International Symposium on Computer Architecture,* Honolulu, June 1988.

[Agra87] R. Agrawal, M.J. Carey, and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications"; *ACM Transactions on Database Systems*, Vol. 12, No. 4, December 1987.

[Alon90] R. Alonso, D. Barbara, and H. Garcia-Molina, "Data Caching Issues in an Information Retrieval System"; *ACM Transactions on Database Systems*, Vol. 15, No. 3, September 1990.

[ANSI92] ANSI X3.135-1992, "American National Standard for Information Systems — Database Language — SQL"; November 1992.

[Atlu95] V. Atluri, E. Bertino, and S. Jajodia, "A Theoretical Formulation of Degrees of Isolation"; *Technical Report*, George Mason University, VA, 1995.

[Bars90] T. Barsalou and G. Wiederhold, "Complex Objects for Relational Databases"; *Computer Aided Design*, Vol. 22, No.8, 1990.

[Bars91] T. Barsalou, N. Siambela, A. M. Keller, and G. Wiederhold, "Updating Relational Databases through Object-Based Views"; *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Denver, CO, May 1991.

[Basu97] 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.

[Bere95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A Critique of ANSI SQL Isolation Levels"; *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1995, San Jose, CA.

[Bern87] P.A. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Systems*; Addison-Wesley, Reading, Massachusetts, 1987.

[Bert92] E. Bertino and D. Musto, "Query Optimization by Using Knowledge about Data Semantics"; *Data & Knowledge Engineering*, Vol. 9, No. 2, 1992.

[Blak86] J.A. Blakeley, P.-A. Larson, and F. W. Tompa, "Efficiently Updating Materialized Views"; *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, DC, May 1986.

[Blak89] 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.

[Brei92] Y. Breibart, H. Garcia-Molina, and A. Silberschatz, "Overview of Multidatabase Transaction Management"; *The VLDB Journal*, Vol. 1, No. 2, Oct. 1992.

[Bune78] P.O. Buneman and E.K. Clemons, "Efficiently Monitoring Relational Databases"; *ACM Transactions on Database Systems*, Vol. 4, No. 3, September 1979.

[Care91a] M. Carey, M. Franklin, M. Livny, and E. Shekita, "Data Caching Tradeoffs in Client-Server DBMS Architecture"; *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Denver, CO, May 1991.

[Care91b] M. Carey and M. Livny, "Conflict Detection Tradeoffs for Replicated Data"; *ACM Transactions on Database Systems*, Vol. 16, No. 4, December 1991.

[Care94] M. Carey, M.J. Franklin, and M. Zaharioudakis, "Fine-Grained Sharing in a Page Server OODBMS"; *ACM SIGMOD International Conference on Management of Data*, Minneapolis, MI, May 1994.

[Cast97] M. Castro, A. Adya, B. Liskov, A.C. Myers, "HAC: Hybrid Adaptive Caching for Distributed Storage Systems"; *Symposium on Operating Systems Principles*, St. Malo, France, October 1997.

[Catt91] R.G.G. Cattell, *Object Data Management*; Addison-Wesley, Reading, Massachusetts, 1991.

[Ceri91] S. Ceri and J. Widom, "Deriving Production Rules for Incremental View Maintenance"; *Proceedings of the 17th International Conference on Very Large Data Bases*, Barcelona, Spain, September 1991.

[Ceri84] S. Ceri and G. Pelagatti, *Distributed Databases:Principles and Systems*; McGraw-Hill, New York, 1984.

[Davi82] J. Davidson, "Natural Language Access to Databases: User Modeling and Focus"; *Proceedings of the CSCSI/SCEIO Conference*, Saskatoon, Canada, May 1982.

[Davi85] S. Davidson, H. Garcia-Molina, D. Skeen, "Consistency in Partitioned Networks"; *ACM Computing Surveys*, Vol. 17, No. 3, September 1985.

[Derr92] M. A. Derr, "Adaptive Optimization In A Database Programming Language"; PhD thesis, Technical Report No. STAN-CS-92-1460, Computer Science Department, Stanford University, December 1992.

[Elbe94] B. Elbert and M. Bobby, *Client/Server Computing, Architecture, Application, and Distributed Systems Management*; Artech House, Boston, London, 1994.

[Dar96] S. Dar, M. J. Franklin, B. Jonsson, D. Srivastava, and M. Tan, "Semantic Data Caching and Replacement"; *Proceedings of the 22nd International Conference on Very Large Data Bases*, Bombay, India, September, 1996.

[Deli92] A. Delis and N. Roussopoulos, "Performance and Scalability of Client-Server Database Architectures"; *Proceedings of the 18th International Conference on Very Large Data Bases*, Vancouver, B.C., Canada, 1992.

[Dewi90] D.J. DeWitt, D. Maier, P. Futtersack, and F. Velez, "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems"; *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, 1990.

[Eswa76] K.P. Eswaran, J.N. Gray. R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System"; *Communications of the ACM*, Vol. 19, No. 11, 1976.

[Fink82] S.J. Finkelstein, "Common Subexpression Analysis in Database Applications"; *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Orlando, Florida, June 1982.

[Fran93] M.J. Franklin, "Caching and Memory Management in Client-Server Database Systems"; PhD thesis, Technical Report No. 1168, Computer Sciences Department, University of Wisconsin-Madison, 1993.

[Fran93] M.J. Franklin, M.J. Carey, and M. Livny, "Local Disk Caching for Client-Server Database Systems"; *Proceedings of the 19th International Conference on Very Large Data Bases*, Dublin, Ireland, August 1993.

[Fran96] M.J. Franklin, B.T. Jonsson, and D. Kossmann, "Performance Tradeoffs for Client-Server Query Processing"; *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1996.

[Gall95] R. Gallersdorfer and M. Nicola, "Improving Performance in Replicated Databases through Relaxed Coherency"; *Proceedings of the 21st International Conference on Very Large Data Bases*, Zurich, Switzerland, Sept. 1995.

[Good83] J. Goodman, "Using Cache Memory to Reduce Processor Memory Traffic"; *Proceedings of the 10th International Symposium on Computer Architecture,* Stockholm, Sweden, June 1983.

[Gott96] V. Gottemukkala, E. Omiecinski, and U. Ramachandran, "Relaxed Index Consistency for a Client-Server Database"; *Proceedings of the International Conference on Data Engineering*, Birmingham, U.K., 1996.

[Gray93a] J. Gray, *The Benchmark Handbook for Database and Transaction Processing Systems*; Morgan Kaufmann Publishers, Inc., San Mateo, USA, 1993.

[Gray93b] J. Gray and A. Reuter, "Isolation Concepts"; in *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[Gray94] J. Gray, R. Lorie, G. Putzolu, and I. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Database"; in *Readings in Database Systems*, Second Edition, Chapter 3, Michael Stonebraker, Ed., Morgan Kaufmann 1994 (paper was originally published in 1977).

[Gupt93] A. Gupta, I.S. Mumick, and V.S. Subrahmanian, "Maintaining Views Incrementally"; *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, DC, May 1993.

[Gupt95] A. Gupta and I. S. Mumick, "Maintenance of Materialized Views: Problems, Techniques, and Applications"; *IEEE Data Engineering Bulletin*, Vol. 18, No. 2, June 1995.

[Hans90] E.N. Hanson, M. Chaabouni, C.H. Kim, and Y.W. Wang, "A Predicate Matching Algorithm for Database Rule Systems"; *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Atlantic City, NJ, May 1990.

[Hans93] E.N. Hanson and J. Widom, "Rule Processing in Active Database Systems"; *International Journal of Expert Systems Research and Applications*, Vol. 6, No. 1, 1993.

[Howa88] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, "Scale and Performance in a Distributed File System"; *ACM Transactions on Computer Systems,* Vol. 6, No. 1, February 1988.

[Hull96] R. Hull and G. Zhou, "A Framework for Supporting Data Integration Using the Materialized and Virtual Approaches"; *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 1996, Montreal, Canada.

[Hunt 79] H.B. Hunt III, and D.J. Rosenkrantz, "The Complexity of Testing Predicate Locks"; *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1979.

[Jaro92] J. W. Jaromczyk and G.T. Toussaint, "Relative Neighborhood Graphs and Their Relatives"; *Proceedings of the IEEE*, Vol. 80, No. 9, September 1992.

[Jord81] J.R. Jordan, J. Banerjee, and R.B. Batman, "Precision Locks"; *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Ann Arbor, MI, April 1981.

[Kame92] N. Kamel, and R. King. "Intelligent Database Caching Through the Use of Page-Answers and Page-Traces"; *ACM Transactions on Database Systems*, Vol. 17, No. 4, 1992.

[Kawa96] A. Kawaguchi, D. Lieuwen, I.S. Mumick, D. Quass, and K.A. Ross, "Concurrency Control Theory for Deferred Materialized Views"; *Proceedings of the International Conference on Database Theory (ICDT)*, 1997.

[Kell85] A.M. Keller, "Updating Relational Databases through Views"; PhD thesis, Technical Report No. STAN-CS-85-1040, Computer Science Department, Stanford University, February 1985.

[Kell94] A.M. Keller and J. Basu, "A Predicate-based Caching Scheme for Client-Server Database Architectures"; *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS)*, Austin, Texas, September 1994.

[Kell96] A.M. Keller and J. Basu, "A Predicate-based Caching Scheme for Client-Server Database Architectures"; *The VLDB Journal*, Vol. 5, No. 1, Jan 1996.

[King84] J.J. King, "Query Optimization by Semantic Reasoning"; University of Michigan Press, Ann Arbor, MI, 1984.

[Lars87] P.-A. Larson and H.Z. Yang, "Computing Queries from Derived Relations: Theoretical Foundation"; *Research Report CS-87-35*, Computer Science Department, University of Waterloo, 1987.

[Lee90] B. S. Lee, "Efficiency in Instantiating Objects from Relational Databases Through Views"; PhD thesis, Technical Report No. STAN-CS-90-1346, Computer Science Department, Stanford University, December 1990.

[Levy90] E. Levy and A. Silbershatz, "Distributed File Systems: Concepts and Examples"; *ACM Computing Surveys,* Vol. 22, No. 4, December 1990.

[Levy93] A. Levy and Y. Sagiv, "Queries Independent of Updates"; *Proceedings of the 19th International Conference on Very Large Data Bases*, Dublin, Ireland, August 1993.

[Levy95] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava, "Answering Queries Using Views"; *Proceedings of the Fourteeth Symposium on Principles of Database Systems (PODS)*, San Jose, CA, May 1995.

[Li89] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems"; *ACM Transactions on Computer Systems,* Vol. 7, No. 4, November 1989.

[Lome94] D. Lomet, "Private Locking and Distributed Cache Management"; *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, Austin, TX, September 1994.

[Mesq94] Mesquite Software, "C++/CSim User's Guide"; Austin, Texas, USA, August 1994.

[Orac95] Oracle Corporation, "Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7"; White Paper, Part No. A33745, July 1995.

[Orac96] Oracle Corporation, *Oracle 7 Server Concepts Manual*, April 1996.

[Poes97] M. Poess, "Simulation Analysis of SQL*Cache"; Diplomarbeit, University of Karlsruhe, Germany, January 1997.

[Riss 77] J. Rissanen, "Independent Components of Relations"; *ACM Transactions on Database Systems*, Vol. 2, No. 4, December 1977.

[Rose80] D.J. Rosenkrantz and H.B. Hunt, "Processing Conjunctive Predicates and Queries"; *Proceedings of the 6th International Symposium on Very Large Databases*, New York, NY, 1980.

[Rous85] N. Roussopoulos and D. Leifker, "Direct Spatial Search on Pictorial Databases Using Packed R-Trees"; *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Austin, 1985.

[Rous86] N. Roussopoulos and H. Kang, "Principles and Techniques in the Design of ADMS±"; *IEEE Computer*, December 1986.

[Rous91] N. Roussopoulos, "The Incremental Access Method of View Cache: Concepts, Algorithms, and Cost Analysis"; *ACM Transactions on Database Systems*, Vol. 16, No. 3, 1991.

[Rous95] N. Roussopoulos, C.M. Chen, and S. Kelly, "The ADMS Project: Views R Us"; *IEEE Data Engineering Bulletin*, Vol. 18, No. 2, June 1995.

[Same90] Hanan Samet, *Spatial Data Structures*; Addison-Wesley, Reading, Massachusetts, 1990.

[Sche96] P. Scheuermann, J. Shim, and R. Vingralek, "WATCHMAN: A Data Warehouse Intelligent Cache Manager"; *Proceedings of the 22nd International Conference on Very Large Data Bases*, Bombay, India.

[Sell87] T. Sellis, "Intelligent Caching and Indexing Techniques for Relational Database Systems"; *Technical Report CS-TR-1927*, Computer Science Department, University of Maryland, College Park, MD, 1987.

[Sell92] T. Sellis and C.-C. Lin, "A Geometric Approach to Indexing Large Rule Bases"; in *Advances in Database Technology - EDBT '92. 3rd International Conference on Extending Database Technology Proceedings*, A. Pirotte, C. Delobel, and G. Gottlob (editors), Berlin, Germany, Springer-Verlag, 1992.

[Shet91] A.P. Sheth and A.B. O'Hare, "The Architecture of *BrAID*: A System for Bridging AI/DB Systems"; *Proceedings of the Seventh International Conference on Data Engineering*, Kobe, Japan, April 1991.

[Shri96] L. Shrira, B. Liskov, M. Castro, and A. Adya, "Fragment Reconstruction: A New Cache Coherence Scheme for Split Caching Storage Systems"; *Workshop on Persistent Objects*, Cape May, New Jersey, USA, May 1996.

[Sriv96] D. Srivastava, S. Dar, H.V. Jagadish, and A.Y. Levy, "Answering Queries with Aggregation Using Views"; *Proceedings of the 22nd International Conference on Very Large Data Bases*, Bombay, India, September 1996.

[Stau96] M. Staudt and M. Jarke, "Incremental Maintenance of Externally Materialized Views"; *Proceedings of the 22nd International Conference on Very Large Data Bases*, Bombay, India, September 1996.

[Ston90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos, "On Rules, Procedures, Caching, and Views in Data Base Systems"; *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Atlantic City, NJ, May 1990.

[Ston94] M. Stonebraker, "Mariposa: A New Architecture for Distributed Data"; *IEEE Conference on Data Engineering*, Houston, 1994.

[Ullm90] J. D. Ullman, *Principles of Database and Knowledge-Base Systems*, Volumes 1 and 2, Computer Science Press, Inc., 1988.

[Wang91] Y. Wang and L.A. Rowe, "Cache Consistency and Concurrency Control in a Client-Server DBMS Architecture"; *ACM SIGMOD International Conference on Management of Data*, Denver, CO, May 1991.

[Wied86] G. Wiederhold, "Views, Objects, and Databases"; *IEEE Computer*, December 1986.

[Wied90] G. Wiederhold and X. Qian, "Consistency Control of Replicated Data in Federated Databases"; *Proceedings of the IEEE Workshop on Management of Replicated Data*, Houston, Texas, Nov. 1990.

[Qian91] X. Qian and G. Wiederhold, "Incremental Recomputation of Active Relational Expressions"; *IEEE Transactions on Knowledge and Data Engineering*, Vol. 3, No. 3, September 1991.

[Wilk90] K. Wilkinson and M.-A. Neimat, "Maintaining Consistency of Client-Cached Data"; *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, 1990.

[Zaha97] M. Zaharioudakis and M. Carey, "Highly Concurrent Cache Consistency for Indices in Client-Server Database Systems";, *Proceedings of the ACM SIGMOD International Conference on Management of Data,* Tucson, Arizona, May 1997.

[Zhug95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J.Widom, "View Maintenance in a Warehousing Environment"; *Proceedings of the ACM SIGMOD International Conference on Management of Data,* San Jose, CA, May 1995.