Performance Evaluation of a Predicate-Based Caching System: SQL*CACHE

Diplomarbeit
von

Meikel Pöss

Januar 1997

Betreuer:
Prof. Dr. W. Zorn
Dr. Martin Haas

Institut für Betriebs- und Dialogsysteme
Fakultät für Informatik
Universität Karlsruhe

Hiermit bestätige ich, Meikel Pöss, die vorliegende Arbeit selbständig angefertigt zu haben und keine anderen als die angegebenen Quellen benutzt zu haben.

Karlsruhe, den 15 Januar 1997

(Meikel Pöss)

to my parents

Acknowledgements

This thesis is an outcome from many peoples' support. Above all I want to thank my parents Roswitha and Heinrich Pöss, who supported me all my life. Without their love and faith in me and my abilities, I would not have finished my studies. I also want to thank my grandma, Helene Steenmanns, who always believed in my abilities.

Special thanks to Gio Wiederhold, who invited me to Stanford University and whose suggestions have been very useful. I am especially greatful to Julie Basu, who not only helped me in programming the simulator but who was always around for discussions and for giving suggestions. Many thanks to Kurt Shoens. He helped a lot with his immense experience regarding performance analysis in the field of databases. I am very grateful for his measurements of the ORACLE DBMS. Thanks to Arthur Keller. His discussion and suggestions were very appreciated.

Thanks to Martin Haas who gave me a lot suggestions regarding the structure and contents of this thesis

I want to thank Vikky Yamazaki for her support, discussion and understanding in difficult times. She and Norman Greenberg corrected my sometimes poorly written English to a readable level. Thanks a lot.

I will not forget Marianne Siroker for dealing with all the bureaucracy and Elena Baralis for contacting Gio.

I want to thank all members of the database group at Stanford University. You all made my stay at Stanford a very nice and sucessful one.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The client-server architecture has proved to be a popular one for modern database management systems. It involves a server process that manages a centralized repository of persistent data, and handles requests for data retrieval and update from many clients processes. The clients could be located on the same machine as the servers, or may be executing in different machines altogether. With the current prevalence of distributed computing, the latter scenario is most common, where client transactions are initiated from desktop workstations and communicate with the database server across a local-area or a wide-area network.

A traditional assumption in the design of client-server databases has been that the client machines have very limited resources. Accordingly, *query-shipping* systems were usually designed, in which client functionality is limited to transmission of queries and updates across the network to the server, and presentation of the received results to the user. However, recent advances in computer technology have made substantial computing power and large memories available at a small fraction of their earlier costs, with the result that client workstations are often high-performance machines in their own right. It has therefore become possible to make use of client resources to enhance the overall system performance. Increased client functionality can potentially lower query response times, reduce network traffic, increase server throughput, as well as provide better scalability with larger number of clients.

A result of assuming low-end clients is that data caching is usually done on the server-side only, based on buffering of frequently accessed disk blocks or pages in main memory. A server "buffer pool" of cached pages is used to avoid disk traffic in typical commercial relational databases with "standard" client-server configurations and query-shipping architectures [17]. Recent work [11] has demonstrated that in addition to server buffers, dynamic local caching of data at client sites can also improve the performance of a client-server database, especially when the operational data spaces of clients are mostly disjoint.

The caching schemes investigated in [11] are object ID or page ID based schemes, in which cached objects are accessed by their ID. Such caching does not support *associative* access to cached data, and is adequate only for *navigational* queries as in object-oriented databases. An associative caching scheme provides *content-based* access to the cached data, in addition to data retrieval only through object or page IDs. Associative queries are very common in relational database systems, and hence evaluation of associative queries on the cached data is important for such systems.

A client-side data caching scheme called *SQL*Cache* that supports local execution of associative queries was first proposed in [15]. A SQL*Cache loads query results dynamically, and supports local execution of SQL queries through containment reasoning on a *predicate description* of cached queries. If a client determines from this SQL*Cache description that a new query cannot be answered from its cache, then the query is shipped to the server for processing and the returned result (optionally) is added to the SQL*Cache, whose description is updated appropriately. On the other hand, a query contained in the cached set of queries is evaluated by the client locally. In order to maintain the consistency of the cached data, predicate descriptions of SQL*Cache contents are also stored at the server, which uses them to notify each client of committed updates that are possibly relevant for its cache. Such notification may cause a transaction currently running at the client site to abort and restart.

The goal of this thesis is to simulate the behavior of SQL*Cache, in order to contrast its performance against a "standard" client-server database system with server-side buffering of disk blocks but no client-side caching of data or query predicates. Additionally, I wish to explore the behavior of the SQL*Cache system for different workloads and system characteristics. I have chosen the Wisconsin benchmark for relational database systems [27] for the simulation analysis. This is a well-known benchmark for associative *range* queries in relational database systems. However, it does not model associative updates or data contention among different clients. I have extended this benchmark with predicate-based updates and with different workload profiles for the various clients, so as to be more appropriate for this scenario.

I first develop queueing-based simulation models for both the standard client-server system and for the SQL*Cache scheme. The standard model is a purely query-shipping one. For SQL*Cache, I model the behavior of the SQL engine and data caching at a client site along with the various subcomponents of the caching system such as cache containment reasoning, update notification at the server and update propagation at the clients. Next I verify the standard client-server model against a commercial system. I define several simulation parameters of interest, such as available client memory, the network speed, query and update loads etc., and vary them in simulation experiments. The performance of the standard and SQL*Cache systems are compared, and I try to find fix-points, where the simulator behaves as the commercial system in order to vary one parameter. The simulation system is implemented using the simulation language C++/CSim.

This thesis is organized as follows. Chapter 2 gives an overview of related work in the area. Chapter 3 points out important issues of performance analysis in general and describes some techniques that are used to analyze systems. Chapter 4 discusses different simulation systems, their properties and motivates the decision for CSIM. The following chapter 5 describes in detail SQL*Cache, its main functions and some of its difficulties. Chapter 6 and Chapter 7 design queueing models of the standard client-server model and SQL*Cache. These models are then used to build a simulator in CSIM. Its implementation was done by Julie Basu and Meikel Pöss and is presented in Chapter 8.

Chapter 9 and Chapter 10 verify the models of the standard client-server and SQL*Cache. While the standard client-server is verified against a commercial system, SQL*Cache is compared to the standard client-server model by finding fix points in which both systems behave in the same way.

Future work and a conclusion is mentioned in Chapter 11.

Chapter 2

Related Work

Client-side data caching has been investigated in [6, 7, 2, 28, 30]. All of these papers have presented the results of simulation studies to demonstrate the behavior of their proposed schemes. The focus of this study is local execution of associative queries, which has not been addressed in the above works.

A workstation-mainframe database architecture based on *view cache* techniques [20] is described in [18]. Simulated performance of this caching scheme in a client-server environment is studied in [19]. Update logs are maintained by the server(s), and each query against cached data at a client results in an explicit refresh request to the server(s) to compute and propagate the relevant differential changes from these logs. In contrast, I follow an incremental and flexible notification strategy at the server, and attempt to split the workload of refreshing cached results more evenly amongst the clients and the server. Additionally, I use a standard relational database benchmark for the simulation, which allows a clearer interpretation of the simulation results.

A recent paper [12] investigates a *semantic* data caching scheme that is similar to SQL*Cache in that it supports client-side caching of query results and local execution of associative queries. Their simulation study compares different cache replacement policies in a read-only environment. In contrast, my focus is to investigate the behavior of the SQL*Cache system in the presence of multiple clients.

Chapter 3

Performance Analysis

This chapter deals with the question why a simulation study was chosen to analyze the performance of SQL*Cache. First the major steps that can be found in every performance analysis are described (building a model, mapping it into an implementational model and validating a model). The following section introduces some performance analysis techniques in detail in order to determine which technique is the most appropriate to analyze SQL*Cache. The last section describes the need and issues of benchmarks.

3.1 Steps in Performance Analysis

In this section the main steps of an evaluation study are presented. Beginning with a real or planed system every evaluation study covers the following steps.

1. Building a conceptual model
2. Mapping the conceptual model into an implementational model
3. Running the implementational model
4. Validating the results
5. Interpreting the results

The conceptual model is only a description of the system to be analyzed. The amount of detail representing the systems behavior depends on the purpose of the study. The conceptual study is then mapped into an implementational model that can be used to compute the systems performance measured in throughput, utilization[1], response time etc. After running the implementational model, the

[1] utilization is rather a metric to describe the systems state than a performance metric, but is very useful to analyze systems performance behavior

performance results should be validated against results obtained by monitoring an equivalent real system. If there is no real system available, the conceptual model can be analyzed with another technique in order to get performance results to validate. The last step is interpreting the obtained results by finding relationships between the different input parameters and the performance results.

3.1.1 Building a Model

In order to analyze a system, real or planned, it must be represented as a model. Each model is described in its static structure and its dynamic behavior. While the statical structure consists of different hardware and software components which are described by their individual performance parameters, the dynamical behavior is described as a sequence of systems states. Furthermore, the systems model can be divided into a configuration model and a workload model.

The most common schemes that are used to give a representation of the system are flowcharts, petri-nets and queueing systems.

Configuration Model

The configuration model describes the systems structure consisting of different hardware and software components. Every single componend is characterized by their performance parameters and their interaction to other components. There are open systems which receive input parameters from their environment and closed systems which do not have an interaction with their environment. However, closed systems are easier to analyze.

Workload Model

The workload model describes all the input a system receives over its interface from its environment. The term "system" is not restricted only to the entire computer system, but every single part (hardware/software) can be seen as a separate system with its own input data. Thus the workload model describes input data as well as commands on different levels of abstraction. E.g. a command to a DBMS might be described as a query (records) or if send to a specific subsystem of that DBMS, e.g. the log manager, as a list of required logs (number of records, logtype, ...). The workload of a network might be simplified by messages with a certain length of bytes.

3.1.2 Mapping a Model into an Implementational Model

The result of the previous section is a conceptual model of the system that can be used to apply to different analysis techniques like *Analytical Techniques* (deterministic or probabilistic) or *Simulation Techniques* (see section 3.2). In order to run these techniques the conceptual model is mapped into a special language that enables the use of computers. There exist many tools to interpret petri-nets or queueing-systems and there exist also languages to build simulators like DENET- or CSIM-simulation language (see section 3.2.4).

3.1.3 Validating a Model

The conceptual model is only a description of the system to be analyzed. Since it simplifies the system and abstracts certain aspects of the "real world" it should be verified. The credibility of a model depends on how accurately it represents the real system. Within this validation process the accuracy of a model is to be defined with respect to the performance index, or indices selected for the study. This means that within the validation phase the performance results obtained by running the implementational model are compared with performance figures that are taken by monitoring a real system. Both systems are given the same workload as input parameters. As an initial approach, the parts of the system can be analyzed and validated separately. But since the different parts of the system are highly interactive, an analysis of the whole system is also necessary.

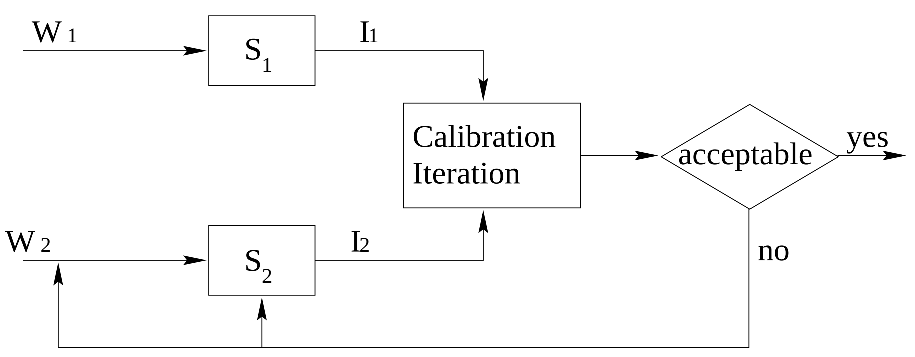

Figure 3.1: *The Iterative Calibration Process*

If the results of the model and the real system diverge, the models parameters might be varied till the results are close enough. The values of the indices resulting from the implementational model and those from the real model must be sufficently close to each other. What is sufficent depends on how detailed the system is modeled. This is an iteractive approach that can lead to wrong results if the parameters are varied till the system reacts correctly with the particular input parameters. This is called overfitting the model. To avoid this mistake, the validation should be repeated with different workloads and different configuration parameters after a few iterations. If the results still diverge the entire model should be revised. Perhaps the abstraction at the model level is too high or facts of the modeled system are represented erroneously.

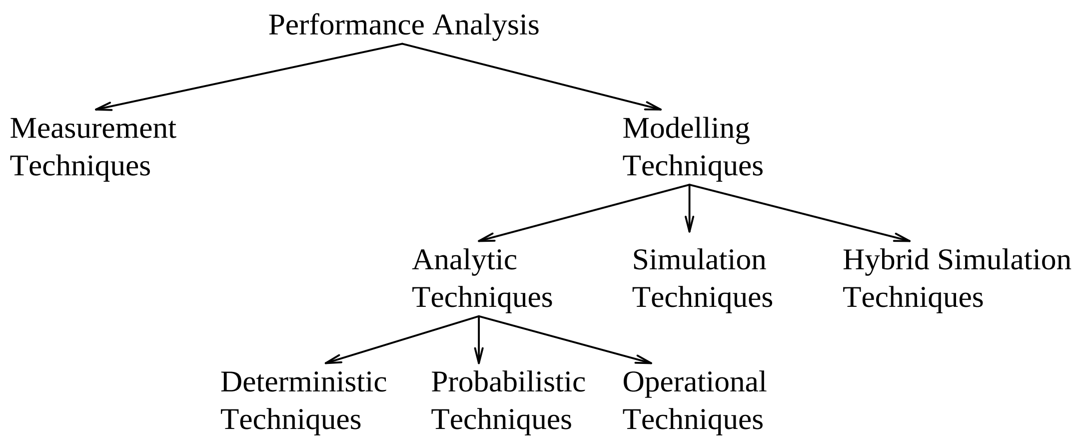

Figure 3.2: *Taxonomy of Performance Analysis Techniques*

Figure 3.1 shows a diagram of the iterative calibration process. Assuming that there is a real system S_1 that produces I_1 as a value for the index with the workload W_1. S_2 now is a model of S_1 which might produces the value I_2 with the workload W_2 which is a model of W_1. After I_1 and I_2 are computed the two values are compared and the difference computed. If the difference between them is acceptable then the model S_2 reppresents the system S_1 accurately enough. If not, parameters of the model must be changed in further iterations. For example, if the query response time is the index, the simple difference $|I_1 - I_2| < e$ can be used as a metric for acceptance.

3.2 Taxonomy of Performance Analysis Techniques

Performance Analysis Techniques can be classified in various ways. The most popular classification which is also presented in [14] and [13] is given in Figure 3.2.

In general *Performance Analysis Techniques* can be divided into *Measurement-* and *Modelling Techniques*. While *Measurement Techniques* are applied to real systems, *Modelling Techniques* can be used either to evaluate real or fictive systems by representing them as models. These models are called conceptual models of the underlaying system. A given system can be represented by many different models. Which model a certain designer chooses depends largely on what purpose the model has, which viewpoint the analyst takes and how detailed the model must be.

The *Modelling Techniques* can further be structured into *Analytical Techniques* (deterministic or probabilistic), *Simulation Techniques* and *Hybrid Simula-*

tion Techniques. This section gives a brief overview over the major techniques. Interested readers are welcome to consider further readings, e.g. ([13],[5],[25],[26]).

3.2.1 Measurement Techniques

Measurement Techniques can be applied only if the system that has to be analyzed is accessable. Given a computer system, first raw data is collected using measurement tools (hardware- or software monitors). After that these data might be analyzed either on-line or off-line. In case of an off-line analysis the raw data is compressed using reduction techniques.

3.2.2 Deterministic Techniques

In *Deterministic Techniques* the system's behavior is modeled with deterministic processes, whose behavior is well known at any time so that the system behavior is predictable for the whole time-window during which the system is analyzed. Deterministic approaches may be applied to worst-case/ best-case or mean-value studies of relatively simple systems, since complex systems are not accurately represented by manageable deterministic models. Therefore, the application domain of *Deterministic Techniques* can be found in the analysis of job scheduling or protocol analysis.

3.2.3 Probabilistic Techniques

Probabilistic Techniques describe the systems behavior by using *Markov and Queueing Models*. These techniques are much more widely applied to computer systems performance analysis than deterministic models, because of their potential for representing complex phenomena in a relatively compact way.

3.2.4 Simulation Techniques

Simulation has been defined in [13] as "... an evaluation technique which represents by a model the behavior of a system in the time domain ...". The observation of the model's behavior in time under defined inputs may be used in evaluation studies. Simulation tools are computers or small physical models of the real system. Performance analysis with simulations is done mostly with computer programs. Important issues of computer simulation are:

1. Discrete or continuous state description
2. Discrete or continuous simulation clock
3. Time- or event driven- simulation

Because of resource restrictions (performance and memory-size) of computer systems, simulation time and state description are always discrete. Whether a simulation is time- or event- driven depends mostly on how many events are occuring in a time window and though how much data must be stored.

3.2.5 Which technique fits best?

Every system goes through the same lifecycle. This lifecycle might be defined as in [14]:

1. Planning
2. High-Level Design
3. Low-Level Design
4. Implementation
5. Test
6. Selection of Alternatives
7. Customer Acceptance
8. Operation
9. Optimisation
10. Capacity Planning

Measurement Techniques need an existing system. Thus they can be used only from the *Implementation* phase to through the Capacity Planning. In *Optimisation* and *Capacity Planning* the system has to be run with different configurations or different, mostly synthetic workloads. It is hard or even impossible to create these scenarios. Therefore measurement techniques are mostly used during the *Implementation* through the *Operation* phase.

Analytical Techniques are employed in the *Planning* and *High-Level Design* phases whose models are quite easy to analyze rapidly with *Analytical Techniques*. *High-Level Design* and *Optimisation* might be too complicated to be analyzed by *Analytical Techniques*. In this case *Simulation Techniques* are used.

3.3 Why Simulation Evaluation for the performance evaluation of SQL*Cache

In order to choose the best technique for the performance evaluation of SQL*Cache, it should to be placed in the general life cycle presented in section 3.2.5. The goal of the performance evaluation of SQL*Cache is to find out under which circumstances this way of client side caching performs better than traditional ways of client caching. The theoretical foundations of SQL*Cache have been developed in various papers [15] [3]. No implementation exists yet.

In a first step a conceptual model of SQL*Cache has been built. This was done using queueing systems as a formal representation of the major parts that are necesary to the performance evaluation. Among these parts are a lock manager with deadlock detection and handling, a query engine, a buffer manager and a notifier at the server side, a network manager between the server and the clients. On the client side are a workload manager, a cache manager, and a query engine.

In order to model the interaction of all these parts they had to be modeled on a very low level (see Section 6.1). Referring to the lifecycle presented in Section 3.2.5 the SQL*Cache system might be found in the low level design phase. Thus only analytical or simulation techniques are applicable. Looking at the level of detail a simulation technique might be the best way analyzing the performance of SQL*Cache.

3.4 Benchmarks

Analyzing a model with a simulation system or measuring a real system needs more than one run of the simulator or one single measurement. The system should be analyzed with different parameter and configuration settings. Thus the performance results from different runs must be comparable and therefore the workload should be similar. Benchmarks were developed in order to guarantee a certain pattern of workload. They produce a synthetical reproduceable workload. Every application domain and computer system has its own benchmark (Whetstone, Drystone, SPEC-Benchmarks, TPC-Benchmarks, AS^{3}AP, OO1-Engineering Database Benchmark, Wisconsin Benchmark, ...). Some of these benchmarks are presented in Section 7.3.

Chapter 4

Simulation Systems

A simulation system is a tool to analyze the systems performance by simulating its behavior in "real live". Most computer simulation systems are time-and-state discrete (see section 3.2.4). In the following section two simulation systems are presented. They are generally used to simulate different models, not necessarlily models of computer systems.

4.1 Simulation Language DENET

DENET (Discrete Event NETwork) is a discrete event simulation language built on top of Modula-2, a general purpose programming language. It was developed 1986 by the Computer Science Department of the Wisconsin-Madison University. DENET allows to define loose coupling modules which leads to fast development of complex parallel systems. Once a module is defined it can be reproduced and huge systems can thereby be easily built. For further detail see DENET User's Guide.

4.2 Simulation Language CSIM

CSIM is a discrete event driven simulation system based on C/C++. A CSIM program modules a system as a collection of processes which interact with each other using mailboxes or simple events. Either synchronized or asynchronized interaction between those processes can be modeled. Facilities of the simulated systems such as processors or disk drives are modeled using user defined CSIM facilities.

CSIM was developed 1987 by Microelectronics and Computer Technology Corporation, Texas. The following is an overview of the main features of CSIM, *Processes*, *Facilities*, *Events*, *Mailboxes* and *Data-reduction and Presentation fea-*

tures.

4.2.1 General Structure of a CSIM Simulation Program

The structure of a CSIM simulation program is similar to an ordinary C++ program with class definitions and function calls. The equivalent function *main* of a C++ program is called *sim* in CSIM. This is the root function for every other process that will run in parallel within the system. As will be described in the next section, this main function becomes a process and coordinates the entire simulation including parameter settings and the shutdown of the whole simulation.

4.2.2 Processes

Processes are the active elements of a CSIM simulation program. In C++ a function is invoked synchronously. The control flow leaves the invoking function and goes to the called function. After the called function has computed its results and finished the return statement the control flow returns with the results to the invoking function. In CSIM one can define asynchrone function calls. When the called function computes the *create* statement, the control flow is split as in the unix *fork* statement. Each instance of every function that computes the *create* statement has its own runtime environment. As operating system processes, a CSIM process can be in one of the four states:

1. Actively computing
2. Ready to begin computing
3. Holding (allowing simulated time to pass)
4. Waiting for an event to happen (or a facility to become available, etc.)

In Figure 4.1 an asynchronous function call in CSIM is described. After the create statement in the client function the control flow is split into one for the client process and one for the sim process.

4.2.3 Facilities

In order to model resources such as CPUs, disk drives or a network, CSIM uses the concept of *facilities*. In the simpliest case, a facility consists of a single server and one queue. Processes are able to use, reserve or hold a facility for a certain amount of time. If there is more than one process accessing a facility these processes are queued into the queue. The order in which queued processes are served depends on the queueing policy that is defined for the facility. One of the following policies are available for facilities:

1. *fcfs* - first come, first serve
2. *inf_srv* - infinite server
3. *lcfs_pr* - last come, first serve
4. *prc_shr* - processor sharing
5. *rnd* - round robin
6. *rnd_pri* - round robin with priority

In addition to the one-queue-one-server architecture, a multiple server can be defined where a process is queued in case none of the servers is available.

4.2.4 Events

To synchronize processes CSIM uses events. An event consists of a state and two queues for processes waiting for the event to occur. A process that executed the waiting statement is queued into the "waiting" queue and a process that has executed the queueing statement is queued into the "queueing" queue. The difference between these two queues is that, if the event occurs, all processes of the waiting queue but only the first process of the queueing queue are allowed to proceed.

4.2.5 Mailboxes

A mailbox is a method which allows processes to communicate asynchroniously. A process can send a message to a mailbox and another process can receive this message from the same mailbox. If the mailbox is empty and a process wants to receive something from the mailbox, it waits until another process sends something to the mailbox. The content of a mailbox is either a pointer to an object, or an integer value.

4.2.6 Data - Reduction and Presentation

During a simulation run, tables or qtables can be used to store status information about processes and queues. Tables record values and print their average, number of entries, varianz minimun and maximum. Qtables store statistical information of queues. This is the average number of processes in the queues or the mean time processes spend in the queue.

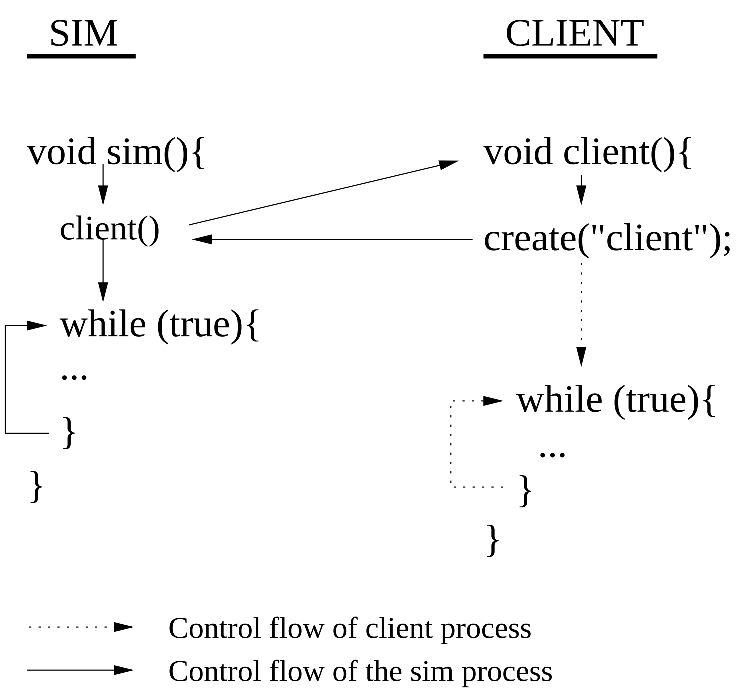

Figure 4.1: *Building Processes in CSIM*

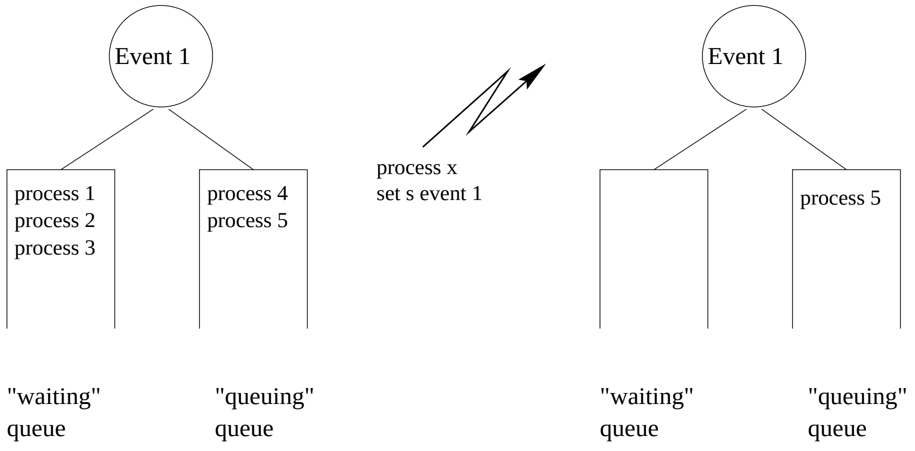

Figure 4.2: *Events in CSIM*

Chapter 5

Predicate Caching

This chapter deals with client-server DBMS-architectures, caching and SQL*Cache in particular.

First client-server DBMS are divided into the two categories *Data Shipping* and *Query Shipping* systems. Section 5.2 gives an overview about SQL*Cache with its different susbsystems on client and server side.

5.1 Client-Server Database Systems

In [11] client-server database systems are categorized according to the unit of interaction among client and server processes. A system may send data requests to the server as queries or as requests for specific data items. The former system type is called a query shipping system and the latter one is referred to as a data shipping system.

5.1.1 Data Shipping Systems

Data shipping systems perform most of the query processing work at client-side. As a result of this, the main DBMS functionality (Application, Query Processing, Object Management, Buffering) is placed at the client.

The client (workstation) sends a page or object request to the server who gets the data item out of the database and sends it to the client. The transaction management is still done on server side. Data shipping systems can be further divided into page server and object server. Figure 5.1 gives an overview over a data shipping system.

5.1.2 Query Shipping Systems

In query shipping systems the client sends a query to the server; the server processes the query and sends back the results. All DBMS functionality is placed at server side. The client side is employed only to display the query results.

5.1.3 Caching in Client-Server Architectures

In [10] caching is defined as "a degenerated and temporal case of replication in which the data requestor receives a read-only copy of data from a server to be held and accessed locally."

Caching can be found in a variety of systems such as registers in processors to avoid memory access or in buffers of file and database systems to avoid disk access, etc. In the case of a page- or object-server system, client caching refers to the ability of clients to retain copies of database pages locally after they were received from the server. Pages are stored dynamically according to the need and activity of the client. There also exists another kind of replication where pieces of the database are statically loaded as replica in different client cites. Static replication is done as part of the database definition process.

In the following sections I will only talk about caching as a dynamic and temporal replication of database objects. The major problems in caching are consistency and cache replacement policy. Consistency is referred to as the content in the database and in every single cache. Especially in distributed systems, the network delay affects the type of algorithm that should to be chosen. The replacement policy refers to the problem of when and what to forget in the cache when a new item must be brought into the cache and there is no free space remaining.

5.2 SQL*Cache - a Predicate Caching System

This section gives a brief overview about SQL*Cache. For a detailed description see [24]. The idea of SQL*Cache was first presented by Julie Basu and Arthur Keller from Stanford University on the Third International Conference on Parallel and Distributed Information Systems [15]. SQL*Cache is a client-side, data-caching scheme for relational databases with a central server and multiple clients. In typical commercial client-server relational databases, caching is done on the server site only. The server buffers frequently accessed disk blocks or pages to avoid disk traffic. Within this schema the assumption is, that clients are low-end workstations not being able to perform queries but only used to display query results.

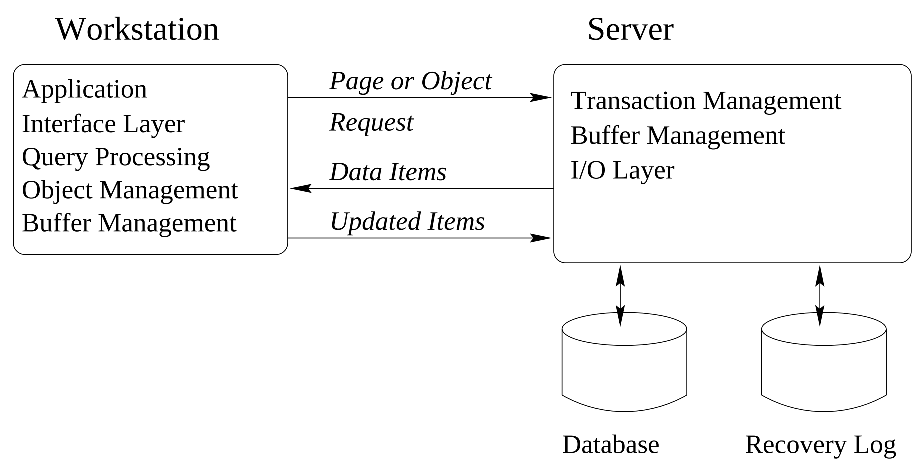

Figure 5.1: *Data Shipping Systems (Page or Object Server)*

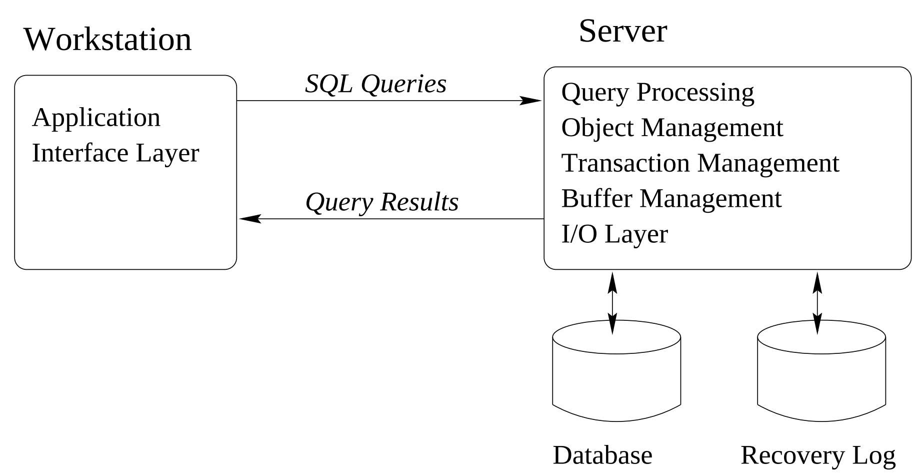

Figure 5.2: *Query Shipping Systems*

According to the general classification of client-server architectures and different caching schema, SQL*Cache can not be seen as a purely *Data* nor a *Query Shipping* system. It sends queries to the server in case it has not cached enough data, but also computes queries on its own cached data if possible. As proposed in [15] and [3], the system can be seen as a object-based caching system. It caches objects rather than pages.

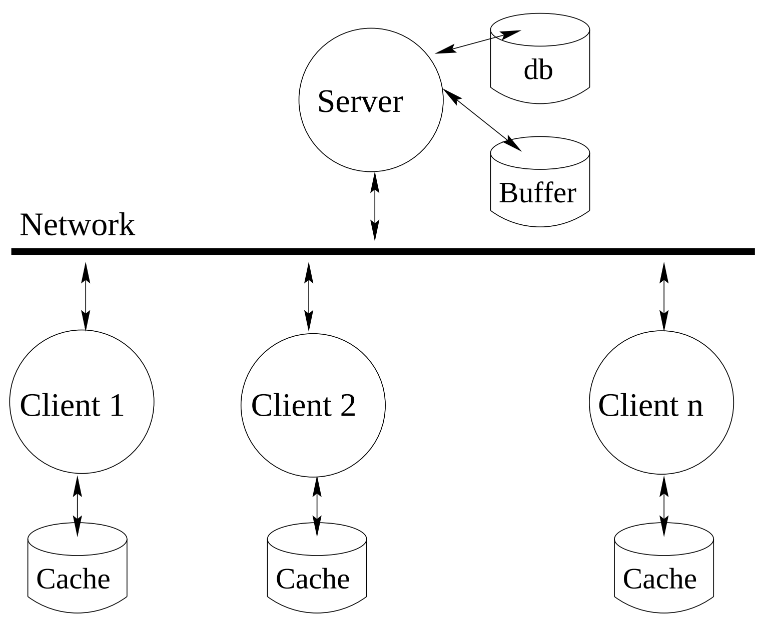

Figure 5.3: *SQL*Cache Client-Server Configuration*

Today the assumption that clients are low-end workstations is no longer true. Because it is increasingly common to find high-end workstations as clients and mainframes or minicomputers as servers. High-end clients are capable of performing intensive computations on their own. In this case the database which is still resident at the server is only used as a remote information source. Increased local functionality and autonomy can lead to less network traffic, faster response time, improved utilization of local computing power and higher throughput. Besides static performance aspects this approach leads also to a better scalability.

Queries executed at server site are used to load the client cache. Both the server and the particular client keep a cache description of the cached data to examine and maintain the contents of the cache. Over time the downloading of tuples from the database will create a certain small view on the database in every client. If a client then determines from its local cache description that a new query can be computed locally using cached data, the server does not need to be contacted immediately. Only in the case of updates at the client site the server is contacted to update the database and notify the clients that may have stored the updated tuples.

In case the query cannot be computed entirely at the client site, the query or only parts of it are sent to the server to compute the results. Each client can treat the cached data as an ordinary relational database and therefore it can have its own locally maintained indexes or access paths to facilitate local query execution.

Local caching as it is presented here involves also cache consistency maintenance and complicates transaction concurrency control. Solutions for these problems are not addressed in this paper but can be found in [15] and [24].

The remainder of this section covers the SQL*Cache architecture with its different subsystems, the *Local Query Execution*, *the Cache Containment Reasoning*, the *Cache Refresh* and the *Space Manager* on client side and the *Notifier* on server side.

5.2.1 The SQL*Cache Architecture

In the SQL*Cache Architecture a central server and multiple clients that are individually connected to the server are considered. Transactions are initiated from the clients, which are high-end workstations to perform the various tasks addressed in this section. The relational database is resident at the server and provides facilities for shared data access. A SQL*Cache server side subsystem is present at the central server and a client-side subsystem exists at each client site. Figure 5.4 gives an overview of the architecture of a one client-server configuration.

5.2.2 Client-Side Subsystem

The main functions of the client-side subsystem are to decide whether a query should be cached, how to reclaim space when the cache is full (cache replacement policy), how to execute queries locally and whether a query can be answered using the cached results. The system can be broken down into the following four major componends:

1. A local SQL engine with an associated data store
2. A cache containment reasoning subsystem
3. An update notification handler for cache refresh operations
4. A space manager for loading and discarding query results

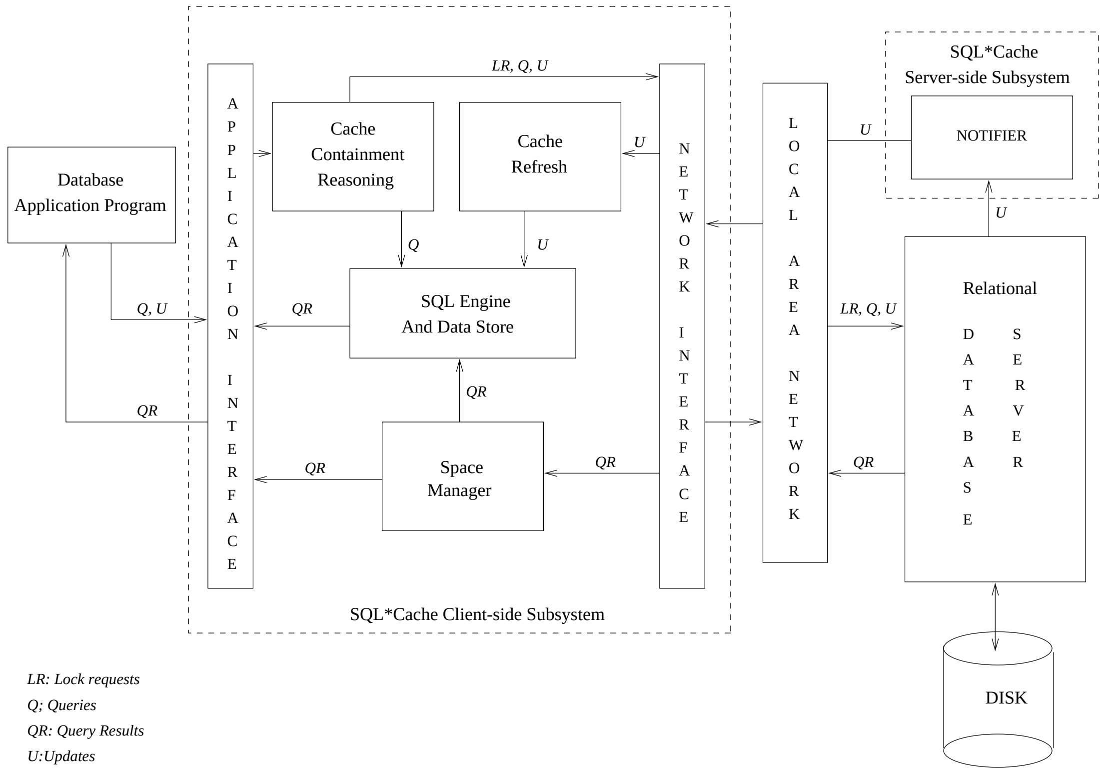

Figure 5.4: *Overview of the SQL*Cache Architecture*

Local Query Evaluation

The *Local Query Evaluation* performs queries on locally cached data. It is a SQL engine with an SQL parser, a query plan optimizer, and a query executer. It uses locally defined indexes on the cached relations. These local indexes may be different from those in place at the server, and may, in particular, be defined on different sets of attributes, depending on data usage and access patterns at a client.

5.3 Cache Containment Reasoning at Client Side

The function of the cache containment reasoning system is to 'conservatively' compare each user query against the current SQL*Cache description, and return a Yes/No answer denoting whether the query result is contained in the cache or not. If the answer is Yes, then the query is submitted to the local SQL engine, which evaluates the query on the cached data; else the query is shipped to the server.

In our simulation, the predicate of each cached query is stored in the SQL*-

Cache description in the form of an interval (or range) on the constrained attribute for the relation. The predicates are organized by relation, and then by the name of the constrained attribute. When a new query is submitted, it is compared with the ranges of all cached queries for that relation and attribute. If the ranges one or more cached queries completely contain the range of the new query, then the cache containment reasoning system returns a Yes result. Otherwise, the query is executed at the server, and the results cached by the client. Caching of new query results may cause previously cached data to be flushed following an LRU replacement policy.

Space Manager

The *Space Manager* has the function of determining whether a new set of tuples will be cached locally. This analysis is based on the anticipated frequency of usage of the query result and on space availability. Predicates and associated tuples may also need to be purged from the cache in order to reclaim space, and the cache description at the client and the server changed accordingly. Reclamation of space is done using a reference counting scheme, so that a cached tuple can be purged only when it is not referred to by any predicate in the client's cache description.

Update Notification Handler

An update handler process at each client propagates updates sent by the notifier to the client cache. First, the cache containment reasoning system is invoked to determine whether the current cache contents are actually affected by the update. This step is required since notification by the server is liberal and involves network delays, and a cached predicate may have been flushed in the meantime due to the LRU replacement of old query results.

If an update notification is found to affect any cached data, the cache is updated with the new values of the tuples. This update may additionally affect the transaction currently executing at the client site. If the update overlaps any predicate or tuples read by the current transaction, then the transaction is aborted and restarted.

5.3.1 Server-Side Subsystem

The server-side subsystem contains a system to inform clients that are affected by updates of other clients. The cache contents of the different clients can overlap, so that changes made by one client can affect other caches and though other clients. The *Notifier* handles these notifications.

Notifier

It is the responsibility of the notifier to keep track of committed updates, and to propagate the updates to the clients based on their SQL*Cache descriptions at the server. The notifier compares the update predicate with cached ones so as to eliminate updates that are irrelevant for a client cache. A notification is generated for a particular client only if the update interval overlaps the range of any query cached by the client.

Our notification scheme follows a “liberal” policy, whereby each SQL*Cache may receive some irrelevant notifications, but is guaranteed to receive all relevant ones. For example, only update interval overlap is checked by the server. Not all tuples in the update interval may have been updated (depending on the write probablity for each tuple); however, the server does not keep track of individual tuples at the clients, and thus only detects possible conflicts at the predicate level.

Chapter 6

Simulation Model

6.1 Logical Queueing Model of the Standard Client-Server

In this section the logical queueing model of the *Standard Client-Server* architecture is defined. The model consists of a single server and a varying number of client workstations, all of which are connected by a network. The number of client workstations in the system is a parameter of the model (see section 6.4). Figure 6.1 gives an overview of the three major parts of the queueing model, namely the clients, the network manager and the server. Although the number of clients is a parameter that could be bigger than one, for brevity in the given model it is depicted as one. Jobs are serviced at all queues with a FIFO discipline.

6.1.1 Client Model

The model of each client consists of a *Workload Generator* (WLG) and an *Application Processor* (APPL). Queries generated by the *Workload Generator* are sent to the *Receive Queue* of the *Network Manager* who sends it to the server. Results received by the *Network Manager* are sent to the *Application Processor*. After he has finished his work, a new query is generated by the *Workload Generator*. What kind of query is generated depends on the selected workload type (see chapter 7, section 7.4.2). The parameters used to specify a client are listed in Table 9.1.

6.1.2 Network Model

It is not in the scope of this thesis to discover the affect of a specific network on the system behavior. Thus we choose a simple model, which has only an input and output FIFO queue to model both directions, to the server and from the server to the clients. The network parameters are its bandwidth, a fixed

Standard Client		
Parameter	**Description**	**Value**
ClientCPU	Instruction rate of client CPU	33 MIPS
SQL*Cache Client		
ClientCPU	Instruction rate of client CPU	33 MIPS
ClientBufferSize	Client buffer size	25% - 50% of the database
BufferTime	Time to get a page out of the buffer	3.5 milliseconds
InstInAppl	Instruction in the client application	200000
ClSelectInst	Instruction for processing a Selection	100
ClProjectInst	Instruction for processing a Projection	1000
ClJoinInst	Instruction for processing a Join	1000
InstAbort	Instruction for abort a transaction in the client	10000
InstCommit	Instruction for commit a transaction in the client	10000

Table 6.1: *Client Parameters*

CPU per-message instruction count plus an additional per byte CPU instruction increment cost. Table 6.2 lists all network parameters. The cost-function for the network-cost is:

$$Networkcost(\#bytes) = \frac{FixMsgInst + PerByteMsgInst * \frac{\#bytes}{4096}}{CPU_{sender}}$$

$$+\frac{\#bytes}{NetTransSpeed} + \frac{FixMsgInst + PerByteMsgInst * \frac{\#bytes}{4096}}{CPU_{receiver}}$$

6.1.3 Server Model

A job submitted at a client passes the network and arrives at the *Ready Queue* of the server, in which it awaits service. The *Multi Programming Level* (MPL) defined in the server, limits the maximum number of jobs currently active and competing for system resources. Until the number of active transactions becomes less than the multi programming level, inactive jobs remain in the *Ready Queue*.

Parameter	Description	Value
NetTransSpeed	bandwidth of the network	$\frac{5Mbits}{sec}$
FixMsgInst	Fixed no. of instruction per message	10000 Instr.
PerByteMsgInst	No. of instructions per msg. byte	5000 Instr. per 4KB page

Table 6.2: *Network Parameters*

Active jobs are queued in the *Concurrency Control Queue* and they are now ready to compete for system resources such as CPU, access to disk, buffer and lock tables.

Within the *Concurrency Control Queue* the jobs try to get all locks which are necessary for the given query. The *Concurrency Control Queue* allows a two level locking mechanism on both tables and records. For a job requesting a lock there are two possibilities to continue depending on whether the lock is granted or not. The cost to get a lock is given in instructions the CPU has to perform (LockInst) devided by the CPU-speed (ServerCPU)

$$\frac{LockInst}{ServerCPU} \leq LockCost_{tuple} \leq 4 * \frac{LockInst}{ServerCPU} + 4 * 10msec.$$

If the lock request is denied, the job is queued in the *Blocked Queue* where its next attempt to get a lock is delayed for a certain period (10 msec) and a counter is increased. If the same lock-request unsuccessfully passes a predefined number of times through the *Blocked Queue*, e.g four times, then a deadlock detection mechanism is triggered. The detection mechanism simply looks for cycles in a wait-for graph that is maintained throughout the processing of the jobs. In case a cycle is found the job with the least amount of processing is aborted and restarted, that is, they release all their locks and inform the client which resents the query rejoining the system's *Ready Queue*. The costs for this operation are defined in the *InstAbort* parameter, i.e., the costs in terms of CPU-instructions.

If the lock is granted and the query needs more locks, it is queued in the *Concurrency Control Queue*. After the last lock is granted for a certain query, it is passed to the *Query Processing Unit*, i.e., the requested pages are taken from the buffer (cache hit) or, cache-fail, it is loaded from the disk into the buffer. In this process three queues are involved, the *Buffer Pool Queue*, CPU-queue and the *Disk Queue*. After a page is taken from the *Buffer Pool* it is passed to the *Processing Queue* (PR) where a certain cost is assumed according to the kind of query.

The server-CPU-cost of the queries can be divided into three categories: SELECTION, PROJECTION or JOIN. These costs are all linear to the number of tuples. For example for Join it is:

$$\frac{JOIN * \#tuple}{ServerCPU}$$

After processing the last page needed to complete a given query, the results are sent to the client by passing them to the *Output Queue* of the *Network Manager*. The client then sends his *Commit*-message to the server. The *Commit*-Process flushes all dirty pages then releases all locks held by that client. The successful completion of this process is sent as a message to the client, which than starts a new query after waiting the *thinktime*.

Table 6.3 lists all server parameters with their possible value or if the parameter might be varied with their range.

Parameter	**Description**	**Value**
ServerCPU	Instruction rate of server CPU	50 MIPS
MulProGra	Multi Programming Grade	$1 \cdots$ *NoClients*
DeadlockInt	Deadlock detection interval	5 times
DiskTime	Maximum disk access time	10 milliseconds
DiskOverheadInst	No of instructions to get a page out of the buffer	5000
LockInst	No. of instructions per lock or unlock	400 instr.
ServerBufSize	Server buffer size	1% $\cdots$ 100% of the *DatabaseSize*
BufferTime	Time to get a page out of the buffer	3.5 milliseconds
RegisterPred	No. of instructions to register or unregister a predicate in the notifier	2000
BlockedTime	Time the lock request is delayed after a blocked request	4
SerSelectInst	Instructions for processing a Selection	100
SerProjectInst	Instructions for processing a Projection	1000
SerJoinInst	Instructions for processing a Join	1000
CompCost	Istructions to compare one predicate	500

Table 6.3: *Server Parameters*

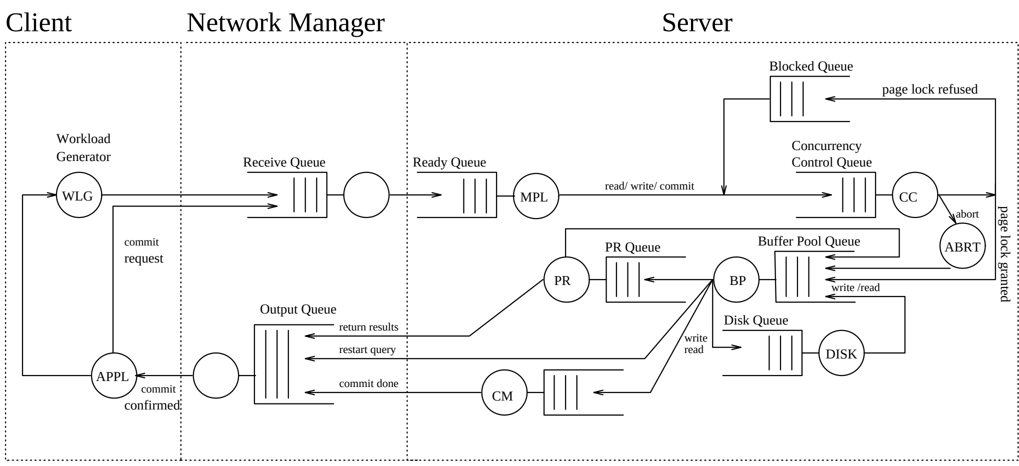

Figure 6.1: *Logical Standard Client-Server Model*

6.2 Logical Queueing Model of SQL*Cache

This section describes the queueing model of *SQL*Cache*, using the *Standard Client-Server Model* as a basis. A description of SQL*Cache was given in chapter **??**. All properties of SQL*Cache which are important for a simulation-based performance evaluation are brought into the logical queueing model. Beneath these properties are the *Notifier* at *Server* side, the *Cache Containment Reasoning* and the *Client Buffer Pool*, called *Client Cache*. Most of the parameters are the same as in the *Standard Client-Server Model*, they are listed in Table 9.1, 6.2 and 6.3. Differences in using these parameters are mentioned where necessary.

Figure 6.2 shows a detailed model of the way the SQL*Cache is mapped into a logical queueing model.

6.2.1 Client Model

In addition to the *Workload Generator* and the *Application Processor* of the *Standard Client-Server Model* the *SQL*Cache Client* contains a *Cache Containment Reasoning* (CCR) to determine whether the query can be processed within the client or if it must be sent to the server.

$$InCache(q^1) = \frac{NumberPredicatesInCache * CompCost}{ClientCPU}$$

[1]q is a SQL query

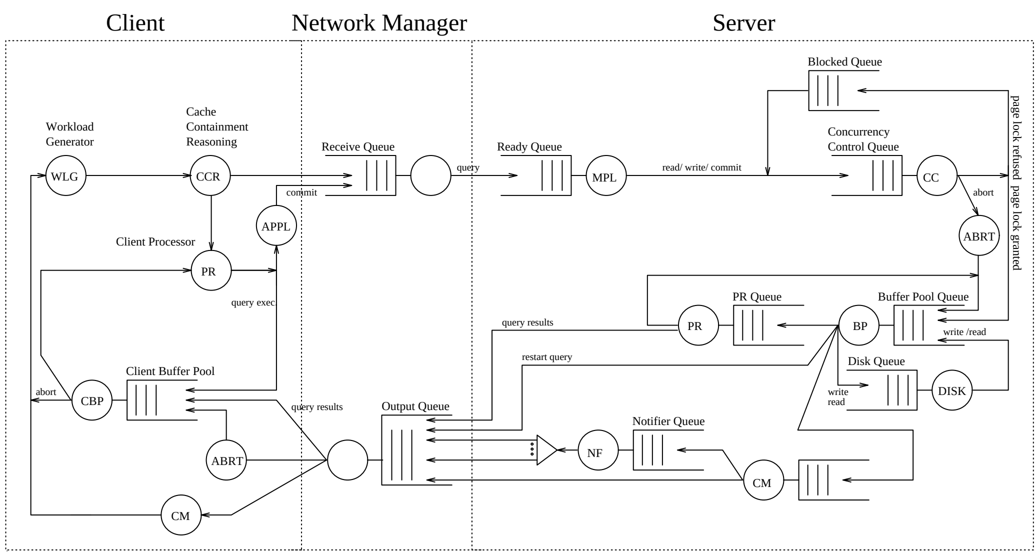

Figure 6.2: *Logical SQL*Cache Model*

It also has a *Client Buffer Pool* (CBP) to store query results locally. The total cost is

$$CacheReplacementPolicy + NoCostforFlushing + CostforUpdate.$$

In order to process queries on locally cached data, a local *SQL Query Engine* (PR), an *Abort Handler* (ARBT) and a *Commit Handler* (CM) were added.

There are two ways of computing a new query generated in the *Workload Generator*. If the results of the new query are a subset of previously computed queries from this client, the query might be computed by using only cached results. If the query results are a superset of previously computed results or if cached data was swapped out of the cache, the query is sent to the server.

The *Cache Containment Reasoning* module decides whether the query can be computed at the client or at the server side. The cost assigned to this decision is relative to the number of queries that are stored in the client.

If the query cannot be computed in the client, it is sent via the network to the server. As mentioned in the description of the *Standard Client-Server*, the server might send the query results or an abort transaction, if there is a deadlock at server side. If the query is executed successfully on the server side, the client updates his buffer with the received results. Now the client is ready for a new query from the *Workload Generator*.

If all necessary tuples are stored within the Client Buffer Pool, the query is sent from the *Cache Containment Reasoning* to the local query engine (PR). Because different clients may compute queries on the same data stored in their cache, an enhanced consistency protocoll must be applied to maintain consistency of this data.

In our simulation model we use a technique that was presented in [30] by Kevin Wilkinson and Marie-Anne Neimat from Hewlett Packard Laboratories. Their algorithms are based on update notifications to every client which is affected by those updates. Notifications are sent after all updates are written into the database. When a server receives a request for commit from a client, it assumes that the client has enforced consistency correctly. The updates are committed at the server side, and all clients affected by them are notified. It is necessary to maintain a cache description of every client cache in the server in order to decide which client is currently affected by a given query. When the client receives a notification it updates its own cache and checks whether the current query is affected by the changes. If it is affected, the transaction is aborted. There are two ways consistency can be lost by the delay in messages transmission. Figure 6.3 and Figure 6.4 show time diagrams of two different problem cases.

1. A client commits by sending a commit request to the server. Until the server receives that commit request, it may still be sending notification messages to the client that could abort the transaction.

2. After a transaction has been accepted for commit by the server, while the deferred updates are being applied to the central database, another transaction can invalidate the read/write set of the committing transaction.

To protect the system against this exposure a handshake is required between the client and the server to ensure that the client has seen the most recent notification. In this simulation it is implemented by assigning a sequential number to every message which is sent to clients. The client sends the number of the last message he received, together with the commit request to the server, which then checks this number against the last message number it sent to this particular client. The last message number must be checked again after all deferred updates have been posted.

6.2.2 Network Model

The network model is exactly the same as in the *Standard Client-Server Model*. See Section 6.1.2 in this chapter.

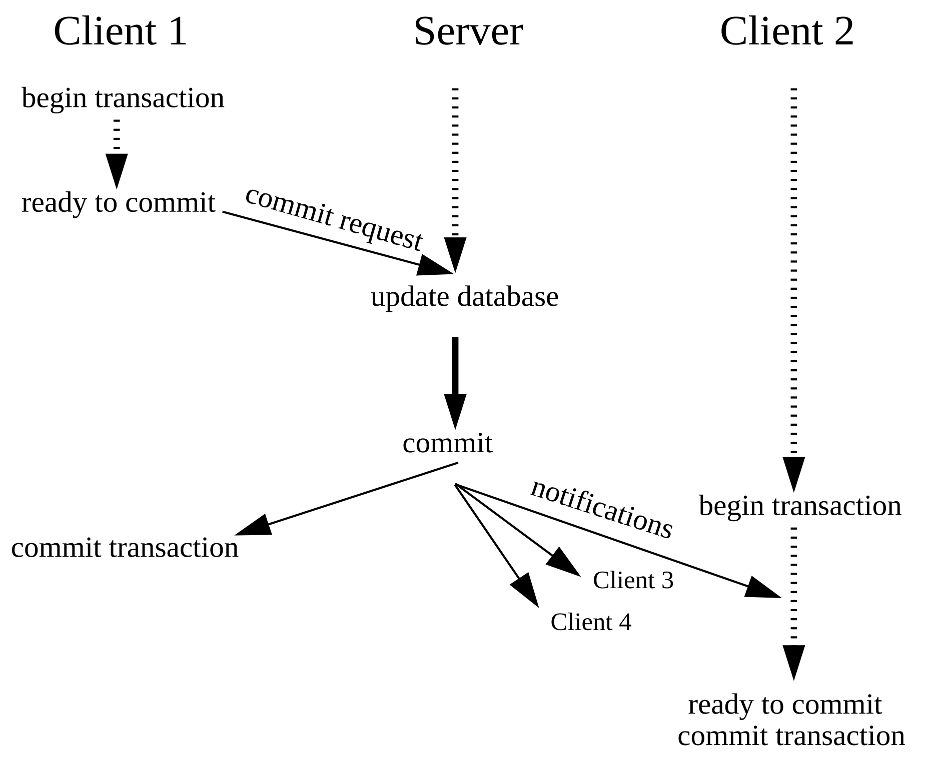

Figure 6.3: *Cache Consistency problem: Case 1*

6.2.3 Server Model

The server model of the *SQL*Cache* carries out two tasks. As in the *Standard Client-Server* model, it must perform queries sent from the clients. Further, it notifies clients if their cache must be updated. The part of the server in charge of performing queries is the same as in the *Standard Client-Server* model. To inform clients to update their buffer we added the notifier. When a query with updates is performed by the server, or a client sends tuples that were updated in a query performed locally in this client, the notifier sends messages with the necessary updates to the clients. To decide which client is affected by this cache-update procedure, we added a cache description of every client in the server. This is basically the same cache description as in the clients. I assume that inconsistencies between these two descriptions can occur because of network delay and a liberal policy. If a query is flushed from a client cache, the messages informing the server of the cache changes can be delayed in the network, and so the two caches are no longer consistent. Because pages might be flushed out of a clients cache, the server may still sent notifications to this client using his "old" cache description. In this case the client receives but ignores the message.

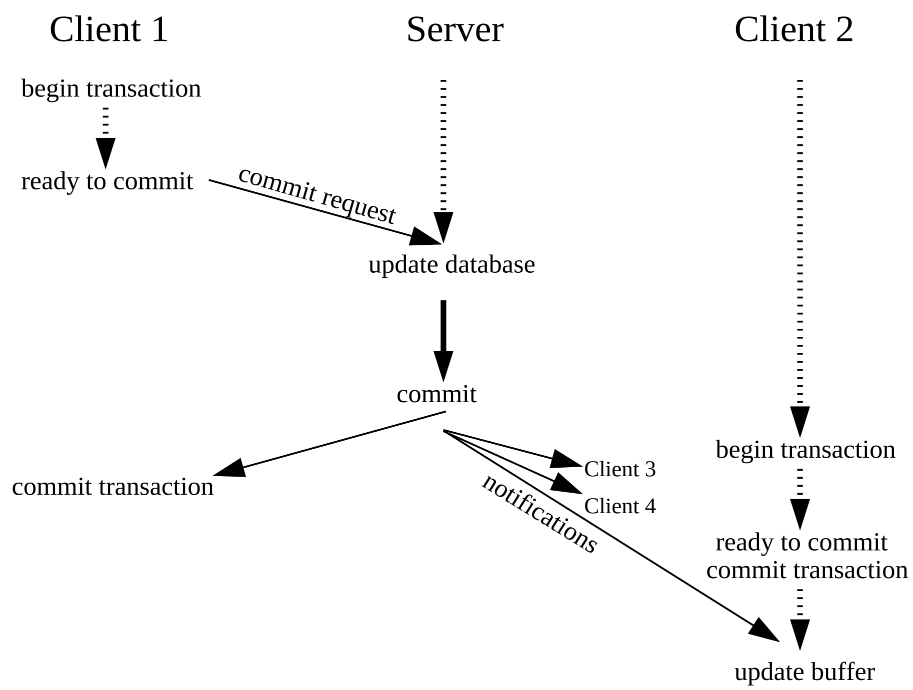

Figure 6.4: *Cache Consistency problem: Case 2*

6.3 Mapping the Logical Queueing Model into a Physical Queueing Model

In this section the mapping of the logical queueing model of both the *Standard Client-Server* and the *SQL*Cache* into a physical model is described. Figure 6.5 shows the overall physical queueing model of both systems. Except for the client side, the two models look exactly the same.

6.3.1 Standard Client Model

The *Standard Client Model*, see Figure 6.6, consists of $1 \cdots n$ server-stations without any queues. Each server-station represents a user accessing one client. There is no contention within one client and therefore no queues are used. Thus the client can be seen as a *Infinite Server* with a delay of the *ThinkTime*.

6.3.2 SQL*Cache Client Model

As in the *Standard Client Model*, see Figure 6.7, there are also $1 \cdots n$ server-stations in the *SQL*Cache Client Model*. In contrast to the *Standard Client* there

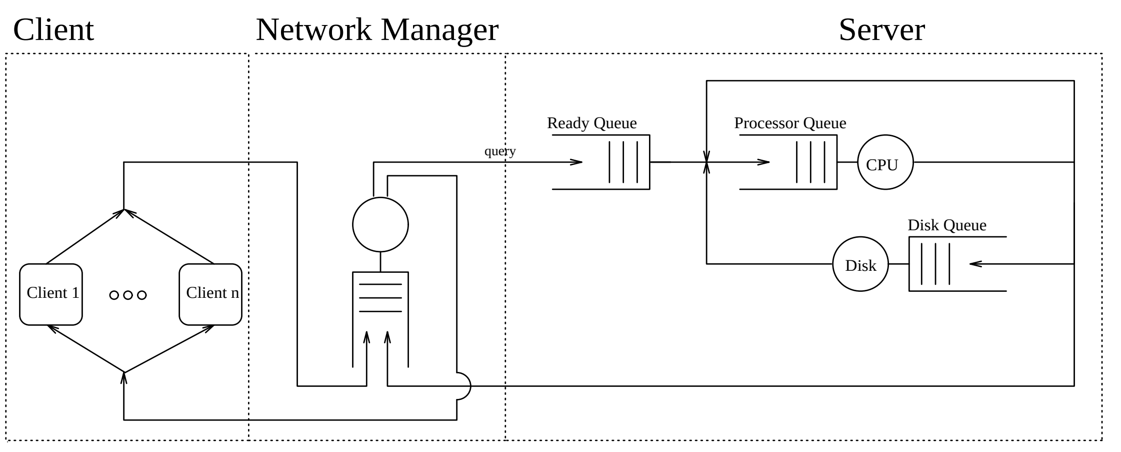

Figure 6.5: *Physical Queuing Model of both the Standard and the SQL*Client-Server Architecture*

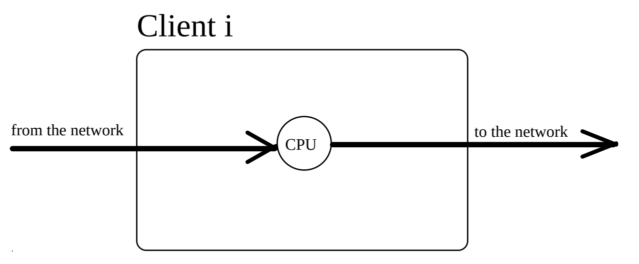

Figure 6.6: *Physical Queueing Model of the Standard Client Architecture*

could be contention in using client resources in the SQL*Cache Client Model. Because of updates triggered by the server's notifier, one task of the system is to update the buffer. On the other hand, the client must work on the query processing. This is modeled with two queues, a CPU and a buffer queue.

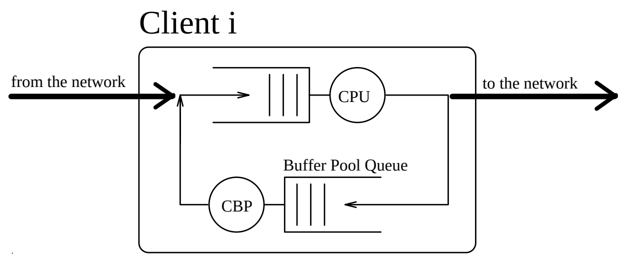

Figure 6.7: *Physical Queueing Model of the SQL*Cache Client Architecture*

6.3.3 Network Model

It is not in the scope of this thesis to analyze affects of various network architectures. Thus a very simple network model is assumed. It consists basically of one queue with a delay that depends on the *NetTransSpeed*. The service discipline function of the queue is FCFS. Furthermore the cost both at sender and receiver site to send a message is taken into consideration. For every message a fixed and a variable cost are assigned to the client and server side CPUs (*FixMsgInst, PerByteMsgInst*). The total cost can be computed as:

$$Networkcost(\#bytes) = \frac{FixMsgInst + PerByteMsgInst * \frac{\#bytes}{4096}}{CPU_{sender}}$$

$$+ \frac{mathit\#bytes}{NetTransSpeed} + \frac{FixMsgInst + PerByteMsgInst * \frac{\#bytes}{4096}}{CPU_{receiver}}$$

6.3.4 Server Model

The *Server Model* consists of three queues of which two have server-stations. These server-stations represent the CPU and the disk facilities, whereas the *Ready Queue* controls the *Multi Programming Level*. Their service discipline functions are all FCFS.

6.4 Other Parameters of the Simulation Model

There are some more system wide parameters, such as the *PageSize* and the *NumClients*. They are used within all parts of the system. Table 6.4 gives an overview over these parameters.

Parameter	**Description**	**Value**
PageSize	Size of a page in bytes	4096
NumClients	Number of Clients connected to the Server	$1 \cdots n$
StandardClientServer	Which model has to be chosen	boolean (1,0)
Tracedriven	If the queries are generated (o) or taken from an input file (1)	boolean (1,0)

Table 6.4: *System Wide Parameters*

Chapter 7

Workload Model

In this chapter the workload model used for the simulation and the measurements is described. Every simulation must be given some sort of workload. This is always a very difficult question to answer, because the workload type strongly influences the simulation results.

There are two methods of defining a workload. On the one hand a trace of a real situation can be taken and used as an input. On the other hand, synthetic benchmarks can be used to generate input to the simulation. The trace in the first solution may be more realistic but less flexible and very hard to obtain. A synthetic benchmark has only a close resemblance to the real situation, but input is easy to produce and different shades can be applied to the simulation.

Therefore, a synthetic benchmark is chosen to analyze the given predicate cache system (SQL*Cache). First, the requirements for analyzing a predicate cache are defined and several benchmarks are examined to choose the one satisfying most of the requirements. These steps have let to the selection of the Wisconsin benchmark, which is subsequently described and enhanced

7.1 Requirements of the Workload Model

Each computer system is typically designed for a few problem domains and may perform poorly on other tasks. Therefore no single metric can measure the performance on all application domains.

In order to measure numeric computations in workstations benchmarks were developed (SPEC-benchmark). These scientific and time sharing benchmarks are inappropriate to evaluate a database system or a transaction processing system. They focus on numeric computations, while the performance of database and transaction processing systems are affected by software algorithms more than by raw hardware speed.

Database and transaction processing systems can be divided further into systems that are excellent at performing simple update intensive transactions, but which may perform poorly on complex queries or vice versa. In [27] the following key criteria for a domain-specific benchmark are defined as:

1. **Relevance:** It must measure the peak performance and price/performance of the system when performing typical operations within that problem domain.
2. **Portability:** It should be easy to implement on many different systems and architectures.
3. **Scaleability:** It should be applicable to small and large computer systems. It should be possible to scale up to larger systems and to parallel computer systems as computer performance and architecture evolve.
4. **Simplicity:** It must be understandable, otherwise it will lack credibility.

To choose the right benchmark, a domain analysis of the system to be examined should be made.

7.2 The Application Domain of SQL*Cache

Since SQL*Cache is not an existing commercial system, it is hard to define its application domain. However the key criteria and the special properties of the system and its behavior can be described. The performance analysis of SQL*Cache is focused on query processing in a multiuser environment where the contention among different users is a key factor. It should be possible to vary the level of contention between clients. Queries could have a simple structure, but they must support associative access on the underlying database.

In order to test the different properties of SQL*Cache in several scenarios, the benchmark should further be scaleable. Among the properties of SQL*Cache are the type of access paths, buffer size both at client and at server side and percentage of updates and query length.

7.3 Different Benchmarks and their Domains

This chapter dicusses various synthetic benchmarks to find an appropriate one for the performance evaluation of SQL*Cache. Among these benchmarks are the

TPC, the Wisconsin, the AS^{3}AP, and the OO1 benchmark, all of which were developed to analyze database or transaction processing systems. Table 7.2 summarizes the results found by discussing the different benchmarks.

7.3.1 TPC BENCHMARK^{TM}

The Transaction Processing Performance Council[1] (TPC) defined several benchmarks in order to analyze on-line transaction processing (OLTP) systems. These are the TPC-A, TPC-B and TPC-C benchmarks. Performance is measured in throughput (tps-transactions per second).

TPC-A and TPC-B originate from the Debit/Credit benchmark, which defines a simple transaction type based on the bank environment. Now they are too simple, since they were developed in the 1980s and the advances in hardware and software technology improved the system performance significantly. Because of this performance improvement the TPC-C benchmark has been developed. It defines a more complex database, as well as a set of five transaction profiles ranging from a simple read query to a complex mini-batch online update. Overall TPC-C is about ten times "heavier" than TPC-A, so the performance metric is transactions per minute (tpm-c).

7.3.2 AS^{3}AP

The AS^{3}AP benchmark is an ANSI SQL Standard Scaleable and Portable Benchmark for relational database systems. It includes both a single and a multiuser environment. The single user test includes utilities for loading and structuring the database, as well as the different query types (selections, simple joins, projections, aggregates, one-tuple updates, and bulk updates. The multiuser test models different database workloads (on-line transaction processing and information retrieval).

The AS^{3}AP benchmark provides an absolute performance metric. For a particular DBMS, the AS^{3}AP benchmark determines an equivalent database size, which is the maximum size of the AS^{3}AP database for which the system is able to perform the designated AS^{3}AP set of single and multiuser test in less than 12 hours.

[1]TPC consists of many companies working in the area of OLTP. For a full list of members see [27]

```
SELECT * FROM TENKTUP1
WHERE unique2 BETWEEN 792 AND 1791
```

Figure 7.1: *1000 tuple result query in SQL on table TENKTUP1.*

7.3.3 OO1 - Engineering Database Benchmark

The OO1 benchmark measures the performance of features essential to engineering applications, such as Computer Aided Software Engineering (CASE) or Computer Aided Design (CAD). OO1 is a redesign of a more primitive benchmark (Sun benchmark) described by Rubenstein, Kubicar and Cattell (see [21]).The OO1 measure includes inserting object-looking update objects and following connections among objects of object oriented, relation, network or hierarchial DBMS. The performance is measured in response time.

7.3.4 The Wisconsin Benchmark

The benchmark now known as the Wisconsin Benchmark was first presented in the Ninth Conference on VERY LARGE DATA BASES 1983 by Dina Bitton, David J. DeWitt and Carolyn Turbyfill [4]. Since than it has become a very popular and widely used benchmark in the database area. In contrast to other benchmarks like the Debit/Credit or the Datamation benchmark the Wisconsin benchmark reduces the system to just one number, namely the transactions per second (tps).

In this section an overview about the benchmark is given. These are the relations, the queries, scalability and the framework in which it has to be used. Enhancements making the Wisconsin benchmark applicable in a multiuser environment are then pointed out.

Relations

The benchmark consist of three tables (TENKTUP1, TENKTUP2, ONEKTUP) synthetically filled with data. Therefore it is very easy to create a query with a certain number of results. Table 7.1 shows the specification of the scaleable Wisconsin benchmark relation and Figure 7.1 shows a SQL query that has exactly 1000 result tuples.

Attribute Name	Range of Values	Order	Comment
unique1	0-(MAXTUPLES-1)	random	unique, random order
unique2	0-(MAXTUPLES-1)	sequential	unique, sequential
two	0-1	random	(unique1 mod 2)
four	0-3	random	(unique1 mod 4)
ten	0-9	random	(unique1 mod 10)
twenty	0-19	random	(unique1 mod 20)
onePercent	0-99	random	(unique1 mod 100)
tenPercent	0-9	random	(unique1 mod 10)
twentyPercent	0-4	random	(unique1 mod 5)
fiftyPercent	0-1	random	(unique1 mod 2)
unique3	0-(MAXTUPLES-1)	sequential	unique1
evenOnePercent	0,2,3,...,198	random	(onePercent*2)
oddOnePercent	1,3,5,...,199	random	(onePercent*2)+1
stringu1	-	random	candidate key
stringu2	-	random	candidate key
string4	-	cyclic	-

Table 7.1: *Attribute Specification of Scaleable Wisconsin Benchmark Relation*

Queries

The Wisconsin Benchmark measures the performance of all the basic relational operations including:

1. selection with different selectivity factors
2. projections with different percentages of duplicate attributes
3. single and multiple joins
4. simple aggregates and aggregate functions
5. append, delete and modify

For a detailed description of all 32 queries see Appendix B. A simple SELECT query can be found in Figure 7.1. In section 7.4 several queries which are used in the experiments of the simulation model are described in detail.

Scaleability

The Wisconsin benchmark can and should be customized for every single system measured by it. On the one hand one can scale the size of the database. The content of every field depends on the total number of entries in that database. On the other hand because of the simple structure of the tables one can easily add or change certain queries. This can be seen in section 7.4 where the enhanced Wisconsin benchmark for a multi user environment is described.

Framework

In order to obtain accurate results several rules should be followed. First, the database should be five times larger than the main memory buffer space available for the DBMS. Second, for each query a certain number of equivalent queries should be run on the system. Ten query-variations for selection, update and delete and four for aggregate and projection test are sufficient. For a detailed description of the rules see [4].

Restrictions of the Wisconsin Benchmark

There are several facts which the benchmark does not take into consideration. Most importantly no test of concurrency control is done. Also no bulk updates take place and lastly no recovery systems of the database system are tested. Some of these limitations, namely the concurrency test, are eliminated in the next section and bulk updates are modeled as a parameter (thinktime) in the general workload model. The simulation model does not consider any recovery systems of the DBMS.

7.3.5 Benchmark Comparison

Table 7.2 shows that only two of the benchmarks described above can be used in the performance analysis of SQL*Cache, namely the AS^{3}AP and the Wisconsin. Both have one deficit in the requirement matrix, but both benchmarks can be enhanced so that they meet all requirements. For the following performance analysis the Wisconsin benchmark is chosen. It appears to be easier and more practicable to put the Wisconsin benchmark into a multiuser environment than to support full associative access within the AS^{3}AP benchmark. The wide acceptance and popularity of the Wisconsin benchmark make it the better choice.

7.4 Customization of the WISCONSIN Benchmark

The workload model that is used in the following performance analysis is based on the Wisconsin benchmark. The queries, the databases and the framework

	TPC-X	**AS^{3}AP**	**OO1**	**Wisconsin**
Multiuser Environment	$\surd$	$\surd$		
Query Processing		$\surd$	$\surd$	$\surd$
Scaleable	$\surd$	$\surd$	$\surd$	$\surd$
Associative Queries				$\surd$
Varying Updates		$\surd$		$\surd$
Query Length	$\surd$	$\surd$		$\surd$
Different Access Paths		$\surd$		$\surd$

Table 7.2: *Comparison of different Benchmarks*

are taken from the Wisconsin benchmark. Features that were not covered by it (see section 7.1) were added, like contention between different clients and the bulk updates and others like the workload type. Table 7.3 shows all workload parameters.

7.4.1 Bulk Arrivals

The number of queries that were received by the server is either determined by the query response time or by the thinking time[2] (*ThinkTime*) of each client. If one presumes that the system has a more or less constant query response time after the system is in a steady state, then *ThinkTime* becomes a very important parameter to influence the amount of queries per second.

7.4.2 Contention

In the examined systems many clients are competing for the same data of the database. In order to model this contention among clients the parameter *workloadtype* is used. Workloadtype can be:

1. HotCold
2. Private
3. HiCon
4. Uniform

In each of these workload types the database is divided into a hot and cold part. From the hot part the most tuples are taken. The parts are defined by setting borders in terms of attribute values of the field *unique1* and *unique2*.

[2]thinking time is the time between two queries.

The parameter *HotRangeRatio* defines the ratio between the hot and the cold part. For example a *HotRangeRatio* of 60% of a 10.000 tuple table means that records with *unique1* in $[0 \cdots 5999]$ are in the hot part and $[6000 \cdots 9999]$ are in the cold part. The Parameter *HotAccessProb* then defines the probability of a query taking place in the hot or the cold part. In order to model different environments in which a DBMS system can be employed, the different workload types are defined as follows.

Workloadtype: HotCold

In the *HotCold* workload type every single client defines its own hot region. This is done by taking into consideration *HotRangeRatio* and *ClientId*. The table is divided into two parts relating to the *HotRangeRatio*. First part ("hot part") goes from attribute value

$$0 \cdots (HotRangeRatio * tablesize - 1)$$

further called *HotBound*. The hot part for client c then is defined as

$$LowerHotBound_c = \frac{HotRangeRatio * tablesize}{NumberClients * (c-1)}$$

$$\cdots$$

$$UpperHotBound_c = \frac{HotRangeRatio * tablesize}{NumberClients * (c-1)} + (HotRangeRatio * tablesize) - 1.$$

The remainder of the table represents the cold part. Though the hot region of one client overlaps with the cold region of all other clients, the hot regions never overlap. The cold part is the same for every client.

Workloadtype: Private

In the *Private* workload type the hot range of a client is defined as in the previous one, but the cold part is defined differently. The cold region is defined as the non hot region of all clients. If the hot region goes from

$$0 \cdots (HotRangeRatio * tablesize) - 1,$$

the cold region goes from

$$(HotRangeRatio * tablesize) \cdots tablesize.$$

It is guaranteed that a cold region does not overlap with a hot region of any client. This workload presents a typical engineering design application.

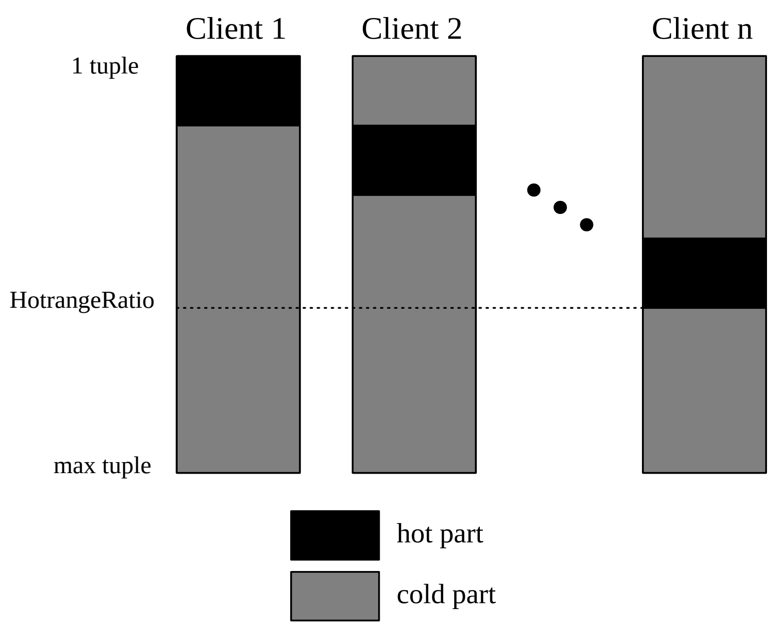

Figure 7.2: *Workloadtype: HotCold*

Workloadtype: HiCon

In the *HiCon* workload the hot region for every single client is the same, whereby the hot and cold bounds are defined as in the previous types. This workload type represents the highest contention among all other types.

Workloadtype: Uniform

The *Uniform* does not have specific hot or cold regions. The queries are directed uniformly to all attribute values.

7.4.3 Read/Write Probability

For this simulation model it is important to model updates as well as select queries. Although the Wisconsin benchmark provides a wide variety of queries, it defines just a few update queries. These update queries basically support updates on one record.

In order to support different update queries and above all range update queries the parameters *HotQueryWriteprob*, *ColdQueryWriteprob*, *HotTupleWriteprob* and *ColdTupleWriteprob* were added. These parameters support a two level read/write probability specification both for tables and objects. First it is specified if the query taken from the query pool is used just to read or to read and write (*HotQueryWriteprob*,*ColdQueryWriteprob*). In order to give the write prob-

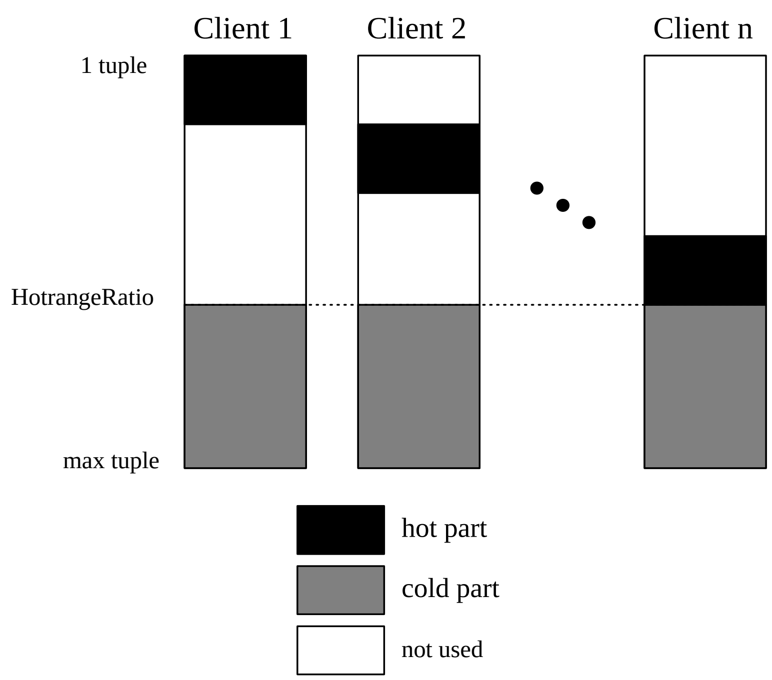

Figure 7.3: *Workloadtype: Private*

ability there are two parameters, both for the hot and the cold part. This will influence the kind of lock that is needed on either the query or the tuple and therefore the waiting time for these locks. In addition, the real update on the tuples in specified by *HotTupleWriteprob* and *ColdTupleWriteprob*. This affects the disk traffic because on query commit there will be writes for dirty pages.

7.4.4 Further Enhancements

Parameters RQ_1 to RQ_{17} were added to define the set of queries used in an experiment. A 0 value in RQ_n means that the corresponding query will never be used by a client, a value larger than 0 indicates the weight of that specific query. The probability of choosing query i is

$$p_i = \frac{RQ_i}{\sum_{q=1}^{17} RQ_q}.$$

This is a useful feature because it can be used to affect the query length. By choosing more or fewer joins (two or three table joins) the number of query results can be increased or reduced.

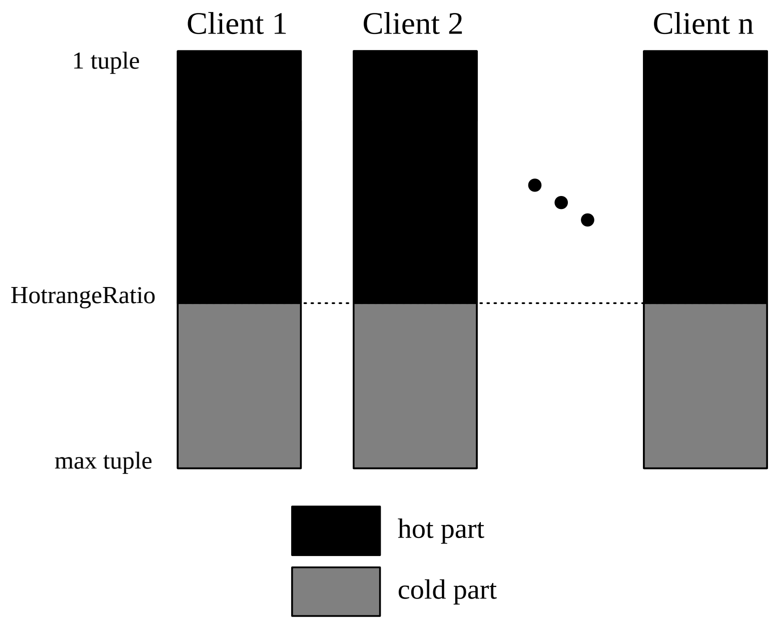

Figure 7.4: *Workloadtype: HiCon*

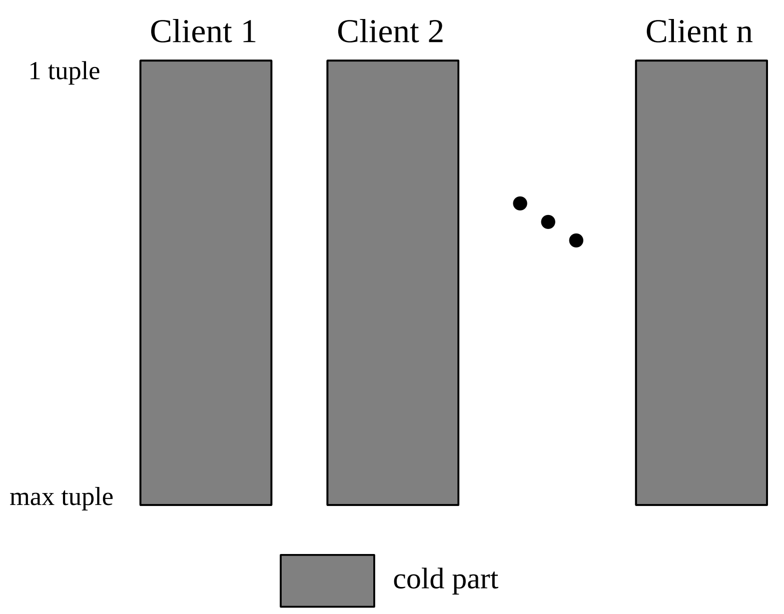

Figure 7.5: *Workloadtype: Uniform*

Parametername	**Description**	**Value**
ThinkTime	User Thinktime between two transactions	0 .. 60 sec.
NoQueries	No of queries to be computed by each client	1 ... 200
QueryLength Percentage	Querylength in tuples as a percentage of the relation size	0.02% ... 1%
RecordSize	Record size in bytes	230 bytes
SizeOfQueryText	Size of the SQL query text	300
TENKTUP1.size	Size of the relation TENKTUP1	10.000/500.000 tuples
TENKTUP2.size	Size of the relation TENKTUP2	10.000/500.000 tuples
ONEKTUP.size	Size of the relation ONEKTUP	1.000/50.000 tuples
HotAccessProb	Probability of an client to access his hotpart of the table	0 .. 100%
HotRangeRatio	Ratio between the hot and the cold part of each table	0 .. 100%
RQ_1 .. RQ_{17}	Usage percentage of query 1 .. 17	0..100%
Workloadtype	The workload type used to define the hot/cold parts of the table	HotCold, Private, HiCon, Uniform
HotQueryWriteprob	Probability of a query being "selected for update" accessing the hot part of the relation	0..100%
ColdQueryWriteprob	Probability of a query being "selected for update" accessing the cold part of the relation	0..100%
HotTupleWriteprob	Probability of a tuple being updated accessing the hot part of the relation	0..100%
ColdTupleWriteprob	Probability of a tuple being updated accessing the cold part of the relation	0..100%

Table 7.3: *Full List of Workload Parameters*

Work-load-type	**Hotbound for client c**	**Coldbound for every client**
HotCold	$\frac{HotRangeRatio * tablesize}{NumberClients} * (c-1)$ $\ldots$ $\frac{HotRangeRatio * tablesize}{NumberClients} * (c-1)$ $+(\frac{HotRangeRatio * tablesize)}{NumberClients} - 1$	rest of the table
Private	$\frac{HotRangeRatio * tablesize}{NumberClients} * (c-1)$ $\ldots$ $\frac{HotRangeRatio * tablesize}{NumberClients} * (c-1)$ $+(\frac{HotRangeRatio * tablesize}{NumberClients} - 1$	$(HotRangeRatio * tablesize)$ $\ldots$ $tablesize$
HiCon	$0 \cdots HotRangeRatio * tablesize - 1$	rest of the database
Uniform	$0 \cdots tabelsize$	-

Table 7.4: *Hot- and Coldbound Definitions of the Different Workloadtypes*

```
UPDATE TENKTUP1
SET unique1 = 10001 WHERE unique1 = 1491
```

Figure 7.6: *Update of an Indexed Nonkey Attribute (query no. 32 of the Wisconsin benchmark).*

Chapter 8

Implementation of the Simulation Model in CSIM

This chapter deals with the implementation both the standard client-server and the SQL*Cache simulator. Every CSIM program use the feature of facilities to define systems recources. The first section shows which facilities are defined and the following section give a more detailed description of the implementation.

8.1 System Resources

This section gives an overview about the system resources; those at the clients sites, the server site and the network between all these sites. The recources of a system are called facilities in CSIM. Theses facilities are the active part of the simulator, that is that there time can ellapse. The system consists of one server and multiple clients connected by a network. The following sections will describe these facilities in detail.

8.1.1 Client - Processor

Each client is assumed to be a high-end one processor-, no disk-workstation with enough main memory to store parts of the database and a processor to run a SQL-engine to execute queries. Thus each client has its own facility *ClientProc*. It is a single-server-single-queue facility; facilities are described in Section 4.2.3. The queue is organized with the first-come-first-serve-policy (fcfs).

The time a processor is hold by a process depends on the activity that is executed in the process. In Table 9.1 all activities with the corresponding number of instruction are listed. $\frac{\#Instruction}{Processor speed}$ is the actual time the processor is used.

8.1.2 Server - Processor

The server is assumed to be a single processor-, single disk-workstation with a buffer to cache data-pages and index-pages. The processor is implemented as a single-server-single-queue facility with the first-come-first-serve-policy (fcfs). As in the description of the client processor mentioned, the time of a processor depends on the activities that he must compute. There is a variety of activities on the server side; prepare messages to send or receive over the network, to read pages from disk, to get or release locks end find deadlocks (see Table 6.3). These activities are given in instructions. The time a processor is used can be computed as $\frac{\#Instruction}{Processor speed}$.

8.1.3 Server Disk

The disk is modeled as a two level memory hierarchy. On the first level is the physical disk and on the second level is a main memory buffer. Figure 8.1 shows the configuration of the disk as a queueing model.

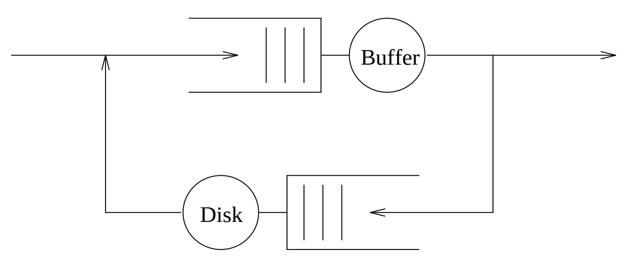

Figure 8.1: *Queueing Model of the Disk-Main-Memory Buffer Interaction*

A page request is first sent to the buffer. The buffer manager then checks whether the page is in the buffer. A CPU-cost is assigned to get a page out of the buffer. If the page is not previously stored into the buffer, the buffer-manager sends a request for this page to the disk. There it is fetched and stored into the buffer.

The raw disk is implemented as a single-server-single-queue facility with a first-come-first-serve (fcfs) policy. If there is no space left to store new pages a victim under all pages currently stored in the buffer, must be chosen. I implemented a last-recently-used (LRU) policy to choose a page-victim that is then flushed to disk in case the dirty bit was set.

8.1.4 Network

I assume a relatively simple network because it is not in the scope of this performance analysis to emphasize the network issues. The network consists of a single-server-single-queue facility with a first-come-first-serve policy. The time the network is used depends on the number of bytes that are sent over the network. I assume that the network has a certain bandwith (see Table 6.2) and the cost is assigned relatively to the number of bytes sent over the network. This is $\frac{\#Bytes}{Bandwith}$. In addition to this cost there is a cost at client and server side to prepare messages to be sent (see section 8.1.1 and section 8.1.2).

8.2 Standard Client-Server Implementation

The *Standard Client-Server* consists of a single server plus a varying number of client workstations all of which are connected via a network to the server, see also Figure 6.1. Queries are generated on client side and sent to the server. The query execution can be described in the following seven steps of which only the first is done on the client side:

1. Query Generation
2. Sending the Querytext to the Server
3. Receiving Execution Permission
4. Getting Read/Write Locks
5. Executing Query
6. Committing Query
7. Sending Results to Client

In the following section I will give a brief overview on how I implemented the seven steps in CSIM. In section 8.2.5 I show which processes I used within the system.

8.2.1 Query Generation

According to the workload parameters (see section 7 table 7.3) each client creates his own queries. Within the WISCONSIN benchmark there are 32 different queries defined (SELECT, single tuple SELECT, 2 table JOIN, 3 table JOIN, PROJECTION, different AGGREGATION functions, single tuple UPDATE and single tuple DELETE). For the simulation I implemented the first 17 variations. The remaining queries cover DBMS testings that are not of interest in this analysis. Within the firsts 17 queries there are four different types of queries:

1. **1%-10% SELECT**:

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

2. **Single Tuple- SELECT**:

```
INSERT INTO  TMP
SELECT * FROM table
WHERE attribute = lower;
```

3. **2 Table Join**:

```
INSERT INTO  TMP
SELECT * FROM table1, table2
WHERE table1.joinattribute = table2.joinattribute
AND   attribute BETWEEN lower AND upper;
```

4. **3 Table Join**:

```
INSERT INTO  TMP
SELECT * FROM table1, table2, table3
WHERE table1.joinattribute = table2.joinattribute
AND   table2.joinattribute = table3.joinattribute
AND   attribute BETWEEN lower AND upper;
```

When a query is created the following information is stored: which type of query it is, which client has created it and at which time[1], the attribute on which the filter is applied, its lower and upper bound on that attribute and status information for query creation.

8.2.2 Sending the Querytext/Queryresults over the Network

Sending a message envolves always three facilities. First the message is prepared on the sender side using the processor facility. Then the facility "network" is used to transport the message and finally at the receiver side the processor is used to prepare the message for further computation.

[1] simulation time, not real time

To send the querytext I assume a fix number of bytes regardless if it is a simple SELECT or a 3 table JOIN query. This is a reasonable simplification since the differences are about 100 bytes. Sending the query results instead takes into account the number of result tuples which can vary enormously.

8.2.3 Receiving Execution Permission

The number of processes that can be active at server side is limited to the multi programming level (MPL). The process carrying a query to execute on the server (see also section 4.2.4) checks the current number of processes currently active at the server side (NoQueriesInServer[2]). If the maximum number of processes active at the server, a new process is queued in the queueing-queue of the event called *ServerReady*. Another process that is active and is about to leave the server sets the event. Only the first process that is queued is released.

8.2.4 Getting Read/Write Locks

Before a process can actually access tuples from the database it must get read/-write locks on these tuples (two phase locking). Locks are provided on two levels of granularity, on tuples or on an entire relation. Which granularity is choosen depends on the access path that is provided on the field on which the filter condition is applied. If there is no index provided a full range scan must be done. In this case a lock in the entire relation is requested. If there is an index on the field only tuple locks are necessary.

If a request for a lock is denied the process that requested the lock is blocked for one second hoping that the lock is released within this second. A process can be blocked several times for the same lock. After four times the lock-manager is invoked to check whether there is a deadlock. The lock-manager choose the calling process as a victim to abort if there is a deadlock.

After the query execution is done (at commit-time) all locks are released at once (strict two phase locking), see [9], [22], [23], [1], [29].

8.2.5 Processes of the Standard Client-Server Implementation

We assume only one query per client transaction. A server call to compute a query can then be seen as a synchronous call although in reality this is not the case. But with the assumption of one query per transaction and no other activity

[2]to protect NoQueriesInServer against multiple access it is implemented as a semaphore

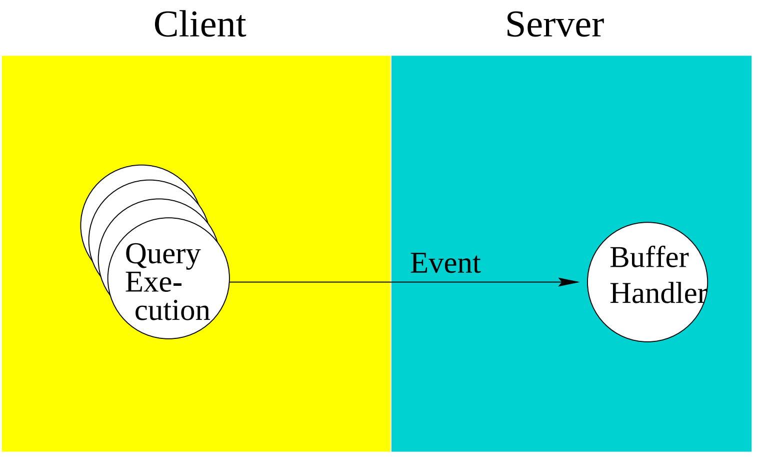

Figure 8.2: *Processes in the Standard Client-Server Architecture*

on client site it is reasonable. Each client is simplified as one process. Instead of implementing the server as a process I use the control flow of the client to envoke functions as for example the query execution. I could have implemented the server as a process of its own, but then I had to deal with all different query execution calls from different clients, while the client control flow waits for the query results. Figure 8.2 gives a brief overview about the process structure of the standard client-server architecture. I took advantage of two features of CSIM.

1. **Different processes can share the same function, but have different work environments:**
   Because within a CSIM program functions can become processes there exists more than one control flow parallely. Every process that calls a function creates a new thread and therefore a new function environment (multi processing, multi threading). One has to take into account that a process cannot be blocked when it does “normal” function calls, but only if the process uses a facility, waits for an event to happen or his mailbox to be filled. It is important if shared memory is used by different processes. This leads to the simplification, that not every shared memory has to be protected by metaphors.

2. **Facilities can hold processes in their queue:**
   Does a process use a facility like the disk, it might be blocked, because the facility is used by another process. This feature and the one mentioned previously allow the implementation of the *Standard Client-Server* with only one process.

8.3 SQL*Cache Client-Server Model

As the standard client-server, the SQL*Server must provide facilities to execute queries. The caching on client side enables the system to execute queries on cached data, too. The steps a query execution takes are:

1. Query Generation
2. Evaluation whether the new query is a subset of previously executed queries
3. In case the new query is a subset
   (a) Query Execution at client side
   (b) In case of updates to cached data: server update
4. In case the new query is not a subset
   (a) Query Execution at the server
   (b) Updating Client Cache & and Client Cache Description
   (c) Updating Client Cache Description at the Server

The following section will give a brief overview on how I implemented these steps in CSIM. The query generation is the same as in the standard client-server model. Similar to the standard client-server model is also the query execution at server side.

8.3.1 Query Execution at Client Side

If in the previous step the cache containment reasoning decides to execute the query at the client side, the local SQL engine executes the query regarding the execution plan of the ORACLE client-server DBMS. After one tuple is fetched from the buffer the system checks if there has been any notifications from the server that affects the current query execution. If there exists such a notification from the server, it means that the cache is invalid and that the query execution must be aborted and be started from the beginning. Until there are no notifications that affects the query the query execution continues and in case of updates in the query execution a list of updated tuples is built. This list will be sent to the server in order to update the database. After the server commited the transaction the client makes sure that he received the last recently notification from the server. If he received all notifications he commits the transaction, too.

8.3.2 Server Update

After the client executed the query on the cached data and before the client can commit the transaction it must send the updated tuples to the server. To make the changes visible to other clients whose cache may be invalid by theses updates, the server sends notifications to all clients who are affected. The update of the database is done with the same code that executes a regular SQL-query. The only difference is that the tuples are taken from the list that was built by the client instead of executing the whole query.

8.3.3 Query Execution at the server

If the cached data is not sufficient to execute the new query the query is shipped to the server and executed there. This is done in the same code as in the standard client-server. Before the query results are sent back to the client, the server updates the cache description it stored for that particular client. On the other hand when the results come back to the client it updates the cache description and maybe he flushes some pages. Because of that the changes the client made to his own cache description must be sent to the server to make sure the cache descriptions are consistent.

8.4 Some Figures regarding the Implementation

The simulator is written in CSIM, a C++ based simulation language (see Section 4.2). The code contains of 5174 lines of code. Table 8.1 shows the lines of code regarding to the different modules. The numbers are from the unix command wc and contain also comment lines.

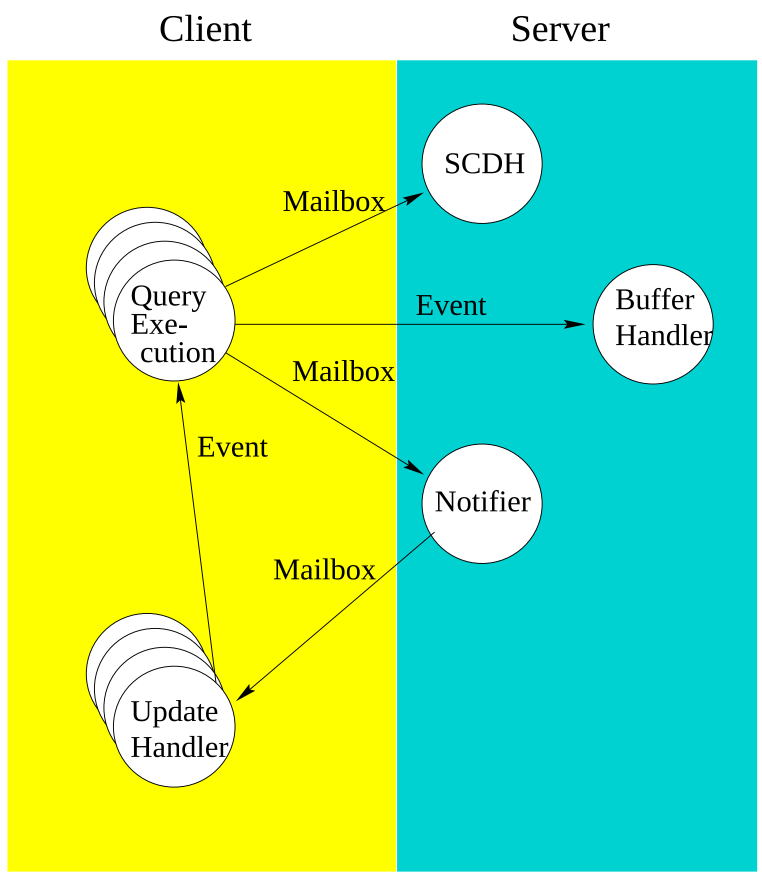

Figure 8.3: *Processes of the SQL*Client-Server Architecture*

Module	**Number of Lines**	**Programmer**
simh.cpp	519	Meikel
simh.h	189	Meikel
simh_buffer.cpp	599	Julie
simh_buffer.h	119	Julie
simh_cache.cpp	520	Julie
simh_cache.h	33	Julie
simh_ccm.cpp	269	Meikel
simh_datastore.cpp	279	Meikel
simh_datastore.h	41	Meikel
simh_general_fkts.cpp	146	Meikel
simh_interval.cpp	87	Julie
simh_interval.h	26	Julie
simh_locktable.cpp	272	Meikel
simh_network.cpp	50	Meikel
simh_network.h	14	Meikel
simh_pred.cpp	59	Julie
simh_pred.h	32	Julie
simh_query.cpp	427	Meikel
simh_query.h	142	Meikel
simh_read_parameter.cpp	274	Meikel
simh_semaphore.cpp	33	Meikel
simh_semaphore.h	15	Meikel
simh_server.cpp	398	Meikel
simh_server.h	28	Meikel
simh_trace.cpp	198	Meikel
simh_trace.h	62	Meikel
simh_transaction.cpp	73	Meikel
simh_wl_generator.cpp	271	Meikel
total	**5174**	

Table 8.1: *Lines of Code in the different Modules*

Chapter 9

Verification of the Standard Client-Server Model

There exist many implementations of the standard client-server DBMS architecture[1] from such companies as ORACLE, Informix or Sybase. Thus, the standard client-server model can be compared directly to a real system. In order to compare the two systems directly, a performance index, which is one of the output parameters must be defined (see Section 3.1.3). The performance index may consist of one output parameter, or a combination of output parameters. To both sytems the same input parameters are given. The behavior is then compared by confronting the performance indices. It will not be possible to retrieve exactly the same performance results from the simulator and the real system. Therefore, a tolerance range must be defined. A tolerable deviation between performance figures from the simulator and the real system depends on how detailed the system is modeled. In general a range from 5% to 10% in assumed. Our simulator models the system in detail. Therefore, the simulator's performance figures must be within ±6% of those obtained by the real system. This credibility is sufficent to our model since the goal of this performance analysis is not to plan a configuration of a new system, but to compare the behavior of SQL*Cache to a standard client-server. The standard client-server is also implemented as a simulator.

In this validation two performance indices are defined: *Query Response Time* and *Disk I/O Operations*. As input parameters (workload), a subset of the possible workloads defined in chapter 7 is depicted.

The next section will define these workloads. Section 9.2 will give an overview of the commercial system while Section 9.3 describes its measurements. The last two sections of this chapter will then present the simulation experiments and the comparison to the measurements made in the previous section.

[1]for a definition see Section 6

9.1 Workload

This section defines the workload that is used to measure the real system and simulates it by the developed simulator. As was demostrated in Section 7, the workload model for this performance evaluation consists of queries taken from the Wisconsin benchmark. In order to keep the validation as clear as possible, four different types of queries are picted from the Wisconsin query pool, namely Query 4 ,6 ,15 ,17. The first two queries are simple *SELECT*-queries on one table. On unique2 a clustered index and on unique1 a non clustered index is defined. Query 4 uses a clustered index[2], while query 6 uses an unclustered index. Query 15 and 17 are join queries on 2 and 3 tables, respectively. The definitions of these queries are:

1. **Query 4**:

```
INSERT INTO  TMP
SELECT * FROM TENKTUP1
WHERE unique2 BETWEEN lower AND upper;
```

2. **Query 6**:

```
INSERT INTO  TMP
SELECT * FROM TENKTUP1
WHERE unique1 BETWEEN lower AND upper;
```

3. **Query 15**:

```
INSERT INTO  TMP
SELECT * FROM TENKTUP1, TENKTUP2
WHERE (TENKTUP1.unique1 = TENKTUP2.unique1)
AND   TENKTUP1.unique1  BETWEEN lower AND upper;
```

4. **Query 17**:

```
INSERT INTO  TMP
SELECT * FROM TENKTUP1, TENKTUP2, ONEKTUP
WHERE (ONEKTUP.unique1 = TENKTUP1.unique1)
AND   (TENKTUP1.unique1 = TENKTUP2.unique1)
AND   TENKTUP1.unique1  BETWEEN lower AND upper;
```

[2] An index is a fast access path to tuples regarding to a small number of fields. The index pages are relatively small in respect to the database and mostly fit entirely in the buffer. Therefore accessing a tuple through an index costs one index-lookup in main memory and one access to the database. The difference between a clustered and a non-clustered index is that the tuples due to the index are located sequentially in pages. Sequential locations increment the buffer-hit-ratio and reduce the query response time, respectively.

During tests with the real system, it turned out that the query response time of Query 4 is significantly shorter than of query 6, 15, 17. This is because of the clustered index. Thus, I decided to define 4 different workloads using theses query-types separately. The workload types are: (WL_1, WL_2, WL_3, WL_4). WL_1 contains 4 queries of type 4, WL_2 contains 4 queries of type 6, WL_3 contains 4 queries of type 15 and WL_4 contains 4 queries of type 17.

Section 7.4.2 presented four different distributions for the upper and lower bound; *HotCold, HiCon, Private, Uniform.* The workload types represent different application domains in which the system can be used. Since validation does not want to analyze the systems behavior looking at the different workloads, no special workload type is chosen. Every single workload contains four query variations regarding the upper and lower bound. The following table lists the ranges of every query for every client.

Lower Bound	Upper Bound
0	9999
10000	19999
20000	29999
10000	19999

Table 9.1: *Upper and Lower bound of the four queries in workload type WL_1 to WL_4*

9.2 Configuration of the Commercial Client-Server Database System - Oracle 7.3.2 Client-Server DBMS

There exist many commercial client-server DBMS for relational databases such as (Oracle, Ilustra, ...). These products are very similar to the standard client-server model presented in Section 6. In order to verify the model, Oracle 7.3.2 is measured using the workload defined in the previous section.

The Oracle 7.3.2 server is configured on a HP 735 machine with 135 MHz and 144 MByte main memory using the operating system OS THP 10.0. It has a 50K bytes buffer for page-buffering (4kbyte pagesize) integrated in the Oracle DBMS. The system was measured in the following configuration. There is one client and one server placed on the same machine (Single-Client-Single-Server Configuration). Network cost is ignored, and later in the simulation the network bandwith is assumed as being high enough so that it does not affect the results. The client process only sends and receives queries so that neither disk I/Os nor CPU-time must be taken into consideration at client side. All use of system resources is assigned to the server process.

The operating system buffers files. This affects the file access time if needed pages are loaded into the buffer. In order to reduce this affect the database is scaled up to contain 630.000 tuples ($\approx 138 Mbyte$), the tuple size remained the same. Table 9.2 shows the different table sizes.

Table	**number of tuples**	**size in MByte**
TENKTUP1	300.000	≈ 65.6
TENKTUP2	300.000	≈ 65.6
ONEKTUP	30.000	≈ 6.6

Table 9.2: *Scaled up Wisconsin Benchmark Tables*

9.3 Measurement of Oracle 7.3 Client-Server DBMS

This section presents the measurement results of the Oracle client-server DBMS configuration (Single-Client-Single-Server Configuration) presented in the previous section. The results are collected in Table 9.3. The following sections will discuss the behaviour of the system as a basis for the comparison to the simulator, done in chapter 10.

Workload Type	**Query Response Time [sec]**		**Disk I/O [4kbyte page]**	
	total	**for each query**	**total**	**for each query**
WL_1	12.1	3.04	3972	993
WL_2	404.6	101.14	37392	9348
WL_3	1136.6	284.16	77918	19480
WL_4	1461.8	365.44	98210	24553

Table 9.3: *Measurement Results for the Single-Client-Single-Server Configuration*

9.3.1 Query Response Time

The query response time varies from 3.04 sec (query no. 4) to 294.4 sec (query no. 17). Query no. 4 and no. 6 have the same number of result tuples[3]. The reason for the different query response times can be found in the different access paths. Query no. 4 accesses the Table *TENKTUP1* using a filter on *UNIQUE2* on which a clustered index is defined. Because of the clustered index less pages must be read from disk. Computing query no. 6 requires more page reads because only an unclustered index is defined.

Looking at query no. 15, one can see that that the query response time rises from 101.14 sec (query no. 6) to 284.16 sec (query no. 15). The reason for that is based on the number of result tuples. While query no. 6 selects tuples from one table, query no. 15 selects tuples from 2 tables. That is, query no. 6 computes 10000 result tuples, and query no. 15 computes 20000 result tuples. Computing the join is done by looking up the indices for table 1 (*TENKTUP1*) and table 2 (*TENKTUP1*) two times each and then by reading the tuples.

The difference in the query response time of query no. 15 and no. 17 is only 81.28 sec. Although the number of result tuples rises by 10000 tuples as from query 6 to query no. 15 the query response time rises only by 81.28 sec and not as before by 183 sec. This is because the table no. 3 (*ONEKTUP*) has only 30000 tuples (10 times less). $\frac{1}{3}$ of the table is accessed. Thus, the buffer hit ratio is very high.

9.3.2 Disk I/O

In the previously discusssed queries the number of pages that are necessary to read the tuples are estimated by eliminating the cost for flushing and rereading pages. This affect can significantly increase the query response time, especially when the buffer is small and the total amount of reading that has to be done is high. *ONEKTUP* contains only 10% of *TENKTUP1/TENKTUP2*. Thus the buffer hit ratio for *ONEKTUP* is significantly higher, and, therefore, the number of disk I/Os to read all tuples necessary for a query is less.

In order to find an explanation for the behaviour of the disk I/Os for query no. 15 and no. 17, one must take into consideration the size of the table *ONEKTUP*, as was mentioned in the previous section.

[3] See Chapter 9.1 for the query definition. Remember that *UNIQUE1/UNIQUE2*, on which the filter is set, has a unique domain.

9.4 Simulation Runs with the Standard Client-Server Simulator

This section presents the results obtained by the simulator for every single workload type (WL_1 ... WL_4). They will be compared with the performance indices of the real system.

Table 9.5 gives an overview over the parameter setting that adjusts the simulator to Oracle 7.3.2. The right parameter setting is found in the calibration step (see Section 3.1).

Workload Type	**Query Response Time [sec]**		**Disk I/O [4KByte page]**	
	total	**for each query**	**total**	**for each query**
WL_1	11.69	2.92	2512	628
WL_2	464.9	116.20	39892	9973
WL_3	1092.2	273.05	80047	20011.8
WL_4	1428.04	357.01	104724	2618

Table 9.4: *Results of the simulator runs with* WL_1 ... WL_4

Table 9.4 presents the results for the performance indizes *query response time* and *disk I/O*.

9.5 Comparison Standard Client-Server Simulator and Oracle 7.3.2 Client-Server DBMS

This section compares the measurement of the Oracle 7.3.2 client-server DBMS, presented in the previous section with the results obtained by the simulator of the standard client-server model. Goal of this section is to verify that the model of the standard client-server behaves reasonable ($\pm 5\%$) close to the real system. First the query response time of both systems are compared, then in a second step the number of disk I/Os that are necessary to compute the queries are compared.

Parameter	**Description**	**Value**
	Server Parameter	
ServerCPU	Instruction rate of server CPU	50 MIPS
MulProGra	Multi Programming Grade	12
DeadlockInt	Deadlock detection interval	5 times
DiskTime	Maximum disk access time	13.5 milliseconds
DiskOverheadInst	No of instructions to get a page out of the buffer	5000
LockInst	No. of instructions per lock or unlock	1000 instr.
ServerBufSize	Server buffer size	50KByte
BufferTime	Time to get a page out of the buffer	4 milliseconds
BlockedTime	Time the lock request is delayed after a blocked request	4
SerSelectInst	Instruction for processing a Selection	100
SerProjectInst	Instruction for processing a Projection	1000
SerJoinInst	Instruction for processing a Join	1000
	Client Parameter	
ClientCPU	Instruction rate of client CPU	33 MIPS
	Network Parameter	
NetTransSpeed	bandwidth of the network	$\frac{5Mbits}{sec}$
FixMsgInst	Fixed no. of instruction per message	20000 Instr.
PerByteMsgInst	No. of instructions per msg. byte	5000 Instr. per 4KB page

Table 9.5: *Parameter Setting for the Simulation Runs (Validation)*

9.5.1 Query Response Time

This section compares the standard client-server model to the real system by the query response time. Column 2 and 3 of Table 9.6 show the query response time of both, the simulator and the Oracle system. Column 4 shows the difference between the two systems in total while column 5 shows the difference in relation to the Oracle query response time. For WL_2 and WL_3 the total difference seems to be too high, but if we look at the last line, where a average of all differences is given, the overall deviation is within the required 5% bound. Figure 9.1 and Figure 9.2 illustrate the results graphically.

Workload Type	**Query Response Time [sec]**		**Difference Simulation - Oracle 7.3.2**	
	Simulation	**Oracle 7.3.2**	**total** [sec]	**relative to Simulator** [%]
WL_1	2.92	3.04	-0.12	4.11
WL_2	116.20	101.14	15.06	12.96
WL_3	273.05	284.16	-11.11	4.06
WL_4	357.01	365.44	-8.43	2.36
average	187.18	188.445	-1.265	5.87

Table 9.6: *Comparison of the Simulation Results and the Measurements (Query Response Time)*

9.5.2 Number of Disk Input and Output Operations

This section compares the standard client-server model to the real system by the number of disk I/O's. The disk I/Os of both systems are collected in Table 9.6. For the disk I/Os the only big deviation between the simulator and the real system can be observed in WL_2. The total deviation is only 761 disk I/Os but the relative deviation is very high because there are only about 4000 disk I/Os necessary to compute the whole query. But the overall difference given in the last line is under 5% as was required. All results are also represented in a diagramm (see Figure 9.3 and Figure 9.4).

This chapter has shown that the performance indices get from the model presented in chapter 6.1 are sufficiently close enough in a single client environment to the indices that can be measured from a specific real system.

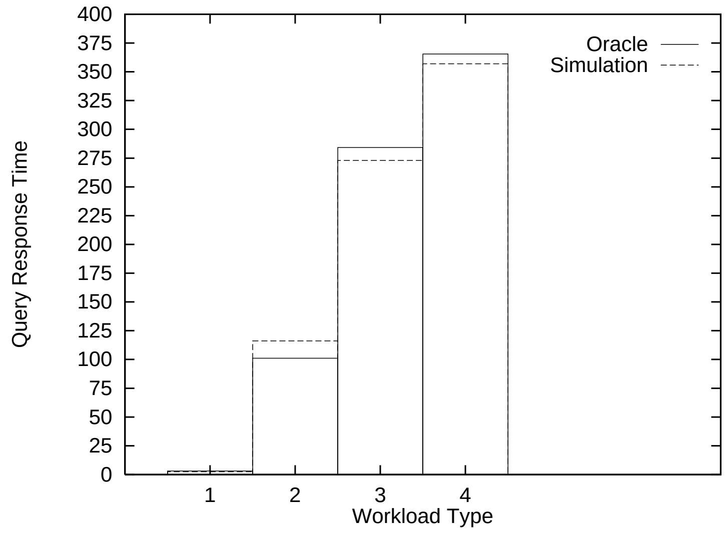

Figure 9.1: *Comparison of the Query Response Time of the Oracle Client-Server DBMS and the Simulator for Single Clients*

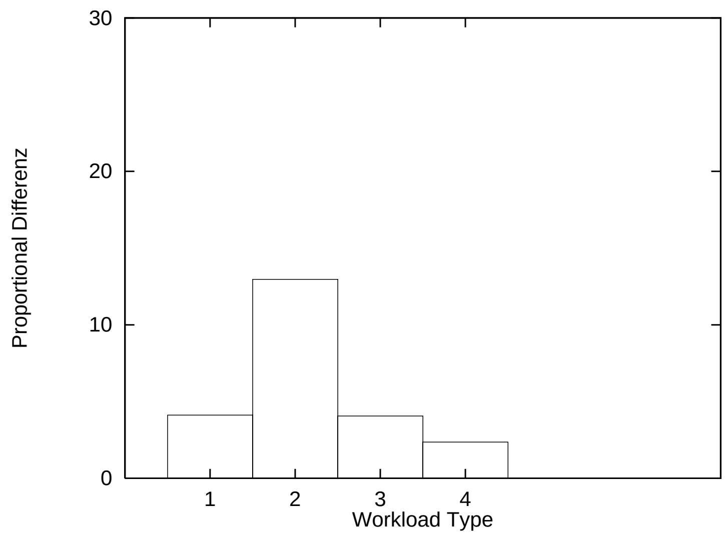

Figure 9.2: *Proportional Difference between the Measurements of the Oracle System and the Output of the Simulator (Basis is the Measurement)*

Workload Type	**Disk I/O [4KByte page]**		**Difference Simulation - Oracle 7.3.2**	
	Simulation	**Oracle 7.3.2**	**total [sec]**	**relativ to Simulator [%]**
WL_1	3211	3972	-761	19.16
WL_2	39892	37392	2500	6.6
WL_3	80047	77918	2129	2.7
WL_4	104724	98210	6514	6.6
average	56969	54373	2596	4.56

Table 9.7: *Comparison of the Simulation Results and the Measurements (Disk I/O time)*

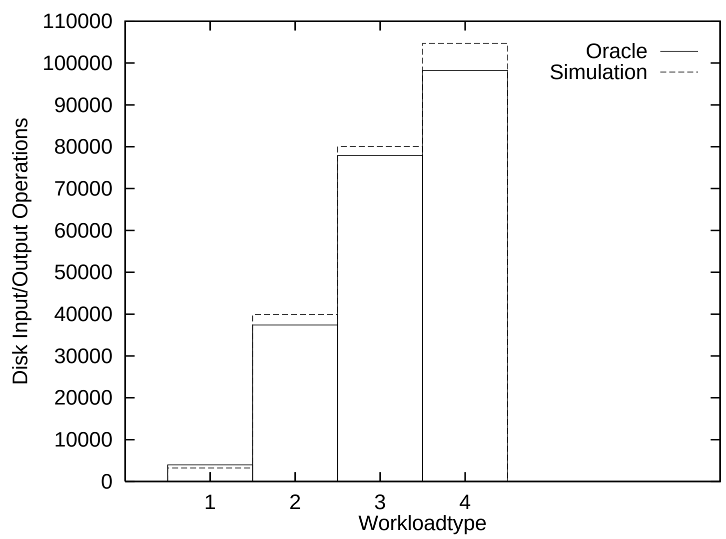

Figure 9.3: *Comparison of the number of Disk I/O of the Oracle Client-Server Architecture and the Simulator for Single Clients*

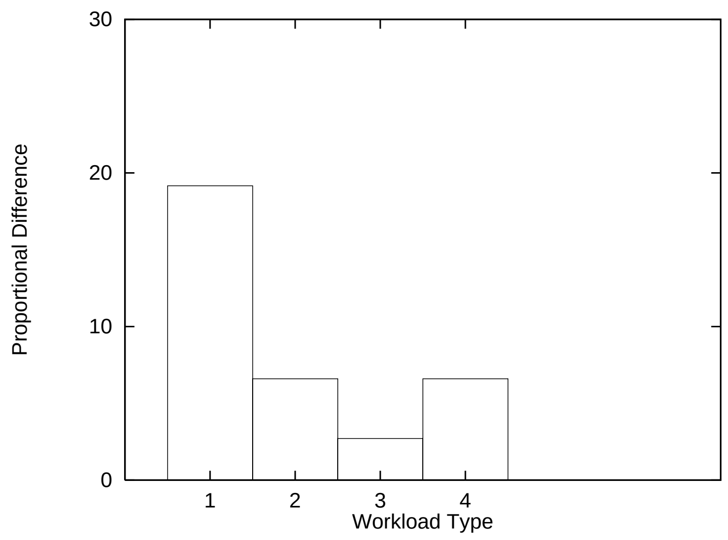

Figure 9.4: *Proportional Difference between the Measurements of the Oracle System and the Output of the Simulator (Basis is the Measurement)*

Chapter 10

Verification of the SQL*Cache Model

This chapter verifies the SQL*Cache model. In order to verify the standard client-server model, a similar commercial system could be measured and the obtained figures could be compared with figures of the simulator. This is a very elegant way to verify a model. Unfortunately there exists no implementation of SQL*Cache yet. Therefore this verification technique could not be used.

Another approach is to examine the model analytically. Analytical exminations lead to fast and accurate results if the underlying model is very small. The model of SQL*Cache is too complex to be analyzed with analytical techniques.

Another technique is to compare it to the standard client-server model and to find fix-points. Fix-points are defined as parameter settings where both systems have the same behavior. Once such a fix-point is found one parameter can be varied in order to examine whether the new system behaves as expected. This is always done paralelly with the standard client model. Because of this, there is always some sort of control experiment to determine whether the behavior of the new system is reasonable or not. Figure 10.1 shows the technique that will be used in the next sections to analyze SQL*Cache.

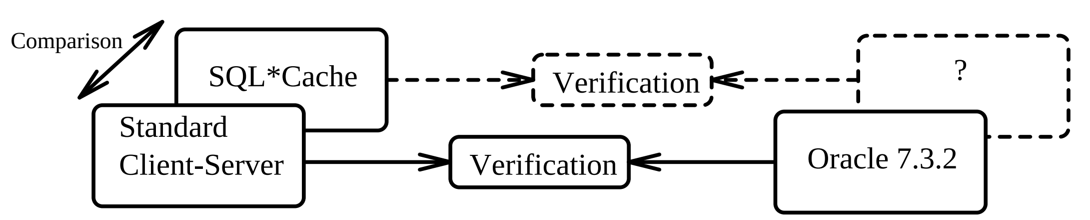

Figure 10.1: *Comparison Technique to Verify SQL*Cache*

In order to find parameter settings that lead to fix-points many experiments are necesarry. In the following sections I will show some of those experiments that lead to fix-points. I will demonstrate experiments in which:

1. Querylength
2. Uniform - Private Workload
3. Cachesize
4. Network bandwidth

are varied.

The query response time depends on many parameters and on the systems state. Before presenting the different experiments done with the simulators, I will point out which parameters and system states affect the query response time; first for the standard client-server and then for the SQL*Cache system.

The query response time for the standard client-server architecture is affected by:

1. **Network Bandwidth:**
   The network is used to send the querytext to the client and to receive its results. The querytext has almost the same amount of bytes for every query, however the number of result tuples can vary enormously (224 kbytes to 674 kbytes). Therefore, the more that tuples are in the query result, the longer it takes to receive these results over the network.

2. **Multi Programming level:**
   The multi programming level affects directly the query response time. While the number of queries sent to the server increases, the throughput of the system rises while, more processes (queries) are being computed in parallel in the system. After a while the throughput is constant or can even fall because of the system being overloaded[1].
   A queue limits the number of queries in the system in order to avoid system overload. The multi programming level is given indirectly by the workload type and directly by the parameter *MPL*.

   (a) **Workload type:**
   In order to fulfill the consistency of the ACID paradigm ([8],[22],[23]), queries that are updating parts of the database must get locks before

[1]Overloading a system is the affect of too many processes competing in parallel for the same resources. This leads to a lower throughput and a higher response time.

they can access these parts. If many queries want to update the same part of the database, only one query is given the lock and the remaining queries are blocked; the queries are put in sequenz. The more concurrent queries for a particular part of the database is, the more queries are blocked.

(b) **Parameter MPL:**
The MPL parameter reduces the number of queries being served in parallel by the system. This is done by a queue where every process is blocked if the maximum number within the system is exeeded [14],[13].

3. **Buffer:**
In order to reduce disk I/O time, parts of the database are cached in main memory (buffer). Accessing a page from the buffer is by far faster than from the disk. Unfortunately, there are costs, too. Filling the buffer with pages and maintaining its contents increases with the buffer size.

The query response time of SQL*Cache depends not only on the network, MPL, querylength and buffersize as in the standard client-server, but also on the cache in every client and on the cost for the cache containment reasoning.

The query response time for the SQL*Cache system is affected also by:

1. **Cache:**
The cache on client side affects the query response time similar to the buffer on the server side. It reduces enormously the query response time if the query can be answered in the cache, but if the cache grows, the maintenance costs grow, too. If many updates are done on the database part that is common among many clients notifications must be sent to various clients. This overhead can lead to lower performance.

2. **Cache Containment Reasoning:**
The cache containment reasoning evaluates whether a query can be computed using only cached data. The cost for this comparison of a given associative query to the stored ranges can be very high. It is affected by the cache size and also by the query length. We assume the cost grows linear with the number of entries in the cache.

These are the main parameters that effect the standard client-server and SQL*Cache performance. Now some of these effects are demonstrated by varying the corresponding parameters.

10.1 Querylength

In this experiment the querylength is varied from 2.3 kbytes (10 tuples) to 272 kbytes (1210 tuples). For each querylength 50 different queries are generated and computed. 25 clients are working in parallel. The query response time is the average of these $50 * 25 = 1250$ queries. HotCold is used as the workload type and no updates are done. The remaining parameters are listed in Table 10.2.

In the standard client-server model, regardless of how long a query is, the query is sent to the server. There, the query response time depends on how long the query is delayed before getting execution permission (multi programming level), how long it waits for tuple locks and how many tuples are read from disk. The number of queries that are waiting for execution permission is constant if we assume the system to be in a steady state. This delay is therefore constant. Each query is read-only and besides lock management costs, no delay occurs waiting for locks.

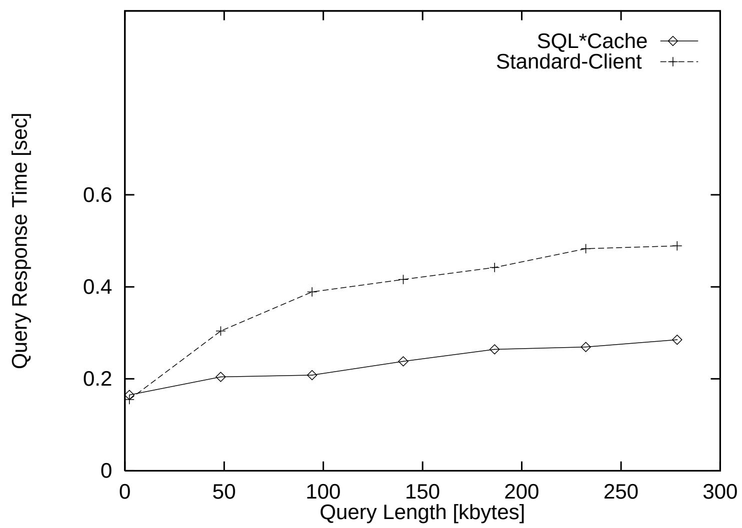

Figure 10.2: *Query Response Time by Variation of the Querylength*

Figure 10.2 shows the query response time depending on the query length for both systems. The query response time of the standard client-server and the SQL*Cache grows continuously with the queries get longer. With a query length of 2.3 kbytes both systems have the same query response time (fix-point). Then the standard client-server query response time grows significantly faster until a query length of 226.86 kbytes. After that, both systems grow equally. It is very interesting that SQL*Cache grows slower and is always less than the query response time of the standard client-server.

Figure 10.3 explains why. This figure shows the cache hit ratio of the client's cache. For very short queries the cache hit ratio is very low (10%). That means that SQL*Cache behaves almost like the standard client-server model, because it must send the query almost always to the server. A query length of 47.2 kbytes (210 tuples) is long enough to increase the cache hit ratio to 70%.

The sensibility of the cache hit ratio to the query length is because of the patterns within the cache. Each query is an associative query. The cache manages ranges, and the cache containment reasoning looks whether a given range is included entirely by a range in the cache. The smaller the ranges are the more little disconnected ranges are in the cache and the more often a new range is not entirely included by a range in the cache.

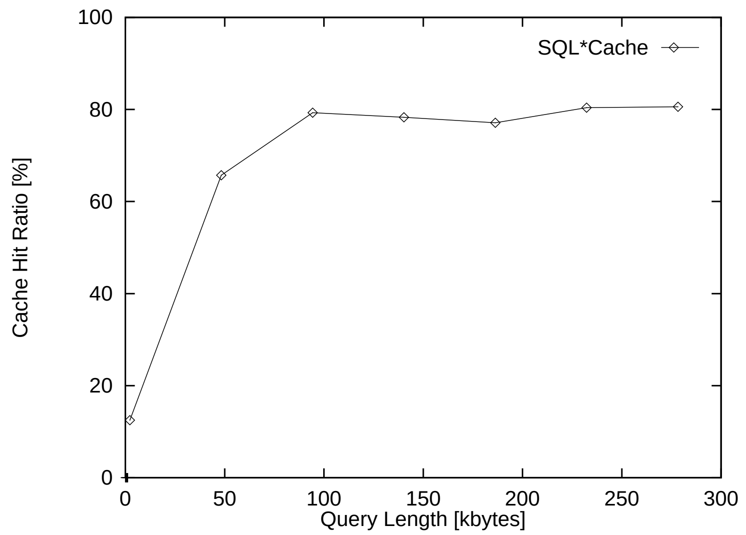

Figure 10.3: *Cache Hit Ratio by Variation of the Query Length*

Parameter	**Value**
Number of Clients	25
Number of Queries	50
Thinktime between two queries	0
Fix intstruction per message	20000
Net Speed old is 5000000. in byte	50000000
Query Length (Basis is 100 type query)	0.01—0
Server CPU Speed 50000000.	60000000
TENKTUP1	10000
TENKTUP2	10000
ONEKTUP	1000
Pagesize	4096
Hotaccess probability	80
HotRangeRatio	80
Multiprogramminggrad	4
Disk Speed	0.0115
Size of Querytext	300
PerByteMsgInst for 4096 Bytes	10240
Client CPU Speed	33000000
RecordSize	230
BufferSize;2000000. (minimum is 25% of database size)	50000000
DiskOverhead Instructions	5000
Workloadtype 0=HOTCOLD, 1=PRIVATE, 2=HICON, 1=UNIFORM	0
tracedriven (from file: argv[3])	0
Hotwrite probability for a query	0
Coldwrite probability for a query	0
Hotwrite probability for a tuple	0
Coldwrite probability for a tuple	0
Standard Client Server = 1, SQL*Cache = 0	0
Buffer Time to retrieve one tuple from cache (old value is 0.0000035)	0.000004
CacheSize	10000000

Table 10.1: *Parameter Setting for the Simulation Run: Query Length (Part 1)*

Parameter	**Value**
LockInst (Instructions to get a lock)	1000
SerSelectInst	100
SerProjectInst	1000
SerJoinInst	1000
CompCostPre	1000

Table 10.2: *Parameter Setting for the Simulation Run: Query Length (Part 2)*

10.2 Cache Size

In this experiment the clients cache size is varied from 500000 bytes (1/2 Mbyte) to 2500000 bytes (2.5 Mbyte). 8 clients that send 25 queries each are considered. The query response time is the average of all query response times. The database is scaled up to 1123 kbytes (5000 tuples) in ONEKTUP and 11.23 Mbytes (50000 tuples) in TENKTUP1 and TENKTUP2. The queries are generated from the HotCold workload type and no updates are done. The remaining parameters are listed in Table 10.7.

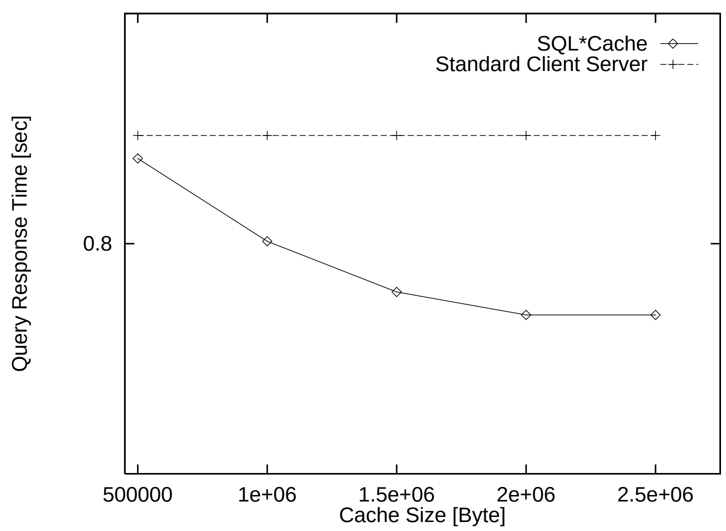

Figure 10.4: *Query Response Time by Variation of the Cache Size on Client Side*

Figure 10.4 shows the query response time depending on the cache size. Obviously the query response time of the standard client-server does not vary at all. However the query response time of SQL*Cache decreases with the cache size increases. With a cache size of ca. 500000 bytes both systems have almost the same query response time (fix-point). Figure 10.5 shows the cache hit ratio depending on the cache size. It explains why both systems have the same behavior for small cache sizes. The cache hit ratio raises from almost 0% with a cache size of 500000 bytes to over 10% with a cache size of 2500000 bytes. If the cache is too small SQL*Cache does not reduce its query response time because it must flush pages out of the buffer and load others. This overhead explains also why the query response time of SQL*Cache is bigger for very small caches.

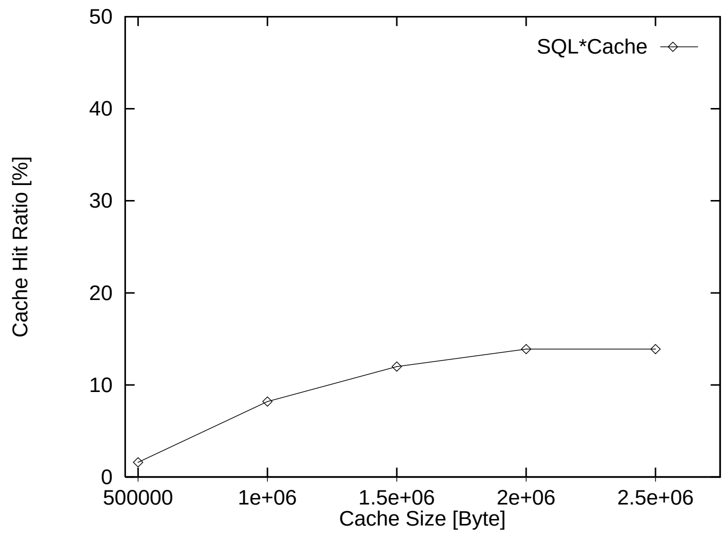

Figure 10.5: *Cache Hit Ratio by Variation of the Cache Size on Client Side*

Parameter	**Value**
Number of Clients	8
Number of Queries	25
Thinktime between two queries	0
Fix intstruction per message	20000
Net Speed old is 5000000. in byte	50000000
Query Length (Basis is 100 type query)	1
Server CPU Speed 50000000.	60000000
TENKTUP1	500000
TENKTUP2	500000
ONEKTUP	50000
Pagesize	4096
Hotaccess probability	50
HotRangeRatio	50
Multiprogramminggrad	12
Disk Speed	0.0115
Query 1	0
Query 2	0
Query 3	0
Query 4.	1
Query 5	0
Query 6.	1
Query 7	0
Query 8	0
Query 9	0
Query 10	0
Query 11	0
Query 12	0
Query 13	0
Query 14	0
Query 15.	1

Table 10.3: *Parameter Setting for the Simulation Run: Cache Size (Part 1)*

Parameter	**Value**
Query 16	0
Query 17.	1
Size of Querytext	300
PerByteMsgInst for 4096 Bytes	10240
Client CPU Speed	33000000
RecordSize	230
BufferSize;2000000. (minimum is 25% of database size)	50000000
DiskOverhead Instructions	5000
Workloadtype 0=HOTCOLD, 1=PRIVATE, 2=HICON, 1=UNIFORM	0
tracedriven (from file: argv[3])	0
Hotwrite probability for a query	0
Coldwrite probability for a query	0
Hotwrite probability for a tuple	0
Coldwrite probability for a tuple	0
Standard Client Server = 1, SQL*Cache = 0	0
Buffer Time to retrieve one tuple from cache (old value is 0.0000035)	0.000004
CacheSize	8192
LockInst (Instructions to get a lock)	1000
SerSelectInst	100
SerProjectInst	1000
SerJoinInst	1000
CompCostPre	100

Table 10.4: *Parameter Setting for the Simulation Run: Cache Size (Part 2)*

10.3 Different Workload Types

This experiment is a comparison of two workload types. Until now only one workload type, the HotCold type was taken into consideration. Figure 10.5 shows the query resonse time of the UNIFORM workload type, while Figure 10.7 presents the query response time of the PRIVATE workload type. I have chosen these two workload types because the query ranges of the first one differ all over the database while in the latter one each client has its own workspace. This leads to a low cache usage in the UNIFORM and a high usage in the PRIVATE cache.

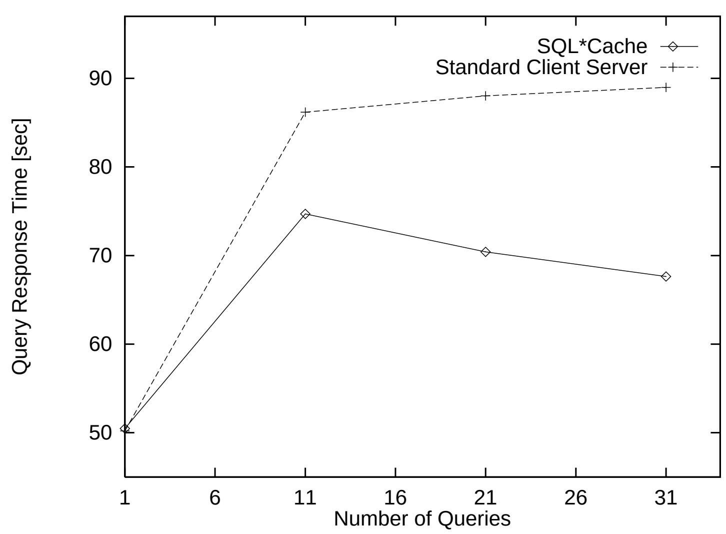

Figure 10.6: *Query Response Time by Variation of Number of Clients (UNIFORM)*

In the UNIFORM workload the query response time for the standard client-server raises from 50 sec if only one query is sent to 90 sec if 11 or more queries are sent. There should not be a deviation in the query response time between one or more queries for the standard client-server model. The population of pages with tuples is done with a random distribution, but is is the same for every simulation run. The low query response time is because of a high page hit ratio for this query. After 11 queries the query response time does not raise anymore for the standard client-server. For SQL*Cache the query response time falls continually. This slow decrease is because of the higher cache hit ratio.

In the private workload type the query response time of the standard client-server model behaves similar to the former workload type. It is much lower because of the higher buffer hit ratio at the server side, but almost constant after 11 queries. SQL*Cache takes advantage of the PRIVATE workload in two ways. The buffer hit ratio is high and the cache hit ratio is even higher. This is because within the PRIVATE workload type the private (hot) part of each client is very small. There are 15 clients, 80% of the database is hot and 80% of the queries are in the hot part. That means the hot part for one client contains about $\frac{80\%*10000tuples}{15Clients} * 230\frac{bytes}{tuple} = 119.72kbytes$ tuples. The query length is about 224 kbytes (1000 tuples). This fills the cache of every client just after one query.

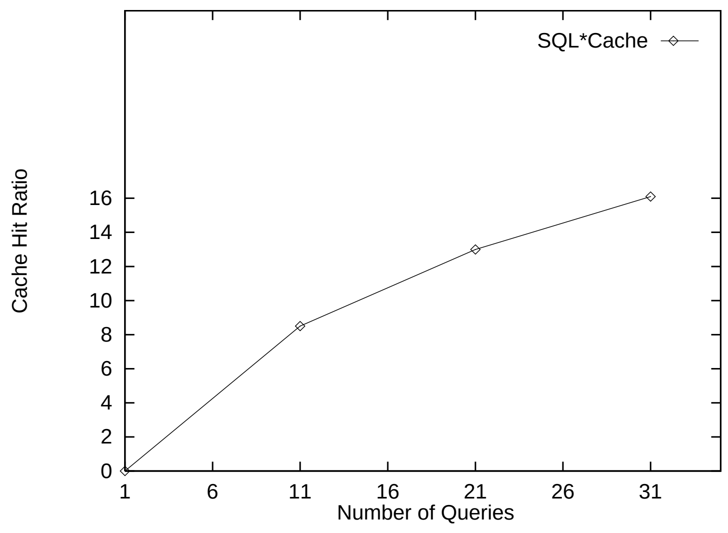

Figure 10.7: *Cache Hit Ratio by Variation of Number of Clients (UNIFORM)*

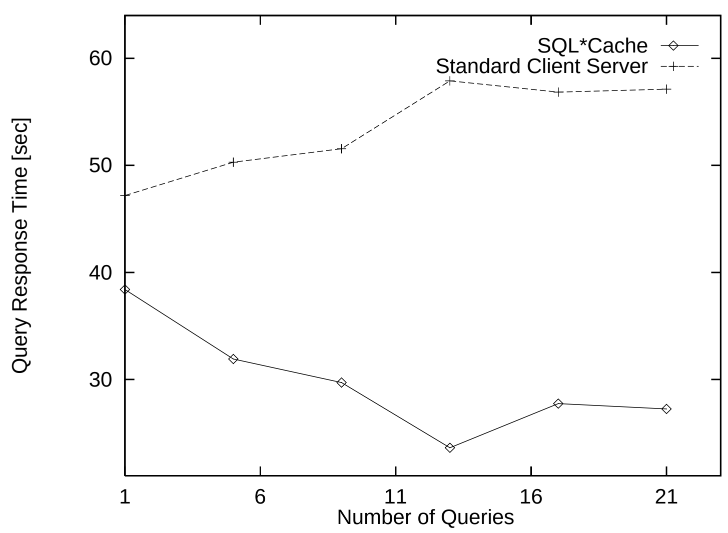

Figure 10.8: *Query Response Time by Variation of Number of Clients (PRIVATE)*

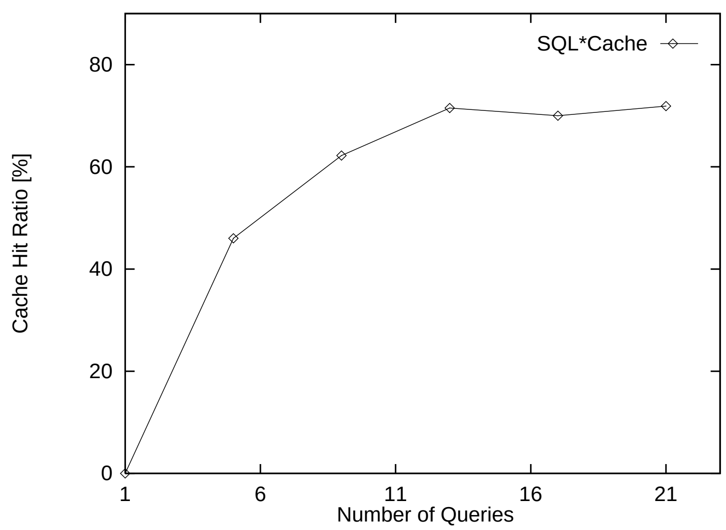

Figure 10.9: *Cache Hit Ratio by Variation of Number of Clients (PRIVATE)*

Parameter	**Value**
Number of Clients	10
Number of Queries	varies
Thinktime between two queries	0
Fix intstruction per message	20000
Net Speed old is 5000000. in byte	50000000
Query Length (Basis is 100 type query)	3
Server CPU Speed 50000000.	60000000
TENKTUP1	10000
TENKTUP2	10000
ONEKTUP	1000
Pagesize	4096
Hotaccess probability	80
HotRangeRatio	50
Multiprogramminggrad	1
Disk Speed	0.0115
Query 1	0
Query 2	0
Query 3	0
Query 4.	0
Query 5	0
Query 6.	1
Query 7	0
Query 8	0
Query 9	0
Query 10	0
Query 11	0
Query 12	0
Query 13	0
Query 14	0
Query 15.	0

Table 10.5: *Parameter Setting for the Simulation Run: Different Workload Types (Part 1)*

Parameter	**Value**
Query 16	0
Query 17.	0
Size of Querytext	300
PerByteMsgInst for 4096 Bytes	10240
Client CPU Speed	33000000
RecordSize	230
BufferSize;2000000. (minimum is 25% of database size)	1207500
DiskOverhead Instructions	5000
Workloadtype 0=HOTCOLD, 1=PRIVATE, 2=HICON, 1=UNIFORM	0
tracedriven (from file: argv[3])	0
Hotwrite probability for a query	0
Coldwrite probability for a query	0
Hotwrite probability for a tuple	0
Coldwrite probability for a tuple	0
Standard Client Server = 1, SQL*Cache = 0	varies
Buffer Time to retrieve one tuple from cache (old value is 0.0000035)	0.000004
CacheSize	1207500
LockInst (Instructions to get a lock)	1000
CacheSize	1207500
LockInst (Instructions to get a lock)	1000
SerSelectInst	100
SerProjectInst	1000
SerJoinInst	1000
CompCostPred	1000

Table 10.6: *Parameter Setting for the Simulation Run: Different Workload Types (Part2)*

10.4 Network Bandwith

The network bandwidth is one of the parameters that influence the query response time. Since now it was not a very important parameter because we assumed a very fast network. Here the affect of a very slow ($\frac{1000 bytes}{sec}$) to a fast ($\frac{110000 bytes}{sec}$) network is investigated.

25 clients are connected using that network. Each client sends 100 queries of 224 kbytes (1000 tuples) and 22.5 kbytes (100 tuples). The database size is 224 kbytes (1000 tuples) in ONEKTUP and 2.25 Mbytes (10000 tuples) in TENKTUP1 and TENKTUP2. The queries are taken from HotCold workload type. The query response time in the figures are the average of $100*25 = 25000$ queries.

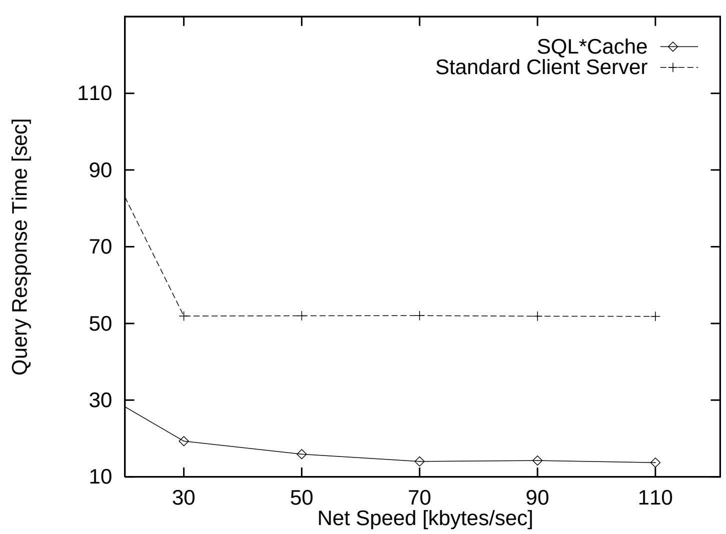

Figure 10.10: *Query Response Time by Variation of the Network Bandwidth*

In this experiment there is no fix-point where both system have the same behavior. But there is another very interesting effect. Figure 10.10 shows the query response time depending on the network speed. The dotted line indicates that with a network bandwidth of 30000 bytes there is no difference in the query response time of the standard client-server regardless how fast the network is. This means that since the network is slower than 30000 bytes it is the bottleneck of the system. After that another component (server) becomes the bottleneck.

The behavior of SQL*Cache is different. The query response time reaches its steady state with a network bandwidth of 90000 bytes. That means that SQL*Cache gains with better network utilization and network bandwidth.

Figure 10.10 shows the cache hit ratio depending on the network bandwidth. Here one can see that the cache hit ratio is very high and constant.

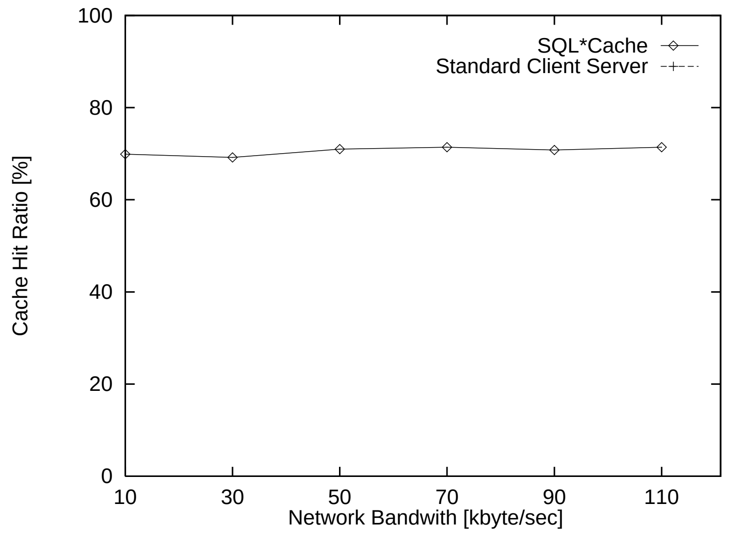

Figure 10.11: *Cache Hit Ratio by Variation of the Network Bandwidth*

Parameter	**Value**
Number of Clients	25
Number of Queries	100
Thinktime between two queries	0
Fix intstruction per message	20000
Net Speed old is 5000000. in byte	10000
Query Length (Basis is 100 type query)	0.1
Server CPU Speed 50000000.	60000000
TENKTUP1	10000
TENKTUP2	10000
ONEKTUP	1000
Pagesize	4096
Hotaccess probability	80
HotRangeRatio	80
Multiprogramminggrad	4
Disk Speed	0.0115
Query 1	0
Query 2	0
Query 3	0
Query 4.	0
Query 5	0
Query 6.	0
Query 7	0
Query 8	0
Query 9	0
Query 10	0
Query 11	0
Query 12	0
Query 13	0
Query 14	0
Query 15.	1

Table 10.7: *Parameter Setting for the Simulation Run: Network Bandwidth (Part 1)*

Parameter	**Value**
Query 16	0
Query 17.	0
Size of Querytext	300
PerByteMsgInst for 4096 Bytes	10240
Client CPU Speed	33000000
RecordSize	230
BufferSize;2000000. (minimum is 25/DiskOverhead Instructions	5000
Workloadtype 0=HOTCOLD, 1=PRIVATE, 2=HICON, 1=UNIFORM	0
tracedriven (from file: argv[3])	0
Hotwrite probability for a query	0
Coldwrite probability for a query	0
Hotwrite probability for a tuple	0
Coldwrite probability for a tuple	0
Standard Client Server = 1, SQL*Cache = 0	0
Buffer Time to retrieve one tuple from cache (old value is 0.0000035)	0.00004
CacheSize	4830000
LockInst (Instructions to get a lock)	1000
SerSelectInst	100
SerProjectInst	1000
SerJoinInst	1000
CompCostPred	1000

Table 10.8: *Parameter Setting for the Simulation Run: Network Bandwidth (Part 2)*

Chapter 11

Conclusion and Future Work

In this thesis a detailed performance evaluation of a predicate based caching system (SQL*Cache) was presented. SQL*Cache was compared to a standard client-server DBMS. This DBMS is a query shipping system with no caching on client side. Queueing models, the most common technique in performance analysis have been used to describe the systems' behavior of both SQL*Cache and the standard client-server DBMS.

A suitable benchmark for SQL*Cache was presented. A study of various benchmarks has shown that there was no suitable benchmark available, and so the well known Wisconsin benchmark was chosen. In order to point out critical issues of SQL*Cache, we customized it to our needs. Among these enhancements was a technique to express contention among many clients and which parts were updated.

Both models were verified. To verify the standard client-server model the very elegant way of comparing the model to a real system (Oracle 7.3.2) could have been used, but in order to verify the SQL*Cache model this technique could not be used because no implementation of a predicate based caching system exists yet. An analytical analysis would have been too laborious and inaccurate. Instead the SQL*Cache model was compared to the standard client-server model. We examined both systems to find common behaviors. As a result, we found parameter settings where both systems' behave commonly (fix-points). Then we varied one parameter and examined the systems behavior.

The simulation analysis has provided valuable insights into the behavior of the SQL*Cache system. There remains further work in finding critical parameter settings where SQL*Cache performs worst or significantly better than a standard client-server DBMS. Therefore, many experiments should be done varying:

1. **Update Probability:**
   There are four parameters that affect updates on the database. The update probability can be defined on tuple and on table granularity. In addition to that, the update probability can be different for the hot and the cold parts of the database. Only with these four parameters a variety of experiments can be done.

2. **Workload:**
   We defined the workload in detail. It would be interesting if a study in a real environment like a bank or a warehouse could be done to get parameters that are typical for these domains. Our workload model could then be calibrated with theses parameters. If this is not possible the experiment space that can be investigated is enormous.

3. **Cache Containment Reasoning:**
   In this simulation study the costs to decide whether a given query is a subset of cached queries grow linear to the number of cached predicates. Maybe this assumption might underestimate the costs in a real system. A study can investigate how susceptible the system performance is to the comparison costs.

Further work remains to be done to compare different cache maintenance strategies and network models.

1. **Cache Maintenance Strategies:**
   Instead of always updating the cache, update notification may cause invalidation of the cached data in a "cold" region and update only "hot" data for the client. Additionally, the cache may itself maintain some data usage statistics in order to have more effective cache storage and replacement policies.

2. **Network Models:**
   The network models used in the simulation study are relatively simple. It is interesting how different network models affect the overall systems' performance.

Appendix A

Example of a Workload Traces for the Real System and the Simulator of the Standard Client-Server Model

```
----- new query -----
Identification No. 1
Wisconsin. No.     15
launch_time        0
select on table    TENKTUP1 with 10000 records
Access path        clustered index
selection interval (where clause)
lower bd.            10000
upper bd.            19999
above results are joined with:
2. table    TENKTUP2 with 10000 records
End Query
----- new query -----
Identification No. 1
Wisconsin. No.     15
launch_time        0
select on table    TENKTUP1 with 10000 records
Access path        clustered index
selection interval (where clause)
lower bd.            20000
upper bd.            29999
above results are joined with:
2. table    TENKTUP2 with 10000 records
End Query
```

```
----- new query -----
Identification No. 1
Wisconsin. No.     15
launch_time        0
select on table    TENKTUP1 with 10000 records
Access path        clustered index
selection interval (where clause)
lower bd.            30000
upper bd.            39999
above results are joined with:
2. table    TENKTUP2 with 10000 records
End Query
----- new query -----
Identification No. 1
Wisconsin. No.     15
launch_time        0
select on table    TENKTUP1 with 10000 records
Access path        clustered index
selection interval (where clause)
lower bd.            25000
upper bd.            34999
above results are joined with:
2. table    TENKTUP2 with 10000 records
End Query
```

Appendix B

The Wisconsin Benchmark Queries Set

```
Query 1 (No Index) and Query 3 (Clustered Index 1 Percent)
QuerySelection
INSERT INTO TMP
SELECT * FROM TENKTUP1 WHERE
WHERE unique2 BETWEEN 0 AND 99

Query 2 (No Index) and Query 4 (Clustered Index 10 Percent)
INSERT INTO TMP
SELECT * FROM TENKTUP1
WHERE unique2 BETWEEN 792 AND 1791

Query 5 1 Percent Selection via a Nonclustered Index
INSERT INTO TMP
SELECT * FROM TENKTUP1
WHERE unique1 BETWEEN 0 AND 99

Query 6 10 Percent Selection via Nonclustered Index
INSERT INTO TMP
SELECT * FROM TENKTUP1
WHERE unique1 BETWEEN 792 AND 1791

Query 7 Tuple Selection via Clustered Index to Screen
SELECT * FROM TENKTUP1
WHERE unique2 = 2001

Query 1 Percent Selection via Clustered Index to Screen
SELECT * FROM TENKTUP1
WHERE unique2 BETWEEN 0 AND 99
```

```
Query 9 (No Index) and Query 12 (Clustered Index) JoinAselB
INSERT INTO TMP
SELECT * FROM TENKTUPl, TENKTUP2
WHERE (TENKTUPl.unique2 = TENKTUP2.unique2)
AND (TENKTUP2.unique2 < 1000)

Query 10 (No Index) and Query 13 (Clustered Index) JoinABprime
INSERT INTO TMP
SELECT * FROM TENKTUPl, BPRIME
WHERE (TENKTUPl.unique2 = BPRIME.unique2)

Query 11 (No Index) and Query 14 (Clustered Index)--
JoinCselAselB
INSERT INTO TMP
SELECT * FROM ONEKTUP, TENKTUPl, TENKTUP2
WHERE (ONEKTUP.unique2 = TENKTUPl.unique2)
AND (TENKTUPl.unique2 = TENKTUP2.unique2)
AND (TENKTUPl.unique2 < 1000)

Query 15(Nonclustered Index JoinAselB)
INSERT INTO TMP
SELECT * FROM TENKTUPl, TENKTUP2
WHERE (TENKTUPl.uniquel = TENKTUP2.uniquel)
AND (TENKTUPl.uniquel < 1000)

Query 16 (Nonclustered Index JoinABprime)
INSERT INTO TMP
SELECT ~ FROM TENKTUPl, BPRIME
WHERE (TENKTUPl.uniquel = BPRIME.uniquel)

Query 17 (Nonclustered Index JoinCselAselB)
INSERT INTO TMP
SELECT * FROM ONEKTUP, TENKTUPl, TENKTUP2
WHERE (ONEKTUP.uniquel = TENKTUPl.uniquel)
AND (TENKTUPl.uniquel = TENKTUP2.uniquel)
AND (TENKTUP1.unique1 < 1000)

Query 18 Projectlon with 1 Percent Projectlon
INSERT INTO TMP
SELECT DISTINCT two, four, ten, twenty, onePercent, string4
FROM TENKTUPl
```

Query 19 Projectlon with 100 Percent Projectlon
```
INSERT INTO TMP                            .
SELECT DISTINCT two, four, ten, twenty, onePercent, tenPercent,
twentyPercent, fiftyPercent, unique3, evenOnePercent,
oddOnePercent, stringul, stringu2, string4
FROM TENKTUP
```

Query 20 (No Index) and Query 23 (with Clustered Index), minimum aggregate function

```
INSERT INTO TMP
SELECT MIN (TENKTUPl.unique2) FROM TENKTUPl
```

Query 21 (No Index) and Query 24 (with Clustered Index), minimum aggregate function with 100 partitlons

```
INSERT INTO TMP
SELECT MIN (TENKTUPl.unique3) FROM TENKTUPl
GROUP BY TENKTUPl.onePercent
```

Query 22 (No Index) and Query 25 (with Clustered Index), sum aggre-ate function with 100 partitlons

```
INSERT INTO TMP
SELECT SUM (TENKTUPl.unique3) FROM TENKTUPl
GROUP BY TENKTUPl.onePercent
```

Query 26 (No Indexes) and Query 29 (with Indexes~lnsert 1 Tuple
```
INSERT INTO TENKTUPl VALUES(10001,74,0, 2,0,10,50,688,
1950,4950,9950,1,100,
'MxxxxxxxxxxxxxxxxxxxxxxxxxGxxxxxxxxxxxxxxxxxxxxxxxxC'
'GxxxxxxxxxxxxxxxxxxxxxxxxxCxxxxxxxxxxxxxxxxxxxxxxxxA',
'OxxxxxxxxxxxxxxxxxxxxxxxxOxxxxxxxxxxxxxxxxxxxxxxxxO' )
```

Query 27 (No Index) and Query 30 (with Indexes)- Delete 1 Tuple
```
DELETE FROM TENKTUPl WHERE uniquel=10001
```

Query 28 (No Indexes) and Query 31 (with Indexes) Update Key Attrlbute

```
UPDATE TENKTUPl
SET unique2 = 10001 WHERE unique2 = 1491
```

```
Query 32 (with Indexes~Update Indexed Nonkey Attribute
UpDATE TENKTUP1
SET unique1 = 10001 WHERE unique1 = 1491
```

Bibliography

[1] Abiteboul, Hull, Vianu, "Foundation of Database," *Addison-Wesley*, 1995

[2] 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.

[3] J. Basu, A.M. Keller, "Degrees of Transaction Isolation in SQL*Cache: A Predicate-based Client-side Caching System," *submitted paper*

[4] D. Bitton, D.J. DeWitt, C. Turbyfill, "Benchmarking Database Systems a Systematic Approach", *VLDB Ninth Int. Conference on VERY LARGE DATA BASES*, Florence, Italy, November 1983.

[5] Bolch, G., "Leistungsbewertung von Rechensystemen mittels analytischer Warteschlangenmodelle" *B.G. Teubner*, Stuttgart, 1989

[6] 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.

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

[8] Jim Cray, Andreas Reuter, "Transaction Processing Concepts and Techniques," *The Morgan Kaufmann Series in Data Management Systems*, San Mateo 1993.

[9] Date C.J., "An Introduction to Database Systems," *Reading, Mass.*, Addison Wesley Verlag, 1981

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

[11] M.J. Franklin, "Caching and Memory Management in Client-Server Database Systems," *PhD thesis*, University of Wisconsin-Madison, 1993.

[12] S. Dar, M. J. Franklin, B. Jonsson, D. Srivastava, and M. Tan, "Semantic Data Caching and Replacement," *22nd Intl. Conference on Very Large Data Bases (VLDB 96)*, Bombay, India, September, 1996, to appear.

[13] D. Ferrari, "Computer Systems Performance Evaluation," *Computer Surveys*, Vol. 10. No. 3: 1978

[14] M. Haas, W. Zorn, "Methodische Leistungsanalyse von Rechensystemen,"*Oldenbourg Verlag*, München Wien 1995.

[15] A. M. Keller and J. Basu, "A Predicate-based Caching Scheme for Client-Server Database Architectures," *Third Intl. Conference on Parallel and Distributed Information Systems*, Austin, TX, Sept. 1994.

[16] J. Liebeherr, E.R. Omiecinski, and I.F. Akyildiz, "The effect of index partitioning schemes on the performance of distributed query processing," *IEEE Transactions on Knowledge and Data Engineering*, June 1993, vol.5, no.3, p. 510-22.

[17] Oracle Corporation, *Oracle 7 Server Concepts Manual*, December 1992.

[18] N. Roussopoulos, C.M. Chen, S. Kelly, "The ADMS Project: Views R Us," *IEEE Data Engineering Bulletin*, Vol. 18, No. 2, June 1995, pp. 19-28.

[19] A. Delis and N. Roussopoulos, "Performance and Scalability of Client-Server Database Architectures," *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, Vancouver, Canada, August 1992.

[20] N. Roussopoulos, "The Incremental Access Method of View Cache: Concepts, Algorithms, and Cost Analysis," *ACM Transactions on Database Systems*, Vol. 16, No. 3, 1991, pp. 535-563.

[21] W.B. Rubenstein, M.S. Kubicar, and R.G.G. Cattell, "Benchmarking simple database operations," *Proceedings ACM SIGMOD*, Where?, 1987.

[22] Jeffrey D., Ullman, "Principles of Database Systems," *Computer Science Press, Inc*, 1982

[23] Jeffrey D., Ullman, "Principles of Databases and Kowledge- Base Systems," *Computer Science Press, Inc*, 1988

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

[25] Kleinrock, L., "Queueing Systems", *John Wiley and Sons*, Volume I, New York, 1975

[26] Kleinrock, L., "Queueing Systems", *John Wiley and Sons*, Volume II, New York, 1976

[27] J. Gray, "The benchmark handbook for database and transaction processing systems,"*Morgan Kaufmann Publishers, Inc*, San Mateo, USA 1993.

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

[29] Gio Wiederhod, "Database Design," *McGraw-Hill international book company*, Japan, 1983

[30] W. Wilkinson, M.-A. Neimat, "Maintaining Consistency of Client-Cached Data",*Proceedings of the 16th VLDB Conference*, Brisbane, Australia 1990.