A Simultaneous Execution Scheme for Database Caching

Der Fakultät Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Steffen Jurk

geboren am 29. November 1974 in Apolda

Gutachter:

Prof. Dr. rer. nat. habil. Bernhard Thalheim

Prof. Dr. rer. pol. habil. Hans-Joachim Lenz

Prof. Dr.-Ing. Jens Nolte

Tag der mündlichen Prüfung: 12.12.2005

Hiermit erkläre ich, dass ich die vorliegende Dissertation selbständig verfasst habe.

Cottbus, den September 22, 2006 ....

Steffen Jurk, Striesower Weg 22, D–03044 Cottbus
Email: `Steffen.Jurk@gmx.de`

Abstract

Database caching techniques promise to improve the performance and scalability of client-server database applications. The task of a cache is to accept requests of clients and to compute them locally on behalf of the server. The content of a cache is filled dynamically based on the application and users' data domain of interest. If data is missing or concurrent access has to be controlled, the computation of the request is completed at the central server. As a result, applications benefit from quick responses of a cache and load is taken from the server. The dynamic nature of a cache, the need of transactional consistency, and the complex nature of a request make database caching a challenging field of research.

This thesis presents a novel approach to the shared and parallel execution of stored procedure code between a cache and the server. Every commercial database product provides such stored procedures that are coded in a complete programming language. Given a request in form of such a procedure, we introduce the concept of *split twin transactions* that logically split the procedure code into two parts, say A and B, such that A is executed at the cache and B at the server in a simultaneous and parallel manner. Furthermore, we analyse the procedure code to detect suitable parts. To the best of our knowledge, this has not yet been addressed by any existing approaches.

Within a detailed case study, we show that our novel scheme improves the performance of existing caching approaches. Furthermore, we demonstrate that different load conditions of the system require different sizes of the parts A and B to gain maximal performance. As a result, we extend database caching by a new dimension of optimization, namely by splitting of the procedure code into A and B.

To solve this problem of dynamically balancing the code execution between cache and server, we define the maximum performance of a database cache over time and propose a stochastic model to capture the average execution time of a procedure. Based on the execution frequencies of primitive database operations, the model allows us to partially predict the response times for different sizes of A and B, hence providing a partial solution to the optimization problem.

Zusammenfassung

Datenbank-Cache-Techniken versprechen eine Verbesserung von Client-Server-Anwendungen hinsichtlich Performance und Skalierbarkeit. Anfragen werden direkt an den Cache geleitet und mit Hilfe lokal gespeicherter Daten ausgeführt. Der Cache-Inhalt wird dabei dynamisch entsprechend dem Verhalten von Anwendungen und Benutzern gefüllt. Fehlen dem Cache Daten oder erfordert die Ausführung der Anfrage die Prüfung transaktionaler Konsistenz, wird die Anfrage vom Server vervollständigt. Anwendungen profitieren von schnellen Antworten des Caches und der Entlastung des Servers. Das dynamische Umfeld, die transaktionale Konsistenz und die Komplexität der Anfragen bilden zusammen einen interessanten Forschungsbereich fr Effizienzsteigerung von Client-Server-Systemen.

In der vorliegenden Arbeit wird eine neue Technik zur geteilten und parallelen Ausführung von Prozeduren aufgezeigt. Nahezu jedes kommerzielle Datenbanksystem bietet heutzutage derartige Prozeduren, die in einer vollständigen Programmiersprache implementiert sind und auf der Ebene des Datenbanksystems ausgeführt werden. Als Erweiterung für Prozeduren werden *Geteilte Zwillingstransaktionen* vorgestellt, die den Code logisch in zwei Teile A und B teilen, so dass A vom Cache und B vom Server zeitgleich ausgeführt werden. Mögliche Teilungen des Codes werden vorweg durch eine Code-Analyse ermittelt. Entsprechend meinen Erkenntnissen wurde dieses Problem bisher von keiner der existierenden Techniken betrachtet.

In einer detaillierten Fallstudie wird die verbesserte Performance der hier dargestellten Methoden aufgezeigt. Weiterhin wird gezeigt, dass verschiedene Lastbedingungen eine unterschiedliche Dimensionierung der Teile A und B zum Erreichen der maximalen Performance erfordern. Somit eröffnet die richtige Wahl von A und B eine neue Dimension zur Optimierung von Datenbank-Cache-Techniken.

Um das Problem des dynamischen Ausbalancierens der Code-Ausführung auf einem Cache sowie dem Server zu lösen, wird die zeitabhängige maximale Cache-Performance und ein stochastisches Modell zur Erfassung der durchschnittlichen Ausführungszeit von Prozeduren definiert. Basierend auf Ausführungshäufigkeiten von elementaren Datenbankoperationen ermöglicht das Modell die partielle Vorhersage von Antwortzeiten für verschiedene Dimensionierungen von A und B. Somit wird eine partielle Lösung des Optimierungsproblems erreicht.

Contents

Abstract iii

Zusammenfassung v

I Overview and Motivation 1

1 Introduction 3
- 1.1 Database Caching . . . 4
- 1.2 Problem Statement . . . 6
- 1.3 Outline . . . 8

2 Database Caching Techniques 9
- 2.1 The Need for Database Caching . . . 9
- 2.2 Database Cache Design . . . 12
- 2.3 Request Execution and Concurrency Control . . . 15
- 2.4 Loading the Cache and Self-Adaptation . . . 18
- 2.5 A Classification of Caching Schemes . . . 20
- 2.6 Related Fields to Database Caching . . . 22
  - 2.6.1 Mobile Databases . . . 22
  - 2.6.2 Integrity Enforcement . . . 24

3 Problem Analysis 27
- 3.1 The Example . . . 27
- 3.2 Traditional Caching Schemes . . . 28
  - 3.2.1 Avoidance-Based Protocols . . . 29
  - 3.2.2 Detection-Based Protocols . . . 31
  - 3.2.3 Discussion . . . 32
- 3.3 A Simultaneous Execution Scheme . . . 33
  - 3.3.1 Twin Transactions . . . 33
  - 3.3.2 Partial Execution and Re-Use of Query Results . . . 34
  - 3.3.3 The Run Time Optimization Problem . . . 39
- 3.4 Summary and Discussion . . . 40

II Split Twin Transactions for Database Caching 43

4 A Client-Server Database System for Twin Transactions 45

4.1 Overview . . . 45
4.2 Data Definition and Stored Procedures . . . 48
4.2.1 Tables and Constraints . . . 48
4.2.2 The Language of Stored Procedures . . . 48
4.2.3 Low-Level Procedures . . . 50
4.3 The Execution Engine . . . 52
4.4 Table Fragmentation . . . 54
4.4.1 The Definition of Fragments . . . 54
4.4.2 Fragment Access of Queries and Low-Level Updates . . . 55
4.4.3 The Management of Fragments . . . 57
4.4.4 Fragment Versions . . . 59
4.5 Fragment Replication and Synchronization . . . 61
4.5.1 Propagate Updates to Caches . . . 62
4.5.2 Receive Updates from the Server . . . 63
4.5.3 Dynamic Data Replication . . . 65
4.6 Executability of IO Statements . . . 65
4.7 Twin Transactions . . . 66
4.7.1 Unique Execution and Tuple Identifiers . . . 67
4.7.2 Consistent Twin Transaction . . . 69
4.8 Related Work . . . 71
4.9 Summary and Discussion . . . 72

5 A Simultaneous Execution Scheme for Split Procedure Code 75

5.1 Split Twin Transactions — An Overview . . . 76
5.2 The Query Result Cache . . . 79
5.3 Execution at the Cache . . . 80
5.3.1 The Partial Execution Model . . . 80
5.3.2 Handling IO Statements . . . 83
5.3.3 Completing an Execution . . . 84
5.4 Execution at the Server . . . 84
5.4.1 Retrieving Query Results from the QRC . . . 84
5.4.2 Handling IO Statements . . . 87
5.4.3 Verifying Query Results . . . 88
5.4.4 Completing an Execution . . . 88
5.5 Identify Dependencies within the Procedure Code . . . 89
5.6 Correctness of Split Twin Transactions . . . 94
5.7 Related Work . . . 95
5.8 Summary and Discussion . . . 96

6 Case Study — The ONE-System 99
6.1 Architecture . . . 99
6.1.1 Database Schema and Procedure Code . . . 99
6.1.2 The Client-Server Database System . . . 100
6.1.3 A Traditional Execution Scheme . . . 101
6.1.4 The Simultaneous Execution Scheme . . . 103
6.2 Experimental Setup . . . 104
6.2.1 Used Hard- and Software . . . 104
6.2.2 Parameters of Experiments . . . 105
6.2.3 The Execution of Experiments . . . 106
6.3 Comparison of the Execution Schemes . . . 108
6.3.1 The Traditional Execution Scheme . . . 108
6.3.2 The Simultaneous Execution Scheme With Dependent Queries . . . 113
6.3.3 The Simultaneous Execution Scheme With Independent Queries . . . 117
6.4 Other Related Experimental Results . . . 120
6.4.1 Performance of Slow Database Caches . . . 121
6.4.2 Shared vs. Normal Mode . . . 122
6.5 Summary and Discussion . . . 125

III Self-Adaptive Load Balancing between Cache and Server 127

7 Dynamics, Performance and the Optimization 129
7.1 The Dynamics of Client-Server Database Systems . . . 129
7.2 Cache Execution Performance . . . 133
7.2.1 Idle and Execution Time of IO statements . . . 133
7.2.2 Minimal Processing Time . . . 135
7.2.3 Maximal Cache Utilization . . . 135
7.2.4 Maximal Code Balancing . . . 137
7.3 Cache Fragment Performance . . . 139
7.3.1 Fragment Access Frequency . . . 140
7.3.2 Dependent Queries in the Procedure Code . . . 140
7.3.3 Fragment Efficiency . . . 142
7.4 The Optimization Problem . . . 143
7.5 Related Work . . . 148
7.6 Summary and Discussion . . . 149

8 Modeling the Process Time of the Simultaneous Execution Scheme 151
8.1 Problem Statement . . . 151
8.2 Valid Subsequences . . . 152
8.2.1 Implied Operations at the Cache . . . 153
8.2.2 Additional Operations at a Cache . . . 155
8.3 Capture the Run Time Behavior of Stored Procedures . . . 159

8.3.1 An Upper Bound for the Number of Valid Subsequences 159
8.3.2 Execution Histories of Stored Procedures 162
8.4 Modeling the Simultaneous Execution Scheme 163
8.4.1 Basic Execution Parameters 163
8.4.2 A Model of the Simultaneous Execution Scheme 164
8.4.3 The Average Processing Time of a Procedure 166
8.5 Preciseness of the Model 168
8.5.1 Evaluation Set 168
8.5.2 Compare Observed and Calculated Processing Time 169
8.5.3 Analyse Frequency Density and Distribution 170
8.6 Summary and Discussion 171

9 Run Time Optimization **173**
9.1 Overview 173
9.1.1 Problem Statement 174
9.1.2 Problem Solution 174
9.2 Simulate the Execution Behavior of Split Parameter 176
9.2.1 Motivating Examples 176
9.2.2 An Algorithm for Modifying the Execution History 179
9.3 Capture Load-Dependent Basic Parameters 181
9.3.1 Determine the Load of a Configuration 182
9.3.2 Determine the Value of Load-Dependent Basic Parameters 183
9.4 Evaluation of the Prediction Technique 185
9.5 Towards Solving the Dynamic Optimization Problem 188
9.5.1 The User-Provided Parameters of the Optimization Problem 188
9.5.2 A 2x2-Phase Greedy Optimization Algorithm 190
9.6 Summary and Discussion 194

10 Conclusion **197**
10.1 Summary and Contributions 197
10.2 Open Directions for Further Work 200

Bibliography **203**

Notations **213**

List of Figures

2.1 Multi-tier database applications 10
2.2 A database cache at an end-user client or application server. 10
2.3 Three requirements of database caching. 12

3.1 Example of an avoidance-based concurrency protocol. 29
3.2 Example of a detection-based concurrency protocol. 31
3.3 Twin Transactions. Left: fast Client. Right: slow Client. 34
3.4 Partial execution and reuse of query results. 35
3.5 Different configurations of the simultaneous execution scheme. 40

4.1 Architecture of a Database Cache. 46

5.1 Architecture of a Database Cache. 75
5.2 Split Twin Transactions with passing a query result to the server. 77
5.3 Retrieve, verify and reuse query result from the query result cache handled by the function $eval_S(.)$. 85

6.1 Diagrams for run time parameters and observed values. 108
6.2 ONE-System with one client and traditional execution scheme. Processing time (left) and cumulated verification time (right) in ms. 109
6.3 ONE-System with one client and traditional execution scheme. Execution scheme for the maximal processing time. 109
6.4 ONE-System with one client and traditional execution scheme. Reuse rate in percent (left) and re-execution time (right) in ms. 110
6.5 ONE-System with four clients and traditional execution scheme. Processing time in ms (left) and throughput (right) of one cache as number of calls. . . 111
6.6 ONE-System with four clients and traditional execution scheme. Reuse rate in percent(left) and re-execution time (right) of one cache. 112
6.7 ONE-System with four clients and traditional execution scheme. Query time in ms (left) and cumulated verification time (right) of one cache. 112
6.8 ONE-System with four clients and traditional execution scheme. Execution scheme for the maximal processing time. 113
6.9 ONE-System with one client and simultaneous execution scheme without independent queries. Processing time in ms (left) and idle time (right). 114

6.10 ONE-System with one client and simultaneous execution scheme without independent queries. Cumulated verification time in ms (left) and re-execution time (right). 114

6.11 ONE-System with one client and simultaneous execution scheme without independent queries. Execution scheme for the minimal processing time. . . . 115

6.12 ONE-System with four clients and simultaneous execution scheme without independent queries. Processing time in ms (left) and throughput (right) of one cache. 116

6.13 ONE-System with four clients and simultaneous execution scheme without independent queries. From left to right: cumulated verification, idle, re-execution and query time. 116

6.14 ONE-System with four clients and simultaneous execution scheme without independent queries. Execution scheme for the minimal processing time. . . 117

6.15 ONE-System with one client and simultaneous execution scheme with independent queries. Processing time (left) and idle time (right). 118

6.16 ONE-System with one client and simultaneous execution scheme with independent queries. Cumulated verification time (left) and re-execution time (right). 118

6.17 ONE-System with one client and simultaneous execution scheme with independent queries. Execution scheme for the minimal processing time. 119

6.18 ONE-System with four clients and simultaneous execution scheme with independent queries. Processing time (left) and throughput (right). 119

6.19 ONE-System with four clients and simultaneous execution scheme with independent queries. From left to right: verification, idle, re-execution and query time. 120

6.20 ONE-System with four clients and simultaneous execution scheme with independent queries. Execution scheme for the minimal processing time. 121

6.21 ONE-System with four clients and simultaneous execution scheme with dependent queries. Processing time (left) and throughput (right) for a slow cache. 121

6.22 ONE-System with four clients and simultaneous execution scheme with independent queries. Processing time (left) and throughput (right) for a slow cache. 122

6.23 ONE-System with four clients and modified simultaneous execution scheme with independent queries. Only shared mode. Processing time (left) and throughput (right). 123

6.24 ONE-System with four clients and modified simultaneous execution scheme with independent queries. Only shared mode. Reuse rate (left) and re-execution time (right). 124

6.25 Throughput and processing time for the ONE-System with four clients and three different execution schemes. 125

7.1 Minimal processing time of the ONE-System for different load conditions and types of requests (left: low-loaded server, middle: high-loaded server, right:slow database cache). 131
7.2 Corrected idle time for a one (left) and four client (right) system with dependent queries. 136
7.3 Processing time for a one (left) and four client (right) system with dependent queries. 137
7.4 Corrected idle time for a one (left) and four client (right) system with independent queries. 137
7.5 Processing time for a one (left) and four client (right) system with independent queries. 138
7.6 Code balance (corrected cache and server idle time) for a one (left) and four client (right) system with dependent queries. 139
7.7 Code balance (corrected cache and server idle time) for a one (left) and four client (right) system with independent queries. 139

8.1 Calculating the processing time of the simultaneous execution scheme. 152
8.2 Example of execution histories. 161
8.3 Example of an execution history. The blue box represents the counter c_{exec}, the white box $c_{invalid}$ and the green box a leaf node and c_{valid}. First (second) sequence refers to lowest (highest) possible update delay. 169
8.4 Observed (left) and calculated (right) processing time for four clients with dependent queries. 170
8.5 Observed (left) and calculated (right) processing time for four clients with independent queries. 170
8.6 Frequency density (black) and distribution (blue) for simultaneous execution scheme without (left) and with (right) independent queries (four clients). . . 171

9.1 Example for modifying the execution frequencies of a history. Left: All statements are executed at the cache. Right: Three statements are removed from the parameter *split*. 176
9.2 Example for modifying the execution frequencies of a history. Left: All statements are executed at the cache. Middle: Four statements are removed from the parameter *split*. Right: One statement with subsequent statements is removed from the parameter *split*. 177
9.3 Example for modifying the execution frequencies of a history. 178
9.4 Query execution time pt_S and server load $mIOS_t$ for a four-client system. . . 184
9.5 Query execution time pt_C and cache load $mIOC_t$ for the four-client ONE-System. 184
9.6 Observed (left) and predicted (right) processing time with four clients and dependent queries. 186

9.7 Preciseness of the prediction technique of relative deviations (frequency density and distribution) between the original observed and the predicted values for dependent queries. 187
9.8 Observed (left) and predicted (right) processing time with four clients and independent queries. 187
9.9 Preciseness of the prediction technique of relative deviations (frequency density and distribution) between the original observed and the predicted values for independent queries. 188
9.10 ONE-System. Left: fragment efficiency in seconds. Right: processing time of a procedure in ms. 190

List of Tables

3.1 Characteristics of the proposed execution scheme. 42

6.1 Range (Domain) of static parameters. 105
6.2 Configuration of the tested client-server systems. 108

Part I

Overview and Motivation

Chapter 1

Introduction

Data-intensive applications often make use of relational databases for maintaining huge amounts of data efficiently. These applications range from small in-house solutions up to global applications that operate on different internet-connected sites. Most of this applications run in a client-server configuration, where multiple clients connect to a central server and initiate data requests. A client can be either a end-user machine or an application server, web server, etc. The server provides techniques for shared data access, thus is responsible for guaranteeing consistency of data. For such systems it is desired that a request is computed quickly and that a high number of requests can be computed simultaneously.

The performance of a client-server database system is a crucial issue that depends on numerous factors. Besides the used hardware these factors mainly cover the following areas:

1. Design and implementation of applications. Starting from the requirement analysis and design of a database application, over its distribution among different sites and its mapping to a physical schema, performance relevant issues can be found all over the life-cycle of an application.

2. Run time behavior. A database system interacts with users and other applications. The imposed load significantly determines the performance of the entire database system.

3. Facilities of the database management system. Research and industry have constantly improved database technology to meet new requirements. Among many others, related topics are: internal data management, buffering, caching, query optimization, integrity enforcement, data types, e.g., objects and XML, data replication, concurrency protocols, etc.

This thesis deals with the latter area, especially with database caching techniques that aim at combining traditional caching of elementary data items with database technology. The dynamic nature of a cache, the need of transactional consistency and the complex nature of a database request make database caching a challenging field of research.

1.1 Database Caching

In a typical client-server relational database environment, clients initiate processes at the central server. The server executes them on shared data and passes the result back to clients. Usually the client is only responsible for presenting results and for handling user input. This is feasible for low-end clients (thin clients) with low computational power and low disk space. However, due to the rapid development and decreasing costs in computer hardware, low-end clients are replaced by more and more powerful clients and workstations. Hence, clients do nothing while awaiting the result of the server, thus, leaving the computational resources and available disk space in an idle state.

The situation is similar for multi-tier architectures, such as the one used by SAP R/3. Here, clients are considered to be very thin and are only used for information display and user-interaction. In the middle tier, application servers perform tasks such as session handling, rendering of forms and HTML pages, and minor data modifications. Such modifications are communicated to other application servers using proprietary synchronization protocols. However, data-centric tasks are performed exclusively inside a central database. The sharing of load between the middle tier and the database server is hard programmed, often leaving the (powerful) application server hardware in an idle state.

In both cases, the central server clearly is a central bottleneck of the system, while at the same time valuable resources at the client (or at the middle tier) are wasted. The basic idea of database caching is to utilize these resources for client-side database computations which requires a light or full database management system to be installed at a client. Then, instead of being idle, a cache (partially) executes the initiated request. The requests, or parts thereof, that can not be computed locally are sent to the server. Local computations reduce network traffic and potentially improve the system's response time and server throughput.

During the last decade numerous caching techniques, such as ([64, 29, 37, 52, 4, 5, 39, 54, 60]), have been proposed. Database caching techniques have to cope with the following main problems:

1. **Database Cache Design**
   Data of a cache has to be organized in some format. This can be data-oriented, such as pages, objects, tuples, or result-oriented, such as full or partial query results. Cache management on a pure tuple basis can be inefficient for databases with a high number of tuples. To efficiently manage tuples, grouping is used to handle multiple tuples at the same time. State-of-the-art are semantic grouping techniques that incorporate dependencies among tuples that result from the database scheme, queries and access patterns.

2. **Request Execution**
   Caching schemes are often highly interwoven with the execution mechanism of a database request. This affects the following instances of a database management system:

   (a) Concurrency Control.
   A database system always has to deliver consistent answers, as does the cache.

Hence, a cache cannot just return a local data item, but has to guarantee that it is consistent w.r.t. concurrent access at the original data item at the server.

Most of the techniques apply an avoidance- or detection-based concurrency protocol. While the first ensures that data is always up-to-date, the second allows the database cache to contain stale data which additionally requires a validation of the data access at the cache.

(b) Execution Engine.

The execution engine is responsible for executing queries and updates, enforcing integrity constraints and executing procedure code as used by stored procedures and triggers.

Caching can be applied to elementary database operations, e.g., SQL queries and updates, or to full requests. In the first case, a cache only looks at a single operation and has to decide whether to execute it on local data or to send it to the server. Thereby, the cache ignores the origin of the operation, e.g., stored procedure, trigger. In the second case, a cache considers all such elementary operations that result from the execution of related code of a database request, e.g., stored procedures, triggers. Then, a cache has to handle multiple operations at once and additionally has to decide which subset of these operations are to be executed by itself or by the server.

The latter case requires a deep integration of a caching technique into the execution engine of a database system.

(c) Query Processor.

For an initiated query at the cache there are three possibilities of its execution. The first is to execute the query fully at the cache. The second is to execute the request fully at the remote server, e.g., if data is not available at the cache.

The third, the mixed execution, requires a deep integration of the caching technique into the query processor. For this, the query is split into two sub-queries, where one is executed locally at the cache and the other at the remote server. After both results have been computed, a combination of both results might be necessary.

3. **Loading and Self-Adaptation**

The traditional purpose of a cache is to provide efficient access to frequently accessed data items. A cache has to detect such items and remove them if they are not up-to-date or not required any more. Hence, a cache has to self-adapt its content dynamically.

Fresh data can be sent by the server to caches, or the cache itself can request new data autonomously. Both are commonly known as push and pull strategy. Cached data can be updated in an asynchronous manner apart from transactional boundaries or in a synchronous manner within a transaction.

Early database caching techniques have emerged from techniques of database page buffering and distributed databases, but there are still big differences.

A cache at a client machine differs from a server in a distributed database system in the following issues. A server uses reliable hardware, maintains a static data schema and hosts primary data. In general, clients (hosting a cache) are not reliable (user shutdown, power safe mode, disconnect from network, etc.). Furthermore, a cache might not be able to access the full computational resources (e.g. a cache at an application server under high load) and, in case of web applications, the connection between cache and remote server might be potentially unreliable concerning transfer rate and data throughput.

The main goal of database buffering is to provide efficient data access on primary memory (RAM), instead of external memory (disk). Thus, buffering techniques cope with problems of page management, such as pre-fetching of pages that are more likely to be accessed.

Beside these areas, database caching techniques benefit from a wide range of techniques in the area of consistency control, data distribution and replication, as well as query planing and optimization. Also, in the field of mobile databases, we identify similarities and overlappings to database caching techniques. Mobile clients are autonomous, temporally disconnected from the network and even desire to perform local operations while being off-line. Hence, a similar situation occurs when secondary data is located at a cache that is used to perform local computations under concurrent access. Among others, one vision is to design a local thin database system (e.g. Oracle Light) that supports data access of applications running at the mobile client and that synchronizes data during the next on-line period.

1.2 Problem Statement

This thesis addresses the problem of database caching specifically for the transactional execution of stored procedures. We propose a new execution protocol for database caching techniques that considers the very nature of stored procedures. The scheme aims at executing parts of the procedure code in a simultaneous and parallel manner between cache and server which further improves the response time and throughput of a client-server database system. To the best of our knowledge, this has not been addressed before.

Consider database applications where processes executed at the server are only implemented as stored procedures, e.g., as pieces of database code executed under transactional semantics. Normally, the client invokes a procedure at the server and awaits the result before continuing. Analogously to existing techniques, a cache is placed between client and server, and a call of a stored procedure is directly sent to the cache which will execute the code.

Essentially, our method works as follows: we automatically translate the code of a procedure into two versions, one installed at the server and one installed at the cache. These two version differ such that the code is logically split into two parts, say A and B. When the procedure is triggered in a cache, both versions start running simultaneously. The cache actually performs the commands in part A and informs the server about doing so, while the server performs part B, and only verifies the results passed over by the client for part A. Only if this verification fails due to stale data, the server also executes part A.

Instead of handling each of the elementary database operations separately, as done by existing techniques, we consider the entire code of a procedure as a whole, thus handle a set

$A + B$ of database operations at the same time. In order to determine the right splitting of the code, we apply an optimization technique that, at run time, tries to select the parameters A and B that lead to the maximum performance of a cache.

Our method will not improve the performance of all applications; rather it relies on situations where other existing caching techniques have been proven successful. Our method will only pay off if a significant amount of computations can be taken moved from the server to the client. The part A has to be chosen carefully by the optimizer, such that the verification of its results at the server succeeds in most of the cases.

In this thesis we work out a solution for the above problem that is divided into the following areas:

- **Client-Server Database System.**
  We define a client-server database system where each client is extended by a primitive database management system. Furthermore, it defines how server data is fragmented and dynamically replicated to those database systems. As a pre-stage of our novel execution scheme it also defines *Twin Transactions* that represent the parallel execution of the code fragments A and B. This system serves as a basis for our work and a database at a cache is extended throughout this work towards a complete database cache. The notion of a client and a cache are used interchangeable.

- **Simultaneous Execution Scheme.**
  On top of the client-server system we define our novel execution scheme and the model for logically splitting the procedure code into the parts A and B. All possible splits are computed at compile time, such that at run time the optimizer has only to select an appropriate split with maximal performance. We define how the execution engine at a client and the server have to be modified. In detail, we define the partial execution model of a stored procedure that is necessary to execute part A at a client and B at the server. Furthermore, we elaborate the server-side verification for query results that are delivered by a client during the execution of part A. Possible splittings of the procedure code are investigated by analysing dependencies within the code and by discovering *independent queries.*

- **Run Time Performance of a Cache.**
  We look at the run time dynamics of client-server database systems and derive possible measures to capture the run time performance of a cache. Based on such a measure, we define the maximum cache performance as a continuous dynamic optimization problem. The problem defines what data to replicate at a cache, how to dimension A, B and how to adapt this setting over time for a changing environment.

- **Run Time optimization.**
  To provide a partial solution for the optimization problem, we develop a stochastic model for the simultaneous execution scheme that is based on the execution frequency of

elementary database operations. This model allows us to partially predict the response time of a procedure for a specific split A, B.

1.3 Outline

This thesis is divided into three parts. The first part consists of the first three chapters, including this introduction, provides an overview on current research in the field of caching and motivates our approach by illustrating an example. The second part defines the client-server system, the simultaneous execution scheme and presents a detailled case study to underline the improved performance of our approach. In the third part we tackle the optimization problem. We look at the dynamics of client-server database systems, define the maximum cache performance and propose the model to predict the response time of procedures for different splits of the code. In the following we briefly sketch the contents of each chapter.

Chapter 2 gives a detailled introduction to the topic of database caches and surveys current techniques. We point to general problems in developing a database caching technique and propose a classification of caching schemes to emphasis open gaps. Chapter 3 provides an illustrating example of a web shopping application that we have analyzed. We compare the performance of two traditional and our own execution scheme for database caching and point to benefits of a simultaneous and parallel execution of the procedure code.

Chapter 4 defines the client-server system which includes our basic assumptions on the underlying client-server database system, the pre-compilation of the procedure code, the execution engine, data fragmentation and replication, and version management. Essentially, this Chapter contains all issues that are *not* directly connected to the novel execution scheme. Chapter 5 defines the novel simultaneous execution scheme that is based on the concept of *split twin transactions*. Chapter 6 present the ONE-System that we have implemented to evaluate our scheme. We perform numerous experiments and underline the performance improvements as presented in Chapter 3. Further, Chapter 6 motivates the dynamic optimization problem.

Chapter 7 discusses the dynamics of client-server database systems, possible performance measures and the dynamic optimization problem. Chapter 8 defines the stochastic model to capture the simultaneous execution scheme. The preciseness of the model and its predicted response time is again evaluated on the ONE-System. In Chapter 9 we extend the model to partially predict the response time for different splittings of the procedure code. This allows us to compute well-performing splittings in advance, thus providing a partial solution for the optimization problem. The preciseness of this prediction is again evaluated on the ONE-System.

Chapter 10 concludes the thesis with a summary and an outlook on future work.

Chapter 2

Database Caching Techniques

Efficient and distributed data management is one of the most important capabilities of future applications. Technological advances in the development of computer hardware (notebook, PDA, mobile phone), fast reliable networks and wireless technologies allow users to participate in global applications independent from their physical location. The key is to access and modify remote information anywhere, anytime, in any way. The development of such systems requires powerful and distributed operating systems, as well as flexible database management techniques. A central issue is efficient data access and semantic correctness of data, which is the basis for collaborative work and transaction processing.

Next we summarize current research in the field of database caching that primarily aims at efficient data access. In detail, we discuss database caching from different perspectives. These are the architecture, design, concurrency control, execution, load and refresh of cache data as well as self-adaptation of cache content. Additionally, we point to related works in the fields of mobile databases and integrity enforcement that partially overlap with database caching.

2.1 The Need for Database Caching

Client-server databases have been well established in modern applications. In a typical setup, clients send requests for data retrieval and update to the server. The server executes them on shared data and passes the result back to clients. The server provides concurrency control, transaction management and recovery facilities for shared data. A client is either a single-user machine for end-users or a multi-user or server-like machine, such as web, application, secondary database servers, etc.

In general, there are three groups of applications as sketched at Figure 2.1. A 2-tier application, e.g., small business software, normally runs in a local network. Internet applications are often implemented in 3-tier architectures where the application server also runs as web server and handles remote requests from Internet users. Multi-tier applications, e.g., SAP/R3, are often uses by big companies that have to maintain data of their world-wide branches efficiently.

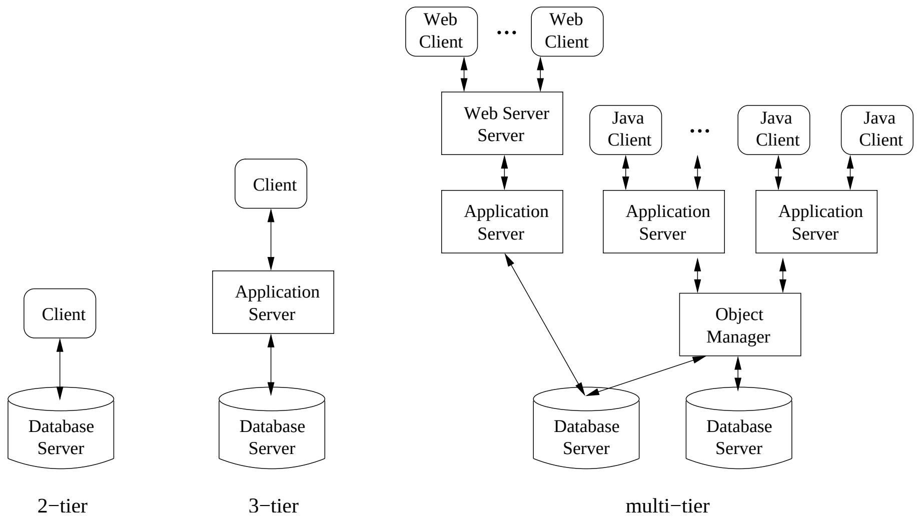


Figure 2.1: Multi-tier database applications

Performance and scalability is important for such applications, since access time is a crucial issue in a highly competitive environment. Static pages can be directly cached at a client or at proxy servers that store frequently accessed pages, pictures and documents for a set of clients, e.g. all web clients of a company. This way load can be taken from the server. However, database applications can not be sufficiently handled by traditional static caching schemes, since requests are complex, e.g. stored procedures or SQL queries, and have to be evaluated at the server due to concurrent access and transactional consistency. To overcome these problems, database caching techniques are proposed by database researchers. The overall goal of a database cache is to improve response time and server throughput.

A database cache is a database system that is integrated between an end-user client, respectively, application and database server (see Figure 2.2). A request from a client or

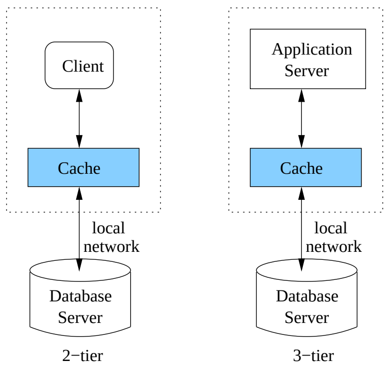


Figure 2.2: A database cache at an end-user client or application server.

application server is first send to the database cache. instead to the server. From the clients point of view the database cache acts exactly as the database server, t.i. processes requests with the same result possibly in a shorter time. From the servers point of view a database cache is a secondary database system with a partial or full copy of the servers data where the server acts as the master and the cache as a slave on the same data schema. In the best case, the database cache is able to compute the query consistently on local data and only communicates with the server for concurrency control. In the worst case, however, the request has to be redirected to the server. Clearly, the challenging issues are data consistency, concurrency control and the amount of data to place at a database cache. In the following sections we take a closer look at this scenario and highlight all relevant issues, including the handling of updates and data synchronization.

The need of database caching technologies is mainly motivated by the following aspects:

- **Increasing price drop in computer and network hardware.**
  End-user clients are only used for information display and user-interaction. Such a client initiates a request at the database server and awaits the result. In the meantime the client is idle. This is feasible for low-end clients (thin clients) with low computational power and small disk space and web clients with an Internet connection of high latency.

  However, due to the rapid development and decreasing costs in computer hardware, low-end clients are replaced by more powerful clients and workstations. Hence, clients do nothing than to wait for the result of the server, thus, leaving the computational resources idle. The situation is similar for application servers in the middle tier where a computation at the database often leaves the (powerful) application server hardware in an idle state.

  In both cases, the central server clearly is a central bottleneck of the system, while at the same time valuable resources at the client (or at the middle tier) are wasted. One of the keys of database caching is to utilize those resources for client-side database computations. Then, instead of being idle, a cache (located at the client) is able to compute the initiated request. Local computations reduce network traffic and potentially improve the system's response time and server throughput.

- **Increasing complexity of applications.**
  As shown at Figure 2.1 complex applications can consist of multiple tiers (layers). Since master data are normally located in the lowest tier, its access from upper tiers is often costly. The main problem for such setups is to provide a *hierarchical* caching mechanism across multiple tiers that provides transactional properties.

  Often such applications are distributed over multiple locations and connected by a wide-area network. The distribution can affect a single or multiple tiers, e.g., main database server on different locations or distributed application server that connect to a central database at one location. As a result, we need caching techniques to support fast local networks as well as slower wide-area networks, c.f. Figure 2.3.

- **Need for transparent and application independent solutions.**
  In the middle tier, application servers perform tasks such as session handling, rendering

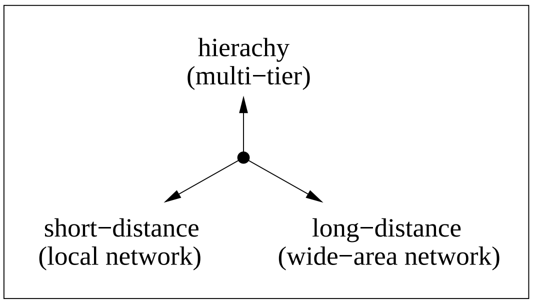


Figure 2.3: Three requirements of database caching.

of forms and HTML pages, object management and minor data modifications. Such modifications are communicated to other application servers using proprietary synchronization protocols. However, data-centric tasks are exclusively performed inside a central database. Often, the sharing of load between the middle tier and the database server is hard programmed and application specific caching is used to provide efficient data access across multiple tiers.

Such specific approaches are used for web page caching, proxy caches and mediators ([6, 15, 70, 2]), or by commercial products, such as EJB IBS WebSphere [95], BEA WebLogic [94] and SAP R/3. A more detailled overview is given in [77, 78].

To avoid application-specific caching solutions, caching has to be integrated at the data level. That is, application servers trigger requests to the database via a standard interface, e.g., a locally installed SQL driver. Independent of the underlying database system and its distribution, the interface might send the request to a remote database or to a local cache that delivers the result. Thus, caching and therewith efficient data access is transparent and independent of applications.

In this work, we cover hierarchical caching, long-distance caching and caching in object-oriented databases only marginal and concentrate on database caching in local networks for relational databases with SQL interfaces. The problem of hierarchical object caching has been studied among other things by [55]. The problem of long-distance caching is mainly relevant in mobile databases as we will point out in Section 2.6.1.

2.2 Database Cache Design

On an intuitive level we can easily sketch some implementations for database caching. So called buttom-up approaches start with data of the database server, aggregate it in some format and ship these data to the cache. Top-down approaches start at the user input level which is the opposite to the data level.

- **Page-Based Caching**
  At the lower level, each database system handles data in terms of pages that are read and

written to disk. A simple caching mechanism would be to replicate LRU (last recently used) or MRU (most recently used) pages at the cache. However, such approaches are not fine-grained enough and pages are replicated even if one single record is accessed by the user.

- **Full Table Caching**
  Another possibility is to use full table caching which, in a sense, is still attractive for existing systems, e.g., Orcale, DBCache [52]. Again, the approach is not very useful, since it is not fine-grained enough and, in the case of large tables, a high synchronization effort is required.

- **Query Result Caching**
  A cache could easily store all queries and their results as posted by users and applications. Whenever the same query is posted, the cache might check whether the previous result is still valid and, if so, answers the query on local data. Whenever the server modifies a table, all corresponding query results are removed from the cache. The problem with this approach is that syntactically different queries cause a new result to be stored in the cache. The challenging issues here are query containment, matching of sub-queries, etc.

State-of-the-art caching approaches combine the benefits of these three intuitive approaches and provide the following properties:

- **Tuple-Based**
  Tuples are the smallest meaningful data unit and allow data caching at a fine-grained level. They also serve as a basis for query answering and integrity enforcement.

- **Grouping**
  The management of a high number of tuples can lead to bad performance behavior, e.g., the checking or faulting of individual tuples causes a large number of small messages between cache and server. For this, tuples have to be grouped meaningfully to be sent from servers to caches into blocks.

- **User-Access Driven**
  A cache should only contain relevant data that is often accessed by users and applications. For this, user-posted data requests have to be taken into account for filling and matching cache content.

In the following we discuss two classes of tuple-based approaches with grouping.

Static Grouping

Client-side data caching has been proposed on the level of page or object identifiers ([42, 102, 106, 16]). Most of these schemes support only *Read* and *Update* operations in transactions on pages and objects to store, retrieve and maintain cache objects. Due to the ID-based access, they are also called *navigation-based* schemes.

The above mentioned, page-based caching can be considered as a static grouping of data. Tables are clustered into blocks and sent to caches. However, in the worst case, a query might retrieve only one tuple out of each block which would require all blocks to placed in the cache. Hence, the problem is to find an optimal static clustering, such that a lowest number of blocks has to be placed in a cache in order to answer frequent user request. In [65] a hybrid system has been proposed to make caching less sensitive to static clustering. The idea is to keep a mixture of pages and individual objects in the cache.

Alternatively, data can be grouped in terms of views. The declarative nature of views allows a better grouping of data. A view, for example, can join together all relevant information of a person that might be spread over multiple tables. Hence, cache data can be built up in terms of materialized views. Materialized view maintenance has been well studied in literature and a number of approaches, e.g., [48, 1, 23, 74], have been proposed. Also academic and commercial database products, e.g., TimesTen Front-Tier [96], use such techniques. However, although materialized views are declarative in nature, they are statically defined, not adaptive and therefore hard to change at run time.

Dynamic Grouping

The major drawback of navigation-based schemes is the inability to handle associate queries that use a predicate, an object class or a relation. Such predicates are very common (e.g. `SELECT * FROM Employee WHERE city='Barcelona' AND Salary>10.000`) and represent accessed data in a declarative, rather then in a procedural manner.

Most of the recent research in literature deals with *semantic caching* which uses a collection of such predicates, derived from queries, to describe the cache content in form of dynamic groups of tuples. Then, the data space of interest of an individual cache is represented by a set of predicates that occur in the queries posted to the cache (*query-shipping*). Given a new request, the cache observes collected predicates and tries to fully or partially execute a query on local data. The query can be executed if (parts of) its predicates (and associated data) are stored in the cache. In case data is missing in the cache, a query has to be executed by the server. For example, the predicate `city='Barcelona' AND Salary>10.000` and tuples matching the predicates in the table `Employee` allow only to derive the query result for a predicate `city='Barcelona' AND Salary<20.000 AND Salary>15.000`.

To our knowledge, the first semantic caching approaches has been suggested by [64, 29] which started with primitive predicates that support projections and selections. These restrictions have been made to decide the implication problem on the predicate level, e.g., a new posted query can only be answered locally if its predicate is implied by the existing predicates in the cache.

Semantic caching has been continuously improved and concepts of *semantic cluster*, *semantic regions* and *cache groups* have been developed. On one side, existing approaches are too restrictive, since they did not address joins. The handling of joins has been discussed by [51, 4, 54, 5]. On the other side, the grouping can be further improved by taking other semantic knowledge into account. In [53, 33, 38, 51] caching is combined with query processing by taking the semantics of an individual query into account to pre-cache data while executing the query plan which represents the procedural execution of a query. As a result it could be shown that different cache operators within query execution plans further improve the execution time of a query.

2.3 Request Execution and Concurrency Control

In the field of client-server databases most of the caching techniques discuss the problem of transactional cache consistency. That is, the client-server system has to guarantee transactional properties in executing database requests. Hence, any usage of the cache does not jeopardize the transactional correctness (ACID properties). We shortly discuss *weak consistency* in transactions processing in Section 2.6.1. Some of the present caching schemes use the built-in concurrency control mechanism of the underlying database system and others propose alternative mechanisms that are often highly interwoven with the caching scheme itself. In the following we discuss the execution of a user-posted request under transactional semantics for the latter kind of caching schemes.

The following instances of a database management system (DBMS) are involved in the execution of a request:

- **Concurrency Control**
  The scheduler of a DBMS is responsible for the transactional execution of the request – its serializability. For this, it uses different locking mechanisms and concurrency protocols.

- **Execution Engine**
  Once a request has been scheduled, the execution engine is responsible for computing the request. This involves:

  - Post queries to the query processor.
  - Apply insert, delete and update of individual tuples. Note that SQL updates normally consists of a query, e.g., `UPDATE Employee SET Salary=X WHERE <cond>`. Hence, the engine has to compute the query results and apply possibly multiple updates of individual tuples.
  - Schema constraints (e.g. primary and foreign keys), rules and trigger are defined for queries and updates. The engine has to check these constraints and has to execute the code of rules and trigger. This is also known as *integrity enforcement* (see Section 2.6.2).
  - Nowadays, every database product provides a procedural language (e.g. stored procedures in Postgres pgplSQL [85], Oracle PL/SQL, etc.). The engine is responsible for executing the code, thus has to handle local variables, recursive procedure calls, exception handling, etc.

- **Query Processor**
  Roughly spoken, a query is executed in the following phases: (1) Compute a query execution plan which might already be known from previous executions. (2) Optimize the plan. (3) Execute the plan on data.

In order to compute a request, a cache has to implement these instances (possibly in a light version). We immediately conclude that concurrency control at a cache has to be done in cooperation with the server. Further, the execution engine either sends updates always to the server or applies them on cached data which requires to withdraw non-updated data or to compute the update on local data (see Section 2.4). To guarantee a consistent database, the data schema of a cache (including schema constraints and procedural code) must be equal to that of the database server. Finally, the query processor has to implement the chosen caching scheme.

Especially recent improvements in database caching benefit from further integrating the caching problem into these instances. As a consequence, caching schemes are integrated more and more into the architecture of database management systems. We outline such integrations briefly.

Utilize the Query Engine to Improve Database Caching

As pointed out at end of Section 2.2, in [53, 33, 38, 51] caching is combined with query processing. The idea is to apply caching during the execution of the query plan. The plan precisely defines which tuples or objects have to be accessed in order to answer the query. Hence, data can be pre-cached (pre-fetched) before the actual execution of the plan accesses these data. As a result it could be shown that different cache operators within query execution plans further improve the execution time of a query.

Concurrency Protocols for Database Caching

In [64] a predicate based caching scheme has been proposed that uses predicate locks for concurrency control. Predicates are needed for their scheme anyhow, such that this extention seems natural. For example, an update `UPDATE Employee SET Salary=X WHERE YearOfBirth<1974` set a lock on the predicate `YearOfBirth<1974` on table `Employee`. Hence, all other queries and updates on this predicate are blocked.

However, a large number of concurrency protocols for database caching has been proposed in literature. The key idea is that transactions are entirely executed at a cache and only communicate with the server in terms of queries, updates, data items (objects, pages, tuples for refreshing and filling the cache content) and concurrency control messages. The goal is that no transaction that accesses stale data at the cache is allowed to commit. According to [66, 41, 37, 39], existing approaches can be classified into *detection-based* and *avoidance-based* algorithms. We outline them both and provide examples in Chapter 3.

- **Detection-Based Protocols**
  These protocols allow the cache to host stale data. Whenever a query is executed by a

transaction on cached data, its result or the accessed data has to be validated by the server. In case of invalid data, the server has to either re-execute the query on shared data or to update the cache with fresh data, such that the cache is able to re-execute the query on valid data.

Once an update is posted to the cache, it has to be registered at the server (e.g. write indication and write lock). The server is responsible for making its decision (approve write access) based on the behavior of other transactions, thus being able to guarantee transactional access.

For committing a transaction, the commit at the cache has to be approved by the server. The server can reject transactions in case already validated queries are spoiled by intermediate updates at the server. This can be easily detected at the server, since all approved queries from the cache and all performed updates at the server are known.

Hence, the advantage of detection-based protocols is its simplicity, since only a single cache and the server are involved in concurrency control. The drawback is the greater dependency on the server which might result in additional overhead, since additional network traffic and validation checks at the server take place.

- **Avoidance-Based Protocols**
  These protocols make it impossible for transactions to ever access stale (invalid) data in their local cache. Invalid data is removed quickly, such that a transaction is prevented from accessing inconsistent data.

  For this, the server must be aware of the content of all caches and remove or update cache data as soon it was modified. In order to do this consistently, a cache must be able to actively take part in concurrency control. Hence, a cache performs read and write indications and has to await the approval of the server. A commit is usually done by a two-phase commit over all attached caches.

  The advantage is that a result computed by the cache need not to be validated at the server. The drawback is the complexity of the protocols, since all caches and the server are involved at commit time.

In general the performance of both protocols depends on the application, complexity of data, etc., such none of both can be ranked superior to the other. In our opinion, the difference between both can be seen as follows: Detection-based protocols lead to explicit overhead for (i) communication with the server, (ii) validation check at the server and (iii) possible compensating actions (re-execution) in case of invalidity. Avoidance-based protocols lead to implicit overhead that occurs during the execution of a transaction. That is (i) overhead by lock management and (ii) possible idle times at the cache in order to await a read or write approval. Detailled performance studies has been done by [41].

Utilize the Execution Engine for Database Caching

Current caching schemes utilize the execution engine only indirectly. For example, tuples are grouped by considering referential constraints. Hence, if a tuple with a reference to other tuples is loaded, the referenced tuples are also loaded into the cache. If referential constraints are considered within grouping, the engine is able to perform integrity checks on local data.

However, to the best of our knowledge, the procedural code of rules, triggers and stored procedures have not been utilized for improving caching schemes. In [29] the authors state:

> *"Obviously, it will only be beneficial to cache objects that are subsequently accessed by the application program. Ideally, one would carry out a data flow analysis of the application program [3] in order to determine which objects of the query result are potentially accessed. Unfortunately, such data flow analysis are impossible in many cases due to the separation of application logic and query processing and interactive applications are totally unpredictable. Thus some heuristic approach to identify the relevant objects is needed."*

In this work we close this gap and propose an execution scheme for database caching that takes the code of stored procedures into account. We show how the execution engine has to be adapted and show the feasibility of our approach within various experiments.

2.4 Loading the Cache and Self-Adaptation

Loading a cache with data is a crucial issue. On one side, the up-to-dateness determines the effectiveness of local query answering. On the other side, data replication is costly and therefore only frequently accessed data should be cached. Furthermore, data that was loaded, but is not accessed any more (due to changing user access behavior), has to be removed from the cache. Otherwise its synchronization is a waste of resources.

Caching is different to traditional data replication schemes, since a cache does not necessarily have to host data and can always redirect a request to the server. Hence, a consistent cache can easily be achieved by removing all stale data. The main approach, however, is to keep cache data up-to-date by loading fresh data or applying updates.

In general, a portion of the server's data is replicated to caches. Replication, however, poses problems in guaranteeing consistency over all replicas. Existing approaches move from *pessimistic* (*eager*) to *optimistic* (*lazy*) [45]. Pessimistic algorithms insist on a single-copy semantics to the user and update all replicas at once by using locking mechanisms to avoid conflicts, e.g., two-phase commit. Optimistic algorithms manipulate replicated data with controlled inconsistencies. According to [89] optimistic algorithms can be classified by three criteria:

- **Single or Multi-Master Systems**
  We consider only single-master systems for database caching, since we target on systems with one central database.

- **Log-transfer or Content-transfer**
  Log-transfer algorithms are characterized by the propagation of changes in data (e.g. updates) rather than whole replicas (e.g. objects, pages or tuples) as in content-transfer algorithms. The advantage of log-transfer algorithms is that updates can be applied to replicas as they are naturally posted by users. However, this might cause additional computational overhead in processing updates at remote sites. The advantage of content-transfer algorithms is that pure data is replicated, such that no updates have to be processed at remote sites. However, this might cause high communication overhead in transferring big objects or pages. Hence, log-transfer is useful for updates with a low computational overhead and content-transfer for small data units. Note that updating one attribute of a big object would require copying the entire object to remote sites in case of content-transfer.

- **Push or Pull-Based Propagation**
  This criteria determines the directions of replication. In push-based schemes the server initiates the replication to remote sites and in pull-based schemes the remote site requests fresh data from the server.

Updating cache data is closely related to concurrency control which guarantees consistent read and write access on cache and shared server data. Basically, we distinguish between synchronous and asynchronous cache updates.

Synchronous Cache Updates

Given a request and the corresponding transaction of its execution, we speak of synchronous updates if all updates on cache data are applied within the boundaries of this transactions. This is mainly used by avoidance-based protocols which require consistent cache data at any time and apply a global commit over all sites. Hence, a synchronous update guarantees that all replicated data and shared server data are in a consistent state.

Instead of updating data of all involved caches, the server can also send notification hints to mark cache data as invalid. This reduces the amount of cache data, but possibly speeds up the global commit. A cache can update marked data within the next requests (transactions) posted to it.

Asynchronous Cache Updates

If cache data is updated independently of transactional boundaries, we speak of an asynchronous update. As a result cache data is potentially not up-to-date and the cache is further responsible for validating local data. This only useful for detection-based protocols, since in case of avoidance-based protocols all cache data has to be up-to-date.

Optimistic replication algorithms are suitable for such scenarios. Depending on data granularity (pages, object, tuples, etc.) a log- or content-transfer method can be used. Whenever there is an update at the server it can be pushed to all caches or alternatively cache data can be marked as invalid by notification hints. A pull-based propagation is useful for two situations:

- Once the cache computes a query on local data and its validation at the server failed, the cache can request fresh data and compute the query again on fresh data. As a result local data has been updated.
- If prefetching is used by the cache, it can bring data into the cache before it is actually accessed.

Self-Adaptation of Cache Content

An important issue, as we can find in many data-distributing systems, is that of self-adaptation or dynamic adaptation of replicated data. The main idea is to make data replication sensitive to changing user behavior, load situations, network properties and topologies, hardware, storage capacity, etc. That includes an "intelligent" placement of new replicas and the removal of unused replicas. In literature a variety of works deal with that problem (e.g. [15, 5, 88, 62, 4, 87] and others) and self-adaptation turns out to be an important property of data-distributing systems.

For loading and refreshing a cache with data the following criteria should be used:

- frequency of data access,
- amount of data to cache,
- synchronization costs for keeping cached data up-to-date and
- the server's load.

The latter one is important, since data that is frequently read and written causes a high synchronization effort, e.g., hot spots, and should not be replicated to caches. Further, it does not always seem advantageous to execute a query at a cache if there is no load at the server. Hence, a database caching approach has to provide an appropriate model for loading and refreshing cache content that takes this criteria into account.

2.5 A Classification of Caching Schemes

Literature proposed different classifications [40, 54] and taxonomies for database caching techniques and related algorithms. In this Section we summarize the classification criteria as observed throughout this Chapter. The criteria are not always independent of each other, such that some of the properties appear in more than one criteria. However, especially the criteria *Integration Level* underlines the novelty of our approach. We classify our approach at the end of Chapter 3 after presenting its main idea.

- **Target Host**
  A cache can be located at single-user machines or multi-user server-like machines (e.g.

application and web server). In the latter, case a separate machine for the cache is also possible.

- **Transparency**
  An application that is accessing the database should not be aware of the existence of a cache.

- **Hierarchy Level**
  Caching can be flat or applied to several levels of a multi-tier application.

- **Data Granularity**
  Data units of a cache can be data-oriented, such as pages, objects, tuples, or result-oriented, such as query results. Both can be mixed as done by predicate-based approaches which keep the result of a query in terms of predicates and all tuples that are relevant for answering the query.

- **Grouping**
  To avoid cache management on a pure tuple basis, grouping is used to handle multiple tuples at the same time. Grouping options are: no grouping, static grouping and dynamic grouping. The latter is used by semantic caching approaches.

- **Local, Mixed and Remote Execution**
  There are three possibilities of executing an initiated request at the cache. The first is to fully execute the request at the cache. The second is a mixed execution at cache and server. For this, the request (usually a query) is split into two sub-requests (sub-queries) where one is executed locally at the cache and the other at the remote server. After the both results have been computed a join of both results might be necessary. The third is to fully execute the request at the remote server, e.g., if data is not available at the cache. Note that a mixed execution is usually used by semantic- and predicate-based approaches. A mixed execution is also known as probe and remainder query [29].

- **Single- or Multi-Requests**
  Current approaches apply caching at the level of a query, an update or an object (single-request). However, often a request or a transaction is of a complex nature, e.g., a stored procedure, as very common in modern database applications (multi-request). However, to the best of our knowledge, the properties and the structure of the procedure code has not be considered by caching schemes yet.

- **Concurrency Control**
  Avoidance-based protocols require that cached data is always up-to-date. Detection-based protocols allow the cache to contain stale data which requires the validation of computations of a cache.

- **Load and Refresh**
  Cached data can be updated in an asynchronous manner apart transactional boundaries

or in a synchronous manner within a transaction. The underlying mechanism has to automatically adapt the content of the cache for changing system and user behavior.

- **Handling of Database Updates**
  Apart from loading and refreshing cached data, a cache either performs an update request on local data or passes it directly to the remote server. Performing the update at the cache improves locality, since a successive request can benefit from the up-to-date cache data. Sending updates directly to the server without performing it on local data, possibly produces stale data and requires a removal or refreshment of local data.

- **Integration Level**
  Caching can affect different parts of the database management system. It can be integrated into concurrency control, the query processor or the execution engine.

2.6 Related Fields to Database Caching

Generally, database caching overlaps with many other main research areas in databases.

Obviously, there is a close relationship to distributed databases, especially transaction management, data replication and materialized view maintenance. A cache differs from a server in a distributed database system in the following terms: A server uses reliable hardware, maintains a static data scheme and hosts primary data. According to [38] client caching can be seen as dynamic replication and second-class ownership. In general we cannot assume that a machine that hosts a cache is active (e.g. user shutdown, power safe mode, disconnect from network, etc.). This results from placing caches on end-user PCs or using farms of applications servers, where one of them might crash. Furthermore, a cache might not be able to access the full computational resources (e.g. cache at an application server under high load) and in case of Internet-connected edge-servers the connection between cache and remote server might be potentially unreliable concerning transfer rate and data throughput.

Another line of related technologies emerges from database internal buffer management. The main goal of database buffering is to provide efficient data access on primary memory (RAM), instead of secondary memory (disk). Thus, buffering techniques cope with problems of page management and prefetching. The purpose of prefetching is to bring data into the cache before it is actually accessed ([81, 27, 44]). As we have shown, first approaches to database caching operated on low-level pages similar to buffering techniques.

Our work is especially influenced by the work in mobile databases and integrity enforcement. We explain the main problems and refer to them in the course of this dissertation.

2.6.1 Mobile Databases

The development of fast and wireless communication networks (GPRS, UMTS, WLAN) and small computing devices (e.g. PDA, mobile phone), founded a new field of reseach — mobile databases. Existing solutions for distributed database could only be partially used and further developments of existing techniques were necessary. Mobile applications have to cope with the following problems:

- **Autonomy of Clients**
  Clients enjoy a high degree of autonomy that has three dimensions: First, clients are able to move between different access points. Second, disconnections naturally occur (e.g. in order to save battery life time), and do not necessarily indicate a failure. Third, a client should be able to provide local support for transactions and queries independently from the other two criteria.

- **Dynamic Network Environments**
  The dynamic behavior of clients yields a dynamic network topology. Additionally, since lifetime of applications usually exceed that of hardware, also the capabilites of hosts (CPU power, storage capacity, display) and networks (band width, speed, geographical expansion) constantly increase. Consequently, mobile computing systems are characterized by dynamic network topology and dynamic hardware equipment.

- **The Consistency - Efficiency Tradeoff**
  Due to the mobility of data hosts and clients, and a possible low bandwidth communication network, data must be replicated, as to efficiently support local data management. However, clients might be willing to trade the up-to-dateness of data for the efficiency of its management. Consistency enforcement in mobile database systems concentrates on achieving an "equilibrium" situation between semantic consistency and efficiency of management.

These problems have a fundamental impact on the architecture and transaction processing in mobile databases.

Transaction Processing

It is commonly accepted that traditional *ACID* properties (*Atomicity, Consistency, Isolation, Durability*) are not suitable for supporting transactions in mobile environments. While durability is the most stable requirement, atomicity, consistency and isolation must be weakened to support autonomy of clients and dynamic network environments. Atomic transactions are problematic, since the distributed computation (supporting mobile hosts) is disturbed by disconnected executions, long delays (movements of clients) and the interactive nature of users at mobile hosts. Similarly, consistency of all replicas and integrity constraints must be adapted to local consistency. As a consequence of weak atomicity and consistency, isolation and concurrency control can also not be supported in the traditional manner.

There is a large body of research on the development of transaction models for mobile and distributed environments [71]. Semantic-based models [100, 101] aim at increasing concurrency by exploiting commutative operations. An open-nested model [25] is used to represent transactions as a set of sub-transactions, allowing disconnections and compensating actions.

In [84, 82] weak and strict transactions are proposed to support dynamic object clustering. Tentative (proxy, twin) transactions are proposed in [45, 87, 26, 86] to capture disconnections.

Another important task of transaction processing is conflict resolution, as to ensure serializability of a set of transactions. Pessimistic strategies detect conflicts and resolve them by aborting or rejecting transactions, thereby requiring transaction logs and recovery. Optimistic strategies dominate in mobile environments which commit transactions and handle possible conflicts afterwards. Hence, transactions can not be necessarily rejected. Here, popular techniques are application-specific resolvers [79]. Another approach [100, 82, 45] is to execute transactions locally and to verify them later at the server which might cause a re-execution in cases of conflicts. Problems of re-execution issues are studied in [69]. The re-execution mechanism is very close to the validation and re-execution step in detection-based protocols for database caching techniques.

Architecture of Mobile Computing Systems

Mobile systems have to replicate data to improve both performance and availability. Replication, however, poses problems in guaranteeing consistency over all replicas. In a mobile computing environment, optimistic algorithms are more suitable, since they allow for site autonomy (requiring less coordination among sites), are less costly, and provide for flexibility (in supporting slow and unreliable networks by propagating updates in the background).

Another important issue in replication is the capability to adapt dynamically to changing network topologies and moving clients. That includes placement of new replicas (e.g. moving clients, client increased storage capacity), removal of unused replicas, and adaptation of background propagation to changing network facilities (e.g. band width or transfer time). Replication techniques try to find optimized placements for replicas.

Architecture of mobile computing systems is often determined by the chosen replication scheme and model of transaction processing. In [45] a two-tier replication algorithm is proposed that allows mobile applications to perform local tentative transactions that are transferred to the server and applied to the master copy later on. They are used in the projects Deno [21, 22] and Bayou [97, 35]. A different architecture that suggests clustering of data according to semantic relationships is proposed in [84, 82]. The semantic clustering (grouping) and the optimistic replication scheme is closely related to the above discussed properties of database caching techniques.

2.6.2 Integrity Enforcement

In database systems, semantic properties of data are defined in terms of integrity constraints. In dynamic situations where operations can violate necessary properties, the role of integrity enforcement is to guarantee a consistent information base by applying additional *repairing actions*. In the simplest case, such a repairing action is to reject a transaction and thereby returning to the previous consistent state. Inconsistencies can also be removed by adding or removing tuples from the database. However, for the latter kind of actions new problems arise that deal with termination, null values, confluence, effect preservation, etc. (see also

[92, 99, 18, 67, 59]). A classification of research efforts in integrity enforcement appears in [74]. Integrity enforcement has also been studied, to some extent, in the area of distributed systems [49, 25, 63, 75].

Rule Triggering System (RTS)

Rules provide an expressive means for implementing database behavior: They cope with changes and their ramifications. Rule mechanisms are used in almost every commercial database system, using features such as `CREATE TRIGGER` or `CREATE RULE`. Rules are commonly used for *Integrity enforcement*, e.g., for *repairing* database actions in a way that integrity constraints are kept ([104, 43, 105]).

For a given set of constraints and operations (events), consistency is enforced by creating a set of *Event Condition Action* (*ECA*) rules. As stated by [104, 18, 92], the automatic generation of such rules is limited for certain classes of integrity constraints.

Since a rule might be repeatedly applied, an application of a set of rules does not necessarily terminate. Sophisticated methods [18, 19] deal with static rule analysis of rules to detect termination problems. Furthermore, different orders of rule execution do not guarantee a unique database state. This is the *confluence* problem of a RTS. Static analysis of confluence in a RTS is studied in [99, 109].

Similar to stored procedures, triggers are implemented in a procedural language. Since a trigger implements additional database behavior, it has to be carefully considered by database caching approaches. We further discuss this issue in Section 4.9.

Effect Preservation

In Rule Triggering Systems a natural expectation is that a fact that was successfully added is retrievable as long as it was not intentionally removed or updated. Such behavior is achievable if contradictory updates that undo each other are avoided. RTS do not meet this expectation, since it is possible that a rule application undoes the actions of previous rule applications in a single repair transaction. The following example demonstrates the problem.

Example 2.1 *Consider a database with a table T_1 with two attributes A, B, a table T_2 with an attribute C, and two integrity constraints: an inclusion constraint ($A \subseteq C$) and an exclusion constraint ($B \cap C = \emptyset$)*[1]*. The inclusion constraint is enforced by the rules:*

$$\begin{array}{lll} R_1: & ON & insert(T_1,(x,\_)) : IF\ (x) \notin T_2\ \ THEN\ insert(T_2,(x)) \\ R_2: & ON & delete(T_2,(x)) : IF\ (x) \in T_1.A\ \ THEN\ delete(T_1,(x,\_)) \end{array}$$

The exclusion constraint is enforced with the rules:

$$\begin{array}{lll} R_3: & ON & insert(T_1,(\_\ ,x)) : IF\ (x) \in T_2\ \ THEN\ delete(T_2,(x)) \\ R_4: & ON & insert(T_2,(x)) : IF\ (x) \in T_1.B\ \ THEN\ delete(T_1,(\_\ ,x)) \end{array}$$

[1]these constraints are simplified versions of possibly more natural constraints like $f(A) \subseteq C$ and $g(B) \cap C = \emptyset$, where f and g are some functions.

The following table show a possible execution of $insert(T_1, (a, a))$. If $a = b$, the resulting update undo the original update.

Rule	*Triggering Primitive Update*	T_1	T_2
–	–	(a, a)	$\emptyset$
R_1	$insert(T_1, (a, a))$	(a, a)	(a)
R_4	$insert(T_2, (a))$	$\emptyset$	(a)
R_3	$insert(T_1, (a, a))$	$\emptyset$	$\emptyset$
R_2	$delete(T_2, (a))$	$\emptyset$	$\emptyset$

□

A seemingly successful insertion update ends up in a state where the insertion is not performed. Moreover, although the insertion actually failed, its repairing updates have taken place. The problem is caused by allowing contradictory updates, e.g., updates that undo the expected effects of each other, within the context of a successful repairing transaction. Rather, the insertion $insert(T_1, (a, a))$ must fail (be rejected) since there is no way to achieve consistency together with effect preservation.

Yet, RTS fall short in enforcing *effect preservation*, e.g., guaranteeing that repairing events do not undo each other, and in particular, do not undo the original triggering event ([91, 92]). Solutions for the problem have been studied in [93, 68, 59, 58].

Chapter 3

Problem Analysis

The goal of all database caching techniques is to execute queries locally in the cache, thus potentially improving the system's throughput and response time. In this work we study database caching with special emphasis on the execution of *stored procedures*.

This Chapter provides an example of our approach and compares it to two traditional approaches. We discuss in detail the resulting run time performance and motivate a simultaneous execution between the cache and server. We address premises and drawbacks of our approach and classify it according to the criteria of Section 2.5.

3.1 The Example

Consider a classical web shop with product groups, products, detailed product information, a shopping cart, payment etc. and also with business processes, such as login, register, add-to-cart, buy, clear-shopping-cart, browse catalog, etc. Also assume that session data is kept in a `Session` table, availability of products in a `Stock` table, and the contents of the shopping carts in a `Cart` table. Each user is identified by a session ID that might become invalid, if the user has not logged out properly.

One of the business processes of the application deals with adding products to the shopping cart. For a given user who wants to put n items of product p into the cart, it could operate as follows:

1. Update the statistic which counts how often the product p has been requested.
2. Check the user's session identifier. If it is invalid, abort the execution and return an error message.
3. Check whether the requested amount n of the product p is currently available. If not completely available, abort the execution and return an error message.
4. If so, add the item to the shopping cart.

Currently, it is very common to implement application behavior in terms of stored procedures within databases. Stored procedures are provided by almost every database vendor and represent complex database programs that can be written in different procedural languages. For the following examples we use a pseudo code notation. A detailed procedural language is introduced in Part II. The above business process could be implemented as follows.

Example 3.1 (The Stored Procedure `AddCart`) *Numbers represent the line of code.* `sessID`, `prodID` *and* `amount` *are parameters of the procedure and represent the user's session identifier, the product identifier, and the amount of requested items. Local variables are* `session` *and* `inStock`*.*

```
PROCEDURE AddCart(sessID,prodID,amount)

 1  UPDATE Stock SET requests=requests+amount WHERE pid=prodID;          $\textcircled{U}$
 2  session := (SELECT * FROM Session WHERE id=sessID);                   $\boxed{S}$
 3  IF session is null THEN
 4     RETURN Error;
 5  inStock := (SELECT storage >= amount FROM Stock WHERE pid=prodID);    $\boxed{A}$
 6  IF NOT inStock THEN
 7     RETURN Error;
 8  INSERT INTO Cart VALUES(sessID,prodID,amount);                         $\textcircled{I}$
 9  RETURN SuccessCode
```

Symbols depicted on the right side represent the SQL queries and updates of the corresponding line of code (query = rectangle, update = circle). They are used in the following to illustrate the execution of the procedure.

In the following sections we consider the scenario where the procedure is executed in common by a cache and a server. We do not consider executions that lead to an error, since they occur rarely and therefore have little impact on the system's performance, and we discuss only those executions that lead to a sequence of SQL operations $\textcircled{U}\,\boxed{S}\,\boxed{A}\,\textcircled{I}$.

3.2 Traditional Caching Schemes

Assume that in our Internet application example each web and/or application server hosts a database cache. Then, a database request initiated by the server is sent directly to the cache instead of to the database. The cache is responsible for processing the entire procedure code.

In our example the cache executes the stored procedure statement by statement. Since the queries $\boxed{S}$ and $\boxed{A}$ have only one predicate, they cannot be split into two sub-queries. Hence, the cache can either perform the full query locally or send the query to the server for execution. Since updates $\textcircled{U}$ and $\textcircled{I}$ must be applied on all data in the client-server system, we assume for this example that updates are performed on cache and server data during the procedure's execution. Additionally, the server is responsible for propagating the updates to other caches. As stated before in Section 2.4, there are different possibilities to handle updates in caching systems.

The following subsections discuss the execution of the example for *avoidance-based* and *detection-based* protocols (see Section 2.3) and motivate our simultaneous execution scheme.

3.2.1 Avoidance-Based Protocols

Avoidance-based protocols make it impossible for the execution at the client to ever access stale data. Hence, the local execution of a query always results in consistent data and the concurrency protocol guarantees this property among all caches that are associated with the client-server database system.

Figure 3.1 depicts a possible execution of the `AddCart` procedure using such a protocol. In detail the execution behaves as follows:

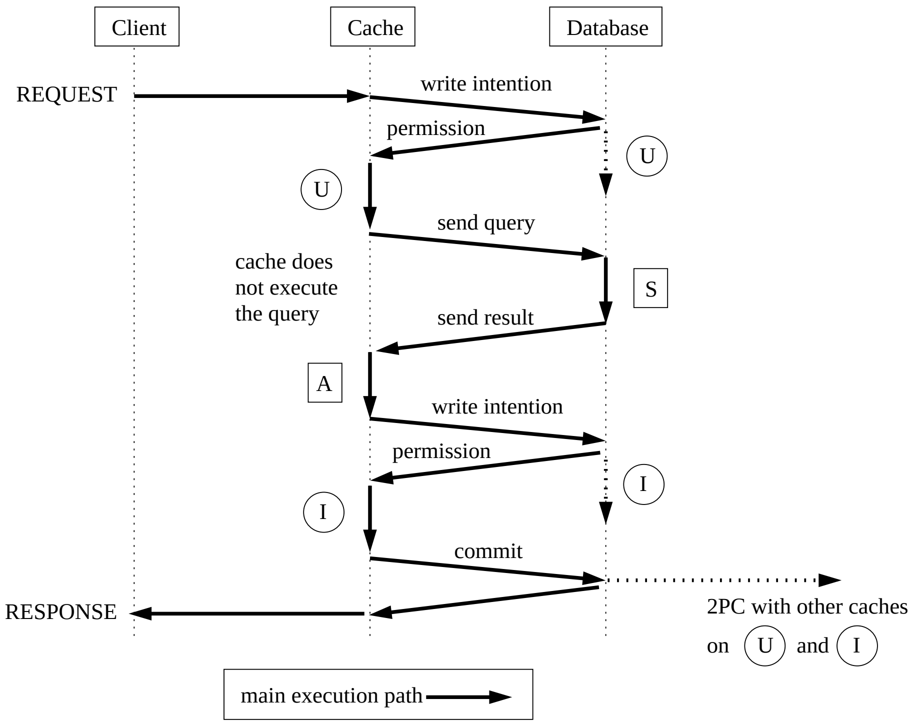


Figure 3.1: Example of an avoidance-based concurrency protocol.

1. A client initiates the procedure `AddCart`
2. Before the update (U) can be executed, the cache sends a write intention to the server. Once the server gives permission, (U) is executed on cache and server data (line of code 1). The server is responsible for the update not interfering with other read and write operations at the server until the procedure commits or aborts. For this, the server normally applies a type of locking.
3. Assume that the session data is frequently updated and therefore not replicated to

a cache. Hence, the query $\boxed{\text{S}}$ that is checking session data is sent to the server for execution.

4. Once the server has retrieved the session data, they are checked (code line 3-4) and the execution proceeds with query $\boxed{\text{A}}$. Assume that the stock data is available in the cache, such that the query can be executed locally.

5. Check stock data (code line 6-7) and the execution proceeds with update $\textcircled{\text{I}}$. The cache also has to indicate a write operation to the server and to wait for permission, before the update is applied on cache and server data.

6. The execution commits by sending an appropriate message to the server. Based on this, the server removes possible read and write locks and applies both committed updates to all associated caches that host the `Stock` and `Cart` tables. For this, it has to perform a two-phase commit (2PC) with all the caches. Once a running transaction at these caches has been interfered with the updates, it must be aborted in order to guarantee consistency of the cache data.

Note that the execution at the cache might be aborted by the server before it is completed. This might happen when another cache commits an update that interferes with query $\boxed{\text{A}}$, or more precisely, that changes the result of query $\boxed{\text{A}}$ at the cache. Updates are propagated to caches by a 2PC protocol. Once such an update occurs, the execution at the cache must be aborted, since otherwise a consistent state cannot be guaranteed.

Let us have a look at the resulting total processing time of the procedure (PT = processing time) and the work load at the server (ST = server load). For this, we use *Net* as network latency (one direction) and *WP* as the time the server takes to give write permission. Note that *WP* is usually implemented by locking. Thus, it causes some delay at the server, but does not require many computational resources. Furthermore, we use U, S, A, I as the processing time of the SQL operations and use subscripts C and D to indicate whether the SQL operation has been executed at the cache or the server. We ignore the execution time of the code in between the SQL operations and assume that `AddCart` commits properly. The server's committing and update propagation time is denoted as $Commit_{2PC}$. According to Figure 3.1, the run and server load is approximated as follows:

$$\begin{aligned} PT_{avoid} &= Net + WP + Net + U_C + Net + S_D + Net + \\ &\quad A_C + Net + WP + Net + I_C + Net + Commit_{2PC} + Net \\ &= U_C + S_D + A_C + I_C + 8\ Net + 2\ WP + Commit_{2PC} \qquad (3.1) \\ ST_{avoid} &= U_D + S_D + I_D + 2\ WP + Commit_{2PC} \qquad (3.2) \end{aligned}$$

In the best case the server gives write permission ($WP \approx 0$) immediately and during the commit process all updates can be applied on remote caches without aborting the transactions. However, in the worst case the server can not give write permission ($WP > 0$), since data is accessed (e.g. locked) by other transactions. Furthermore, a commit might abort several transactions on remote caches, when their current execution is interfered with committed updates.

We use this approximation later on in Section 3.3 to compare the avoidance-based execution protocol with the simultaneous execution protocol.

3.2.2 Detection-Based Protocols

Detection-based protocols, on the contrary, allow the cache to operate on state data. Hence, the result of a local query might be inconsistent. Unlike avoidance-based protocols, detection-based protocols require a validation check for each local read operation. Prior to a read operation the cache connects the server and validates and possibly updates local data. The server is responsible for guaranteeing that this data is valid until the transaction commits.

Figure 3.2 depicts a possible execution of the `AddCart` procedure by such a protocol. The

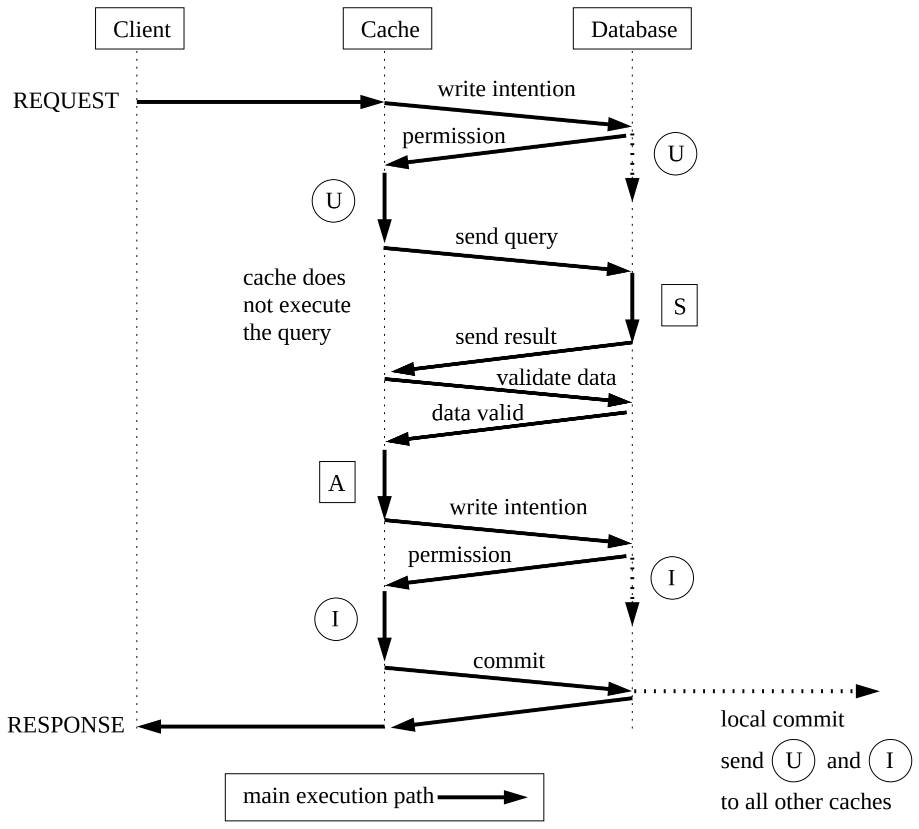


Figure 3.2: Example of a detection-based concurrency protocol.

detection-based protocol differs from the avoidance-based protocol in Figure 3.1 as follows:

1. The validity check prior to query $\boxed{\text{A}}$ requires additional network communication and a time VC at the server to validate and possibly to update cache data if data at the cache is not up-to-date.

2. The transaction is only locally committed at the cache and the server, but not at other attached caches. Hence, no 2PC is performed on connected caches.

3. The update propagation after committing can be done in an asynchronous manner. That is, updates are sent to remote caches and applied without further committing which saves the expensive 2PC between caches.

The total processing time of a procedure (PT) and the server load (ST) can be approximated by

$$\begin{aligned} PT_{detect} &= Net + WP + Net + U_C + Net + S_D + Net + Net + VC + Net + \\ &\quad A_C + Net + WP + Net + I_C + Net + Commit_{Local} + Net \\ &= U_C + S_D + A_C + I_C + 10\ Net + 2\ WP + VC + Commit_{Local} \qquad (3.3) \\ ST_{detect} &= U_D + S_D + I_D + VC + 2\ WP + Commit_{Local} \qquad (3.4) \end{aligned}$$

We use $Commit_{Local}$ to indicate that both protocols implement different committing schemes which potentially lead to different run time behavior. Since the VC might include the shipping of data to the cache, it also affects the work load ST_{detect} of the server.

Again, we use this approximation later on in Section 3.3 for comparing the detection-based and simultaneous protocol.

3.2.3 Discussion

After presenting avoidance- and detection-based protocols, the question arises immediately: Which protocol performs best? This cannot be answered in general and depends on many influencing factors. Intuitively, the processing time of the detection-based protocols seems to be higher, due to additional communication for validation checks. However, in our rough approximation the server run time differs only in the $Commit_{2PC}$ (avoidance-based) and $VC+Commit_{Local}$ (detection-based). When analyzing these values it is necessary to compare the overhead in $Commit_{2PC}$ caused by the 2PC and the overhead caused by additional validity checks (VC). We again point to [40] where a number of avoidance- and detection-based protocols have been experimentally analyzed.

Leaving this problem aside, one can make the following observations about both types of protocols that point us towards parallelism in database caching.

O1. In order to keep data consistent, updates have to be applied on cached and server data anyhow, such that a stored procedure with updates naturally imposes cache-server communication.

O2. An execution of a procedure at the cache handles resulting SQL operations in a pure sequential manner. Those methods do not take into account that the procedure in most cases executes multiple SQL operations as the four in our example.

O3. In order to guarantee transactional consistency, both protocols implement a concurrency control mechanism that leads to high dependency on the server. In fact, a cache must be located close to the server, otherwise a high latency slows the concurrency control down further.

O4. Except SQL operations, the procedure code is executed entirely at the cache. Note that the code is executed in the main memory and that the major overhead results from IO traffic from executing SQL operations.

We favor the detection-based protocols, since in general a 2PC is expensive and leads to a potentially high overhead during committing transactions between all the caches. For a high number of caches the 2PC would be one of the central bottle necks.

3.3 A Simultaneous Execution Scheme

For the given example, we discuss a simultaneous execution scheme and look at how a potential improvement can be achieved by comparing the processing time PT and the server load ST. For this we also take the observations O1 - O4 of Section 3.2.3 into account.

3.3.1 Twin Transactions

Our first key concept is *twin transactions* where cache and server execute the same transaction in parallel. We elaborate on this concept and present a simple execution scheme. Note that the concept of *twin transactions* is also used by [45, 87] in mobile transaction processing. However, they use it in a sequential manner where a transaction is first executed at a mobile host and later at the central server. We use the term to state that both transactions are running in parallel at the cache and the server. Additionally, we use the term *simultaneous* to indicate that both transactions run in parallel. In Section 3.3.2 we extend this scheme to our simultaneous execution scheme.

Since updates have to be performed in any case at the database (observation O1) and the transactional consistency requires a tight coupling between the cache and database (observation O3), we propose that besides the procedure at the cache, the same procedure is executed at the database. Thus, both procedures are executed independently and in parallel — one at the cache and one at the server. Figure 3.3 shows the corresponding twin transactions for our example. Note that due to different hardware both transactions might require a different execution time. Since we prefer to avoid a 2PC, we adhere to a detection-based scheme and allow inconsistent data at the cache.

The synchronization of both transactions is handled as follows. The primary goal is data consistency at the central database. When the server completes the execution of the transaction, it commits updates on server data independently of the execution at the cache. After the commit, the server sends a completion message to the cache which stops its execution, if still running, and rejects (undoes) all local applied updates. The server applies all committed updates to all the other remote caches in an asynchronous manner (no 2PC), including the cache that has rejected its updates. Intuitively, the execution scheme guarantees a consistent

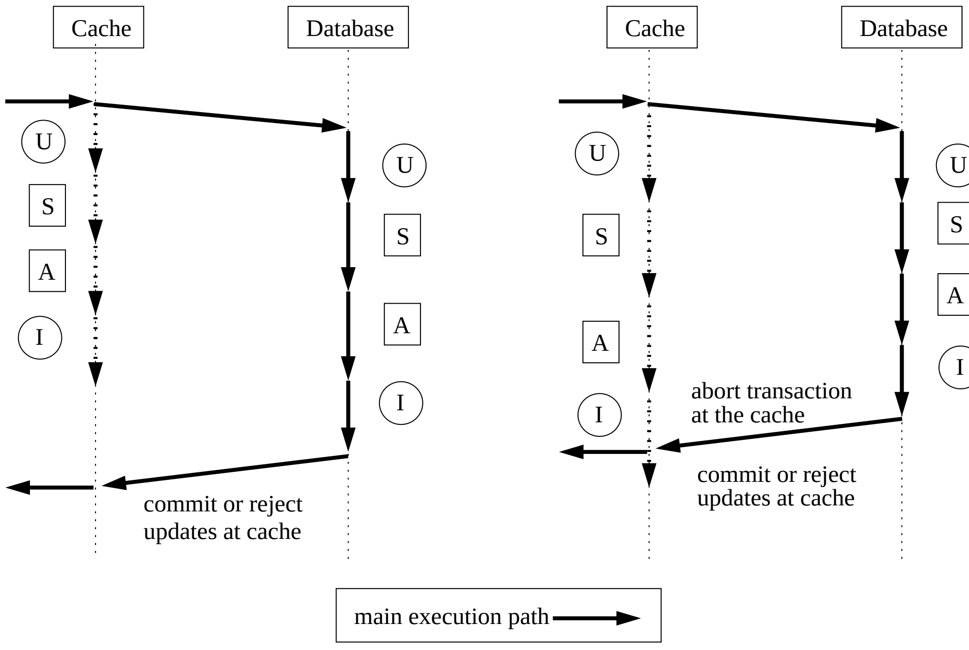

Cache
Database
Cache
Database
U
S
A
I
U
S
A
I
commit or reject updates at cache
U
S
A
I
U
S
A
I
abort transaction at the cache
commit or reject updates at cache
main execution path

Figure 3.3: Twin Transactions. Left: fast Client. Right: slow Client.

database and synchronization of cache data. Note that the scheme can be further optimized, e.g. by committing updates also at a cache instead of rejecting them.

Clearly, the execution at the cache is redundant and the performance does not benefit from the cache, since all queries are executed at the server. In the following section we extend this primitive scheme with a partial execution of the procedure code and the server-side re-use of the query results.

3.3.2 Partial Execution and Re-Use of Query Results

The other key concepts of our approach are

- a partial execution of procedures at the cache,
- the validation and re-use of the query results and
- independent queries in the procedure code.

In contrast to observation O4 (the server sends query results to the cache), we use the opposite direction where the client delivers the query results (based on local executions) that are then re-used in the execution of a transaction at the server. This is useful, since then the cache is not involved in concurrency control and is only responsible for providing query results. Again, according to the original spirit of database caching, queries are outsourced from the server and performed at a cache.

Figure 3.4 depicts one possible execution of the `AddCart` in our simultaneous execution scheme. We have chosen an example with similar behavior to avoidance- and detection-based

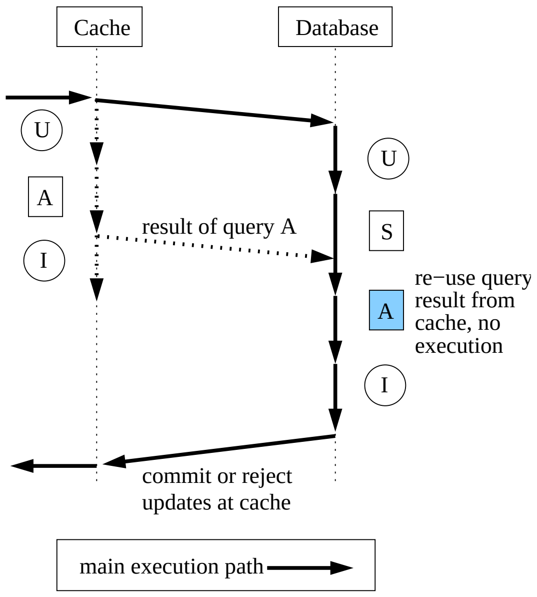

Cache
Database
U
A
I
result of query A
U
S
A
re−use query result from cache, no execution
I
commit or reject updates at cache
main execution path

Figure 3.4: Partial execution and reuse of query results.

protocols where query $\boxed{S}$ is executed at the database and query $\boxed{A}$ at the cache. In detail the execution behaves as follows. We first elaborate the partial execution at the cache and then the execution at the server. After the procedure has started at the cache, it behaves as follows:

1. Execute update $\textcircled{U}$.
2. Do not execute query $\boxed{S}$. Hence the local variable `session` (code line 2) is not assigned and the whole `IF` statement (code line 3-4) is not executed, since the condition of the `IF` statement cannot be evaluated.
3. Execute query $\boxed{A}$ in code line 5 and send the result of the query to the central database. Note that the query does not depend on previous statements in lines 1-4, but only on the input variables. Assign the query result to the local variable `inStock` and proceed.
4. Execute lines 6-7 and the update $\textcircled{I}$ in line 8.
5. Complete the execution in line 9.

Intuitively, this can be seen as a partial execution of the procedure code where some of the SQL statements are left out. We define the partial execution model in Part II.

In parallel to the execution at the cache, the server behaves as follows:

1. Execute update $\textcircled{U}$, query $\boxed{S}$ and the entire `IF` statement (code lines 1-4). Note that in the meantime the cache is executing update $\textcircled{U}$ and query $\boxed{A}$.

2. Before executing query $\boxed{\text{A}}$, check whether the cache has delivered a result for the query and, if so, check whether the query result is valid. Otherwise wait for the result.

   (a) If it is valid, re-use the result and assign it to the local variable `inStock` in line 5.

   (b) If the result is invalid, or no result has been delivered, execute the query and assign the result to the `Stock`.

3. Proceed with lines 6-9, including update $\textcircled{\text{I}}$ and commit.

4. Send updates $\textcircled{\text{U}}$ and $\textcircled{\text{I}}$ to all associated caches in the client-server database system.

Clearly, the key issues to make the approach work are an efficient validation technique, a partial execution model for stored procedures, the existence of independent queries, and delivery on-time. The latter is especially important, which leads to discussions about whether the database should wait for a cache result and, if so, how long it should wait. We discuss the problem further in Part II.

Let us have a look at the resulting total processing time (PT) and the server load (ST). We use the same notations as those in Section 3.2. Similarly, as in the executions of procedures at the cache, we ignore the extra time for executing the procedural code at the database (excluding the execution of SQL statements).

Additionally, we use $\mathit{verify}(A)$ to indicate the execution time for checking whether the result of query $\boxed{\text{A}}$ delivered by the cache is valid, and $re(A)$ to indicate the possible re-execution of an invalid query at the central database. Note that $re(A) = 0$ for a valid query and $re(A) = A_D$ for an invalid query. Furthermore, we represent the idle time (waiting time for the query results) by $\mathit{idle} = A_C - S_D$. Note that the server cannot wait a negative idle time. A negative idle time states that the result of the cache has been delivered before the server is accessing it. Hence, a negative idle time represents no idle time. Clearly, as we will show in Part III, our scheme tries to maximize the re-use of the query results and to minimize the idle time.

$$\begin{aligned} PT_{sim} &= U_D + S_D + \mathit{verify}(A) + re(A) + I_D + 2\ \mathit{Net} + \\ &\quad 2\ \mathit{WP} + \mathit{idle} + Commit_{Local} \qquad (3.5) \\ ST_{sim} &= U_D + S_D + \mathit{verify}(A) + re(A) + I_D + 2\ \mathit{WP} + \\ &\quad Commit_{Local} \qquad (3.6) \end{aligned}$$

We also use $Commit_{Local}$ here, since our commit equals the one used by detection-based protocols. Note that the server time does not include the idle time, since this does not occupy computational resources.

Let us compare the processing time of the simultaneous scheme with those of the avoidance- and detection-based protocols. Note that all approximations reflect best cases without abort and re-scheduling of transactions. As a simplification for this comparison we also assume that the execution time of the updates is similar at the cache and the database ($U_C \approx U_D$, $I_C \approx I_D$).

$$\begin{aligned}
PT_{avoid} &> PT_{sim} \\
U_C + S_D + A_C + I_C + 8\ Net + &> U_D + S_D + \mathit{verify}(A) + re(A) + I_D + \\
2\ WP + Commit_{2PC} & 2\ Net + 2\ WP + idle + Commit_{Local} \\
A_C + 6\ Net + Commit_{2PC} &> \mathit{verify}(A) + re(A) + idle + Commit_{Local}
\end{aligned}$$

As we show in Part II, the validation check can be done fast and therefore we assume $\mathit{verify}(A) \approx 0$. Furthermore, we assume that $Commit_{2PC} = Commit_{Local} + c$ with $c > 0$, since in general a 2PC is more costly than a local commit. Hence, our scheme reduces the execution time and we obtain:

$$\begin{aligned}
PT_{avoid} &> PT_{sim} \\
\text{[Case A: valid query } re(A) = 0, idle = 0] & \\
A_C + 6\ Net + c &> 0 \\
\text{[Case B: valid query } re(A) = 0, idle > 0] & \\
S_D + 6\ Net + c &> 0 \\
\text{[Case C: invalid query } re(A) = A_D \approx A_C, idle = 0] & \\
6\ Net + c &> 0 \\
\text{[Case D: invalid query } re(A) = A_D \approx A_C, idle > 0] & \\
S_D + 6\ Net + c &> A_C
\end{aligned}$$

In the case that our scheme re-uses the query results, it performs at least A_C (or S_D for $idle > 0$) faster than the avoidance-based protocols, since query $\boxed{\text{A}}$ is executed in parallel at the cache and therefore does not appear on the main execution path (see Figure 3.4). In case of a re-execution of a query, we still save the network communication (*Net*) and possible delays that are caused by giving write permissions (*WP*). Our scheme only takes more time (Case D) if a very slow performing cache is used. Then, the execution of A_C can take more time than S_D and network communication.

We have also compared our scheme to the detection-based protocol. Note that both schemes use the same commit mechanism and $\mathit{verify}(A) \approx 0$.

$$\begin{aligned}
PT_{detect} &> PT_{sim} \\
U_C + S_D + A_C + I_C + 10\ Net + 2\ WP + &> U_D + S_D + \mathit{verify}(A) + re(A) + I_D + \\
VC + Commit_{Local} & 2\ Net + 2\ WP + idle + Commit_{Local} \\
A_C + 8\ Net + VC &> \mathit{verify}(A) + re(A) + idle
\end{aligned}$$

In the best case, cache data is up-to-date and the validation check of the detection-based protocol does not need to ship fresh data to the cache ($VC \approx 0$) and our scheme re-uses the query result ($re(A) = 0$). In the worst case, cache data is not up-to-date and the server has to update the data in the cache ($VC > 0$) and our scheme has to re-execute the query ($re(A) = A_S \approx A_C$). Then, we obtain

$$\begin{aligned}
PT_{detect} &> PT_{sim} \\
\text{[Case A: valid query } re(A) = 0, idle = 0] \\
A_C + 8\ Net &> 0 \\
\text{[Case B: valid query } re(A) = 0, idle > 0] \\
S_D + 8\ Net &> 0 \\
\text{[Case C: invalid query } re(A) = A_D \approx A_C, idle = 0] \\
8\ Net + VC &> 0 \\
\text{[Case D: invalid query } re(A) = A_D \approx A_C, ilde > 0] \\
S_D + 8\ Net + VC &> A_C
\end{aligned}$$

Again, we notice that in the case of a re-use of a query result, our scheme performs faster. In the case of an invalid query result at least the network communication and the validation check is saved.

Similarly, we have compared the server load, which reflects the server's load. Again, we assume that $Commit_{2PC} = Commit_{Local} + c$ with $c > 0$ and $verify(A) \approx 0$. In the following we use the symbol $X <> Y$ do denote that $X < Y$, $X = Y$ or $X > Y$ holds.

$$\begin{aligned}
ST_{avoid} &> ST_{sim} \\
U_D + S_D + I_D + 2\ WP + Commit_{2PC} &> U_D + S_D + verify(A) + re(A) + I_D + \\
&\quad\ 2\ WP + Commit_{Local} \\
\text{[Case A: valid query } re(A) = 0] \\
c &> 0 \\
\text{[Case B: invalid query } re(A) = A_D \approx A_C] \\
c &<> A_D
\end{aligned}$$

Note that the server load does not include the idle time which does not occupy computational resources.

For a valid query our approach is preferable, since we can assume in general that the 2PC is less efficient than a local commit. In case of an invalid query, both approaches could be equally useful, depending on the complexity of the query and the number of caches involved in the 2PC protocol. However, our scheme tries to minimize the amount of query re-executions (Case B). Furthermore, our approach does not abort transactions as a result of an update propagation, as in the case of avoidance-based protocols.

The comparison with the detection-based protocol shows that our approach leads to a similar work load at the server.

$$\begin{aligned} ST_{detect} &\approx ST_{sim} \\ U_D + S_D + I_D + VC + &\approx U_D + S_D + \mathit{verify}(A) + re(A) + I_D + \\ 2\ WP + Commit_{Local} & \quad 2\ WP + Commit_{Local} \end{aligned}$$

Again, we have compared the best and worst cases. As stated earlier, consistent cache data cause an efficient validation check ($VC \approx 0$) and a re-use of the query result ($re(A) = 0$). Inconsistent cache data, however, causes the server to ship data to the cache ($VC > 0$) and in our scheme the re-execution of the query ($re(A) = A_D \approx A_C$). Therefore we obtain

$$\begin{aligned} ST_{detect} &\approx ST_{sim} \\ \text{[Case A: valid query } re(A) = 0\text{]} & \\ VC &\approx 0 \\ \text{[Case B: invalid query } re(A) = A_D \approx A_C\text{]} & \\ VC &<> A_C \end{aligned}$$

Note again that our scheme tries to minimize Case B. However, for Case B each protocol could outperform the other. $VC < A_C$ could be the case if only a few data have to be shipped to the cache and the query is expensive. $VC > A_C$ could be the case if a huge amount of data has to be shipped to the cache and the query is less expensive.

As shown in a specific example, our simultaneous execution protocol improves the response time (PT) of a request and results in a nearly equal server load (ST). We have drawn this evidence from our studies and experiments that are presented Chapter 6.

3.3.3 The Run Time Optimization Problem

As shown in Figure 3.5 the simultaneous execution scheme can be differently configured. Our scheme can be considered as kind of load-balancing of procedure code execution between cache and server.

Hence, the fundamental question is: Which statements should be executed at the cache and which data should be replicated, such that the resulting execution time is minimal and throughput is maximal with respect to the current load conditions? Recall from Section 2.4 that a caching scheme must consider frequency of access, size of cached data, synchronization costs and server load for solving this problem. Further, it has to adapt its configuration whenever the load conditions changes significantly. Due to this, a cache at run time is responsible for deciding which data to replicate to the local database and, therefore, which queries to execute locally.

In Part III we formulate this problem as a dynamic optimization problem and suggest a partial model-based optimizer for solving it. The challenging issue of this problem is to take the dynamic load conditions of a database cache and server at run time into account.

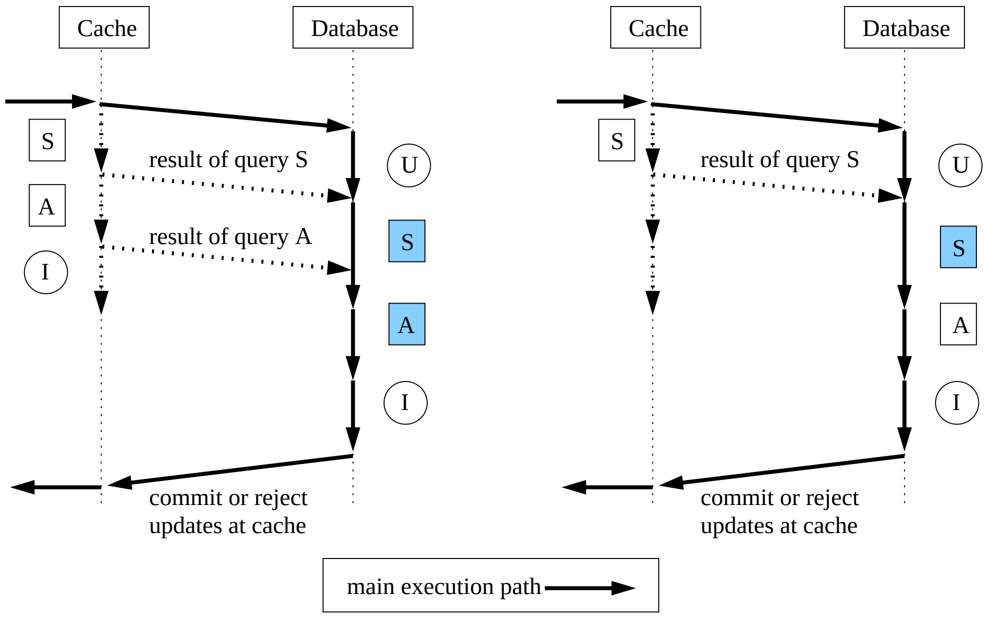


Figure 3.5: Different configurations of the simultaneous execution scheme.

3.4 Summary and Discussion

In general, database caching techniques depend on certain assumptions about the underlying application. Caching is useful under certain circumstances. According to literature these assumptions are:

1. *Locality:* Users of the system access different portions of data. Only under this assumption can a cache be loaded according to the user's data space of interest and therefore does not contain the full database.

2. *Rare Conflicts:* When data is cached and a query is executed locally, it should also be valid (consistent) in most of the cases. This is only possible if the amount of updates is low.

3. *Query Size:* Complex queries are sent directly to the server and not handled by the cache, since they often operate on a large data set that is not necessarily available at a database cache. Hence, the presented caching techniques are not useful for data mining applications.

4. *Reliable Network:* Due to the high dependency on the server that is caused by the concurrency control protocol, caches should be located close to the server and be connected by a reliable network. In our opinion all of the above concurrency control protocols are not useful for unreliable and high-latency networks, such as the Internet and wireless networks. For these classes we have to weaken the transactional properties, as studied in [71, 82].

5. *Fat Clients:* Caching is only useful if a cache can execute a query efficiently.

Considering the above example, our approach also relies on the following assumptions:

1. *Stored Procedures:* Our scheme is only useful for stored procedures that implement a single transaction that consists of several SQL operations. It is not advantageous if only simple operations are involved. Additionally, the partial execution of procedures requires that the entire code is known in advance. Hence, the simultaneous execution cannot be applied to adhoc queries.

2. *Server-Side Result Cache:* Since caches ship query results to the server, we rely on an efficient server-side management for query results. For this, we introduce a query result cache at the server that is responsible for maintaining intermediate query results from caches. As a consequence, our approach is only useful in fast networks with a high bandwith.

3. *Compiled Time Analysis:* Possible partial executions and the identification of simultaneously executable queries requires a compiled time analysis of the procedure code.

Our approach can be seen as an extention of detection-based protocols by reducing communication overhead and adding a new level of parallel executions. Independent queries can be executed by cache and server in parallel which possibly reduces the total execution time. Recall that the cache just sends the query result to the server and (without waiting) immediately continues with processing the procedure code. Hence, the verification and the possible re-execution is done in the meantime in parallel by the server. As we will show in several experiments, the new level of parallel executions further improves the performance.

According to the criteria of Section 2.5, we classify the proposed approach as shown by Table 3.1. As pointed out be the previous sections, our main contribution results from considering *Multi-Requests* and an integration of database caching into the *Execution Engine* of a database management system.

Criteria	Property	Notes
Target Host	client-side	We assume that each client might differ in its resources (CPU, disk, network, load) which has to be considered by the caching technique.
Transparency	yes	-
Hierarchy Level	none	-
Data Granularity	tuples	-
Grouping	horizontal table fragments	We have chosen a simple technique, since data granularity is not the main field of contribution.
Local, Mixed, Remote Execution	local and remote	We do not explicitly consider the split of queries.
Single- or Multi-Request	Multi-Request	We consider a single stored procedure as request. Each procedure normally consists of multiple read and write operations.
Concurrency Control	extended detection-based protocol with parallel verification, re-execution and simultaneous query processing	-
Load and Refresh	asynchronous, push-based, run time adaptive	Cache data is exclusively updated by the server.
Handling of Updates	combined	Cache and server directly apply updates on local data.
Integration Level	execution engine	-

Table 3.1: Characteristics of the proposed execution scheme.

Part II

Split Twin Transactions for Database Caching

Chapter 4

A Client-Server Database System for Twin Transactions

This Chapter defines a client-server database system with data replication to database caches at clients and the simultaneous execution of a stored procedure at the database cache and the server without code splitting. For this we define the concept of *twin transactions* as a preliminary stage of our novel execution scheme that, based on the underlying replication scheme, allows to define conditions for which both executions are equal, t.i. have the same effect on cache and server data. In detail we use the executed sequence of elementary database operations, e.g. updates and queries, of stored procedures and the version number of table fragments that are maintained by the replication scheme. On top of those fundamental conditions we derive in Chapter 5 an efficient validation technique for client computations as motivated in Part I. Further, we will extend in Chapter 5 the concept of *twin transactions* by code splitting, such that the cache and the server share the execution of a stored procedure.

This Chapter is structured as follows: We first give an overview and state the desired properties of such a client-server system. Then, we define the stored procedure language, the execution engine, table fragmentation, version management and data synchronization. Finally, we define *twin transactions* and the above mentioned fundamental conditions that characterize the equal execution of a stored procedure at the database cache and server.

4.1 Overview

Consider a central server which is connected to multiple distributed clients. We assume that the server is a standard database system with a data dictionary, a query processor, a scheduler with concurrency control, a code execution engine, etc. We will only further define these parts as they affect our approach. The server provides its functionality to clients *only* by stored procedures. We do not consider client-side ad-hoc queries to the server, as these can always be encapsulated in a parameterized procedure. In this setup, a client initiates a procedure call, sends it to the server which executes it and passes the result back to the client.

To such a system we apply the following extentions:

- Each client runs a standard database system that is used for caching server data and executing stored procedures. Those database systems do not require advanced features, such as multi-processor capabilities, recovery, or multi-user support. In following we call those databases also *client databases*.
- For distributing data, tables at the central server are partitioned into fragments. A subset of these fragments is replicated at associated client databases.
- Synchronization components at the server and associated clients are responsible for updating replicated fragments. This is done by an optimistic log-transfer method as described in Section 2.4. Hence, there might be a delay in propagating changes from the server to client databases, so that those databases possibly operate on stale data.
- Replication is dynamic, so that the amount of replicated fragments can be adapted at run time to meet the data interests of a client.
- Whenever a stored procedure is triggered at a client, it is immediately forwarded to the server and both the client database and the server execute the procedure simultaneously . As presented in Chapter 3, this corresponds to our notion of *twin transactions*. For this the server and associated clients are extended by special execution engines.

According to those extensions a *database cache* is defined by a standard database system, a replication compontent and an execution engine for twin transactions, c.f. Figure 4.1. In

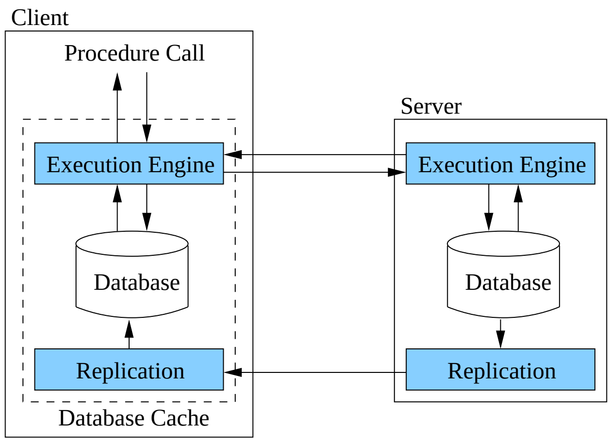


Figure 4.1: Architecture of a Database Cache.

subsequent chapters we further extend this definition.

Desired Properties of the Reference System

To perform the extentions, our approach requires certain properties of the underlying database system. We summarize them and give our motivations for using them. Primarily, we are interested in integrating our approach into existing database systems using very little effort.

- **Determinism**
  To obtain equal executions, a fundamental requirement is determinism. That is, equal procedure code has to produce the same effect on data at a cache and the server. For this, we place some restrictions on the stored procedure language. Furthermore, we require equal data schemata of caches and the central server, including table definitions, integrity constraints, stored procedures, etc.

- **Version Management**
  Another requirement for equal executions is the equality of cache and server data concerning the individual access of a stored procedure. To efficiently check this in our optimistic synchronization scheme, we attach a version number to a fragment. We develop a version management and synchronization scheme, so that a fragment at cache and server with identical version number is also of equal content.

- **Low-Level Updates**
  As proposed by literature (see Section 2.2), database caching should be done on a tuple basis. This is useful for propagating elementary updates to caches and to detect the individual tuple access of SQL statements. For this, we translate complex database updates, such as `UPDATE Stock SET amount=amount+x WHERE productGroupID=29`, into a query and a loop that for each record of the query result performs an update of a single tuple.

- **Side-Effects**
  A stored procedure is a piece of procedural code that is executed at the database. However, the data schema often contains enhanced concepts for integrity constraints (e.g. keys, uniqueness constraints, tuple constraints, rules, triggers, etc.). These concepts define additional database behavior that does not explicitly appear in the procedure code. These constraints have to be considered, since they might access and modify data.

We refer to these properties throughout this Chapter.

Notations

Below we list some naming conventions and common notions that are used in this and following chapters.

- For running indexes we use i, j, k, l, m.
- $\mathcal{C}$ denotes the set of all clients. Each client is denoted by $C \in \mathcal{C}$ or $C_i \in \mathcal{C}$ with $1 \leq i \leq |\mathcal{C}|$. We also use C and C_i for the database cache at a client.
- $\mathcal{S}$ denotes the set of all stored procedures. Each procedure is denoted by the letter S with $S \in \mathcal{S}$ or $S_j \in \mathcal{S}$ with $1 \leq j \leq |\mathcal{S}|$.

- $\mathcal{R}$ denotes the set of all tables at the server. Each table is denoted by $R \in \mathcal{R}$ or $R_k \in \mathcal{R}$ with $1 \leq k \leq |\mathcal{R}|$. A table R consists of columns or attributes $A_1, \ldots, A_n$ of different types. The domain of an attribute A is denoted by $dom(A)$. An instance of a table R is a subset $r \subseteq dom(A_1) \times \cdots \times dom(A_n)$ of tuples $t \in r$. The value of an attribute A in a tuple t is denoted by $t(A)$.

- For table instances r, we use relational algebra expressions to define table fragments. A projection is denoted by $\pi_A(r)$ and represents a set of all values of the column A of table r, that is $\pi_A(r) = \{t(A) \mid t \in r\}$. A selection is denoted by $\sigma_\varphi(r)$ and represents all tuples in r that satisfy the condition φ, that is $\sigma_\varphi(r) = \{t \mid t \in r \wedge \varphi(t) = true\}$.

4.2 Data Definition and Stored Procedures

We define the structure of tables and the stored procedure language. To avoid non-determinism and side-effects, we pose certain restrictions on the language. This is useful in order to keep the client-server system simple. At the end of this chapter, we briefly discuss how these restriction may be relaxed.

4.2.1 Tables and Constraints

A table R consists of attributes $A_1, \ldots, A_n$ of different types. Note that we ignore the different types, since they do not affect our work. By default, each table has a column `ID` which represents the primary surrogate key containing tuple identifiers. The surrogate key is read-only and maintained by the database management system. Tables always correspond to user or application tables and not to internal tables of the system. When we refer to a system table, we explicitly name it as such.

We assume that there are no further constraints defined in the schema, e.g `NULL`, `DEFAULT`, keys, etc., and no advanced concepts, such as triggers, sequences, etc. Hence, additional requirements for data integrity have to be implemented within the code of stored procedures. As an example, we will discuss the handling of uniqueness constraints below.

4.2.2 The Language of Stored Procedures

We assume that there is a stored procedure language like pgplsql used in PostgreSQL ([85]) or PL/SQL used in Oracle. Recall that the database provides its functionality *only* in terms of stored procedures to users and applications. The syntax of a procedure is:

```
CREATE FUNCTION <name>(<param-list>)
RETURNS <type> AS
  <head>
BEGIN
  <body>
END
```

A stored procedure S consists of a name `<name>` and a set of parameters `<param-list>`, each of some type. Beside the basic types, we also support a table type as return value. It is denoted `SETOF ROW` and allows a procedure to return single tuple or a set of tuples. A procedure call is the execution of a procedure for a given set of input parameters.

The head defines parameters and local variables:

```
DECLARE <var> ALIAS FOR $i;
DECLARE <var> AS <type>;
```

The first statement defines a local variable `var` for the i-th input parameter and the second statement defines a local variable `<var>` of type `<type>`. Again we allow basic types and a tuple type, denoted `ROW`.

The body contains the procedural code. Primitive statements are:

- `UPDATE` R `SET A=<expr> WHERE <cond>`,
- `INSERT INTO` R `VALUES (<expr>,..,<expr>)`,
- `DELETE FROM` R `WHERE <cond>`,
- `<var> := (SELECT <expr> FROM` $R_1, \ldots, R_n$ `WHERE <cond>)` that assigns[1] the result of the query to a local variable,
- the assignment `<var> := <expr>`, where the value of the expression is assigned to the variable,
- the fail operator `RAISE EXCEPTION` that aborts the execution of a procedure and undoes all applied updates,
- a return operator `RETURN <expr>` that terminates the execution and returns the evaluated expression,
- a return operator `RETURN NEXT <expr>` that does not terminate the execution and adds the evaluated expression to the list of returned tuples.

Primitive statements are denoted as `<stmt>`. Composite statements `<stmts>` are:

```
<stmts> ::=
    <stmt> | <stmt>;<stmts> |
    IF <cond> THEN <stmts> ELSE <stmts> END IF |
    IF <cond> THEN <stmt> |
    FOR <var> IN <query> LOOP <stmts> END LOOP
```

[1]Commercial database systems us the notation `SELECT <expr> INTO <var> FROM` $R_1, \ldots, R_n$ `WHERE <cond>` to assign a local variable `<var>`.

The second `IF` statement is an abbreviation, if only one statement is used in the `THEN` part. For `<query>` we allow any `SELECT <expr>,..,<expr> FROM` $R_1, \dots, R_n$ `WHERE <cond>` statement with subselects `<expr>` and `<cond>`. The loop statement uses a local variable `var` of the tuple type `ROW`. The expressions `<cond>` and `<expr>` are restricted to contain only

- local variables,
- constant values,
- basic operators according to the types used and
- column names of tables of the form $R.A$ with R as table and A as column.

This prevents procedures to call other procedures or internal functions that are provided by the database system (e.g. build-in functions). Hence, a procedure only accesses tables as they occur in the procedure code.

4.2.3 Low-Level Procedures

As explained above, the procedure code contains `INSERT`, `DELETE`, `UPDATE` statements that might affect multiple tuples of the underlying tables. To handle data modifications on a tuple basis, we translate these statements into a loop and low-level updates.

Low-Level Database Updates

Low-level updates operate on a tuple basis. For a table R, a tuple t and a tuple identifier tid, low-level database updates are:

- $tid = insert(R, t)$ with tid as the new generated identifier,
- $tid = insert(R, tid, t)$ with tid as the given tuple identifier and
- $t = delete(R, tid)$ with t as the deleted tuple.

The unique tuple identifier tid is created by $insert(R, t)$. Its creation is defined later on in Section 4.7. The identifiers are kept in the read-only `ID` column of tables. Note, that we might use low-level updates without considering their return value.

Translating Update Statements

The statements `INSERT`, `DELETE`, `UPDATE` only modify tuples of a single table R. The idea is to replace each of them by: (1) a query that computes which tuples (including their tid) are inserted or deleted and (2) a loop over these tuples that performs the corresponding low-level updates.

For this, we assume that the query on a single table also returns the tuple identifier. We use `SELECT ID,* FROM` R to indicate that the tuple identifier is part of the query result. Note that in general queries on multiple tables cannot provide the tuple identifier, since (e.g.

a join) they mix up tuples. The result of each row is stored in a local variable *row*. We use the same syntax $row.ID$ and $row.A_j$ to access an attribute of a tuple. For a table R with n attributes A_j $(1 \leq j \leq n)$, we provide the translation of the `INSERT`, `DELETE` and `UPDATE` statements.

`UPDATE` R `SET` A_j`=<arith-expr> WHERE <cond>` is translated into:

```
FOR $row$ IN SELECT ID,* FROM $R$ WHERE <cond> LOOP
    $row.A_j$ =<arith-expr'>;
    $delete(R, row.ID)$;
    $insert(R, row.ID, (row.A_1, \ldots, row.A_n))$;
END LOOP
```

Where `<arith-expr'>` results from replacing each A_j in `<arith-expr>` by $row.A_j$.

`DELETE FROM` R `WHERE <cond>` is translated into:

```
FOR $row$ IN SELECT ID,* FROM $R$ WHERE <cond> LOOP
    $delete(R, row.ID)$;
END LOOP
```

`INSERT INTO` R `VALUES (<expr>,..,<expr>)` is translated into:

```
$row$:=(<expr>,..,<expr>);
$insert(R, row)$;
```

The pre-compilation can be done at the creation-time of a procedure. We assume that the resulting low-level procedure is kept within the code base of the caches and the server. In the following we only consider low-level procedures for developing our approach. However, for simplicity we also use the original code within examples.

An Example for Handling of Unique Constraints

As an example of handling schema constraints, that affect the execution of stored procedure code, we discuss uniqueness constraints. Given a table R with the columns $A_1, \ldots, A_n$, a uniqueness constraint on A_i $(1 \leq i \leq n)$ requires that none of the values in A_i occur twice for all instances r of R.

$$\forall t, t' \in r : t(A_i) = t'(A_i) \Rightarrow t = t'$$

A uniqueness constraint can only be violated by insert operations and not by delete operations. As in the above transformations, insert operations occur as $insert(R, t)$ or $insert(R, tid, t)$. Let u denote both types of inserts and $t = (v_1, \ldots, v_n)$. Then, u is replaced by

```
IF (EXISTS SELECT $A_i$ FROM $R$ WHERE $A_i$=$v_i$) THEN
          RAISE EXCEPTION;
ELSE
          $u$;
END IF;
```

If the value v_i already exists in column A_i the update u is rejected. Otherwise the tuple is inserted. Note that replacing the insert operations by handling uniqueness constraints can be optimized further. If, for example, an `UPDATE` statement does not change the column A_i; the replacement is not necessary.

4.3 The Execution Engine

We assume that the standard database systems at the clients and the server implement a concurrency control mechanism for guaranteeing the execution of transactions, and that both use the same type of execution engine for stored procedures. Concurrency control applies to application tables $R_1, \ldots, R_n$ and system tables. A procedure is always executed within a transaction. We assume that each transaction is performed in three phases:

1. Starting Phase: The transaction has been scheduled, but no updates have been applied to data yet.

2. Execution Phase: The procedure code is executed and all updates are applied to data.

3. Commit Phase: If no reject has been performed by the code, all updates are committed to data.

In the following, we show how a procedure is executed by the engine and define *execution sequences* for defining the effect of a procedure.

IO Statements

A low-level procedure interacts with data only in terms of low-level updates or queries which we call *IO statements*. We uniquely identify the IO statements within the code of a low-level procedure and use the resulting IDs for multiple purposes throughput this work, e.g., execution sequences, define the logical split of the procedure code, etc.

Definition 4.1 (Uniquely Identified IO statements) *Let S be a low-level procedure. Each query and low-level update in the low-level code of S is called an IO statement and assigned a unique number $s \in \mathbf{N}$. There are no two IO statements with equal identifiers. The set of such numbers is denoted $ID(S)$.*

Execution of Stored Procedure Code

Intuitively, the execution of a procedure S at run time can be seen as follows. While executing the code of S, the engine reaches an IO statement s (low-level update of query). First, the engine instantiates s (resolve local variables, if any) to an executable IO statement, say e. Next, it executes e on data. For this, we assume that the engine is using a function $val = eval(s, \#s, e)$ (the parameter $\#s$ is defined below) that returns the result value val (tuple or tuple identifier for low-level updates, a table for queries). By using val, the engine further processes the code until the next IO statement is reached. And so forth.

In case of an exception, the execution is aborted and no updates are made persistent. Otherwise the execution enters its commit phase, where all updates are permanentely written to data and the execution is completed. In Chapter 5, we model the simultaneous execution scheme mainly by providing different $eval(.)$ functions for caches and the server.

Execution Sequences

We introduce the notion of *execution sequence*, which captures the sequence of executed IO statements of a procedure.

Definition 4.2 (Execution Sequence) *Let S be a low-level procedure. Its execution is captured by the sequence*

$$seq = (s_1, \#s_1, e_1, val_1), \ldots, (s_n, \#s_n, e_n, val_n)$$

with $s_i \in ID(S)$, e_i the instantiated IO statement and $val_i = eval(s_i, \#s_i, e_i)$ the result. The $\#s_i$ denote the number of IO statements s_j with $j \leq i$ and $s_i = s_j$. For queries, val_i is a value, a tuple or set of tuples, and for low-level updates, the tuple identifier or the deleted tuple.

The above execution sequence is used to capture the run time behavior of executions at caches and the server. Note that in an instantiated IO statement e_i, all variables have been assigned. The value $\#s_i$ represents a counter for each s_i that is maintained by the engine and is used to distinguish between different executions of an s_i during a loop. Thus, the pair of numbers $(s, \#s)$ uniquely identifies an element of the sequence. In the case that an execution performs a `RAISE EXCEPTION`, the execution sequence is empty.

In subsequent sections we use execution sequences for maintaining the versions of fragments and for consistently updating replicated data at caches.

Underlying Assumptions

We have defined a primitive language and the execution engine of the client-server system. The assumptions posed on the system are summarized as follows:

Assumption 4.1 *Let S be a low-level procedure and $R_1, \ldots, R_n$ the corresponding tables in the procedure code of S.*

1. *S does only access the tables $R_1, \ldots, R_n$. Hence, there are no side-effects on other tables.*
2. *S is deterministic.*
   (a) *Let $r_1, \ldots, r_n$, $r'_1, \ldots, r'_n$ and $r''_1, \ldots, r''_n$ be instances of the tables. Let $v_1, \ldots, v_m$ be a set of input values for S. Let further $S(v_1, \ldots, v_m)$ be executed twice on $r_1, \ldots, r_n$ in different points of time and r'_i and r''_i the resulting table instances. Then, $r'_i = r''_i$ holds for all $1 \leq i \leq n$. Hence, S is deterministic.*
   (b) *Both executions on the r_i perform the same execution sequence $seq = (s_1, \#s_1, e_1, val_1), \ldots, (s_k, \#s_k, e_k, val_k)$ for $m \geq 0$.*
3. *Each procedure call is executed as transaction that guarantees the ACID properties (Atomicity, Consistency, Isolation, Durability).*

We base our work on this assumption, since otherwise we would have to define the semantics of the procedural language, which is beyond the scope of this work. In the following, we consider only procedures that satisfy these assumptions. Note also that the determinism affects the automatically created tuple identifier. Hence, both executions of S must produce the same tuple identifier for newly added tuples. The maintenance of tuple identifiers is discussed in Section 4.7.

4.4 Table Fragmentation

Tables are partitioned in non-overlapping horizontal fragments. We show how fragments are managed by the server and how fragment access is determined for IO statements (queries and low-level updates). A client and its local database cache is not involved in fragment management. The replication of server data to a client is defined in Section 4.5.

4.4.1 The Definition of Fragments

We assume that administrators have provided a fragmentation column for each table. Then, a fragment is defined as a selection on a table.

Definition 4.3 (Table Fragment) *Let $R \in \mathcal{R}$ be a table, r an instance of R and A a chosen column of R (fragmentation column). The table R is logically split into n $(1 \leq i \leq n)$ non-overlapping fragments*

$$(R, A, c_i) = \sigma_{A=c_i}(r)$$

with $\{c_1, \ldots, c_n\} = \pi_A(r)$. The tuple (R, A, c) is called a table fragment.

The fragmentation is complete if each tuple in R appears in one of the fragments. That is,

$$r = \bigcup_{i=1}^{n} \sigma_{A=c_i}(r) \qquad \text{and}$$

$$\bigcap_{i=1}^{n} \sigma_{A=c_i}(r) = \emptyset$$

holds. According to Definition 4.3 there are $|\pi_A(r)|$ different fragments for a table instance r ($1 \leq i \leq |\pi_A(r)|$). At run time, where operations on tables insert and delete tuples, the number of fragments and the size of each fragment may vary. Note that by taking the surrogate key column `ID` of a table as fragmentation column, the fragmentation can be performed on a fine-grained tuple basis.

4.4.2 Fragment Access of Queries and Low-Level Updates

Before a given IO statement s (low-level update or query) is executed, the system must be able to determine the fragment access of s in advance. This is necessary for the following reasons:

- Before a cache executes an IO statement, it has to figure out if it can be executed on local data. If data is missing, a query would yield a wrong result and an update would not be performed at all.

- In order to load a cache with data, the server has to analyse the specific fragment access of stored procedures.

For queries we detect fragment access by a syntactic analysis of the SQL expression. A query might potentially operate on multiple fragments and/or full tables. Consider a SQL query q on table R in the form

```
SELECT $expr_1, \ldots, expr_n$
FROM $R$
WHERE $\varphi_1$ AND ... AND $\varphi_m$
```

with no sub-selects in any of the expressions $expr_1, \ldots, expr_n$ and conditions $\varphi_1, \ldots, \varphi_m$ in a conjunctive form. Note that by the restrictions of Section 4.2, the expressions and conditions do not contain calls of internal functions nor stored procedures. Hence, these queries only operate on the table R and, in the worst case, the query processor has to scan all tuples of R to answer the query. Let A be the fragmentation column of R and c a constant value. If there exists a φ_i ($1 \leq i \leq m$) with $\varphi_i \equiv (R.A = c)$, the query processor only has to select tuples t on R with $t(A) = c$ and not to scan the entire table R to answer the query. All other tuples with $t(A) \neq c$ do not influence the result of the query. Consequently, the query only operates on the fragment (R, A, c). The same argumentation applies for queries with a `WHERE` clause, which is true.

On the basis of this observation, we present a primitive algorithm that computes the fragment access for a SQL query on multiple tables *including* subselects. To denote full table access, we use the notation $(R, A, *)$ to represent all fragments of a table R.

Algorithm 4.1 (Fragment Access of a Query)

```
Syntax:   $fragDef(q)$
Input:    SQL query $q$
Output:   set of fragments $frag$
Notes:    the WHERE clause of a query is a conjunction of formulae $\varphi_1, \dots, \varphi_n$,
          $A_i$ is the fragmentation column of a table $R_i$

1. $frag = \emptyset$
2. For all SELECT statements sel in q:
   For all tables $R_i$ in the FROM clause of sel:
       If there exists a $\varphi$ in the WHERE clause of sel with $\varphi \equiv (A_i = c)$ for some
       constant c, then add $(R_i, A_i, c)$ to $frag$, else add $(R_i, A_i, *)$ to $frag$.
3. Remove all tuples $(R, A, c)$ from $frag$ if $(R, A, *)$ exists in $frag$.
4. Return $frag$
```

The algorithm passes all SELECT statements and tries to find conditions that restrict the access on the selected tables. Step 3 is important to minimize the size of $frag$, since (R, A, c) is already included in $(R, A, *)$.

Example 4.1 (Fragment Access of Queries) *Consider the following* SELECT *statements and their fragment access. We use tables* Product *fragmented on column* Group *and* Stock *fragmented on some column* A.

1. *Fragment access of the first query is* (Product,Group,'Toys'), *and that of the second is* (Product,Group,*).

```
SELECT * FROM Product WHERE Product.Group='Toys'
SELECT count(*) FROM Product WHERE TRUE
```

2. *Fragment access of the first query is* (Product,Group,'Toys'), *and that of the second is* (Product,Group,*).

```
SELECT * FROM Product WHERE Product.Group='Toys' AND Product.ID=23
SELECT * FROM Product WHERE Product.ID=23
```

3. *Fragment access is* (Product,Group,'Toys'),(Stock,Amount,*)

```
SELECT Product.Name, Stock.Amount FROM Product, Stock
WHERE Product.Group='Toys' AND Product.ID=Stock.ID AND
Stock.Amount>0
```

4. *Fragment access is* (Product,Group,*)

```
SELECT
      Group,
      count(*)/(SELECT count(*) FROM Product WHERE TRUE)
FROM Product
WHERE Product.Group='Toys'
GROUP BY Group
```

The algorithm computes `(Product,Group,*)` *for the first* `SELECT` *and* `(Product,Group,'Toy')` *for the second. In Step 3 the tuple* `(Product,Group,'Toy')` *is removed, since the full query already operates on the full table* `Product`.

□

Especially the second pair of queries in Example 4.1 shows that fragment access can be improved if a query contains a condition on the fragmentation column in its `WHERE` clause.

Fragment access of insert operations can easily be identified, since the inserted tuple contains the value for the fragmentation column. Note that an insert operation has to add a full tuple to a table, due to the non-existence of null values in the database schema.

Let $tid = insert(R, tid, t)$ and $tid = insert(R, t)$ be both low-level updates for adding a tuple t to a table R, and let A be the fragmentation column of R. Then, the updates operate on the fragment $\{(R, A, c)\}$ with $c = t(A)$.

For a low-level update $delete(R, tid)$ we cannot directly derive the accessed fragment on a syntactical basis, since only the tuple identifier tid appears in the statement expression. However, a delete operation results from the replacement of an `UPDATE` or `DELETE` statement by a query and a loop of low-level updates (see Section 4.2.3). According to the replacements, a delete operation always occurs in the form $delete(R, row.ID)$ in the procedure code with $row.ID$ as the tuple identifier tid. Recall that the local variable row contains the values of all columns of R. Hence, the fragment access of the low-level update is determined by $\{(R, A, row.A)\}$.

Definition 4.4 (Fragment Access of IO Statements) *Let S be a low-level procedure and e the expression of an IO statement at run time. For the above notations the fragment access of an insert operation is defined as $fragDef(e) = \{(R, A, c)\}$, for a delete operation, as $fragDef(e) = \{(R, A, row.A)\}$, and for a query, as the result of Algorithm 4.1.*

Clearly, an IO statement can only be executed at a client if the fragments have been replicated. We follow this issue up in Section 4.6, when we define the replication of fragments to clients (Section 4.5).

4.4.3 The Management of Fragments

The server maintains a system table $\mathcal{F}$ for collecting all fragment definitions. It is defined as follows:

Definition 4.5 (Fragment Definition Table) *Let R be a table with the fragmentation column A and let r be an instance of R. The fragment definition table $\mathcal{F}$ contains, for each fragment $(R, A, c_i) = \sigma_{A=c_i}(r)$ with $\pi_A(r) = \{c_1, \ldots, c_n\}$ and $1 \leq i \leq n$, exactly one tuple (id, R, A, c_i, ver) with id as the internal identifier of the fragment and ver as its version number.*

The identifier is automatically generated by the system. The maintenance of the fragment version is discussed in Section 4.4.4. Note that the definition allows tuples (id, R, A, c, ver) in $\mathcal{F}$ with $c \notin \pi_A(r)$ which represent empty fragments. However, $\mathcal{F}$ cannot contain two tuples (id, R, A, c, ver) and (id', R, A, c, ver') with $id = id'$.

For simplicity, we introduce some conventions for the usage of fragments that apply to the remaining part of this work:

- A fragment in $\mathcal{F}$ is identified by the letter F.
- Giving a $F = (id, R, A, c, ver) \in \mathcal{F}$ we also write $F \in \mathcal{F}$. Analogously, we use a function $id(F) = id$ and write F instead of id.
- There is a function $fragID(R, A, c)$ that retrieves the identifier F of the corresponding fragment. If no such fragment is stored in $\mathcal{F}$, the function returns $undef$.
- The content of a fragment F is denoted $data(F)$. We always use $data(F)$ within the context of a table instance r, such that $data(F)$ is an abbreviation for $\sigma_{A=c}(r)$. Note that all tables R contain a column `ID` that contains the surrogate key of a table and is maintained by the database system. Hence, also $data(F)$ contains this column.
- We say *to access/modify a fragment* F, but mean the access/modification of tuples in $data(F)$.

To access $\mathcal{F}$ efficiently, the server maintains indexes on its columns. In that case, accessing a tuple in $\mathcal{F}$ is of logarithmic complexity. Since the server performs frequent access on $\mathcal{F}$, we also assume that it is kept in main memory.

The definition of $\mathcal{F}$ is based on the instances of tables. Since these are modified by low-level updates, the size of $\mathcal{F}$ is variable at run time. Low-level updates add or remove a tuple to/from a table, thus possibly creating a new or empty fragment. In the following, we only consider the creation of new fragments in $\mathcal{F}$. We do not consider the removal of fragments in $\mathcal{F}$, thus also keeping empty fragments in $\mathcal{F}$.

Fragments in $\mathcal{F}$ are exclusively maintained by low-level insert operations at the server. Let e be such an operation and $fragDef(e)$ the fragment definition (R, A, c) that defines its data access. We assume that the execution of e at run time checks whether a fragment $F = (id, R, A, c, ver)$ exists in $\mathcal{F}$ for some id and ver. If not, it is created with $ver = 0$ as the first version of the fragment and a new identifier id. Recall that F is the fragment identifier and automatically computed by the system. Furthermore, we assume that a newly added tuple in $\mathcal{F}$ is persistent, even if e is later on rejected by a transaction. This is useful, since the rejection of a transactions would remove already added tuples from $\mathcal{F}$, whose identifier might be already read by other statements.

4.4.4 Fragment Versions

The system keeps a version number for each fragment that is exclusively maintained by the server. The overall idea behind versioning is to verify the up-to-dateness of cache data based on versions.

Intuitively, we maintain versions as follows: Consider a stored procedure in its commit phase that has modified the fragments $F_1, \ldots, F_n$. Note that multiple updates could have been performed on each single F_i, since the code of a procedure can contain multiple updates or loops. After all modifications have been applied, the version number *ver* of each fragment F_i is increased by one and the transaction leaves its commit phase. Updating the version number within a transaction guarantees its consistent change under concurrent executions at the server.

Fragment Access History

Changes of fragment versions are tracked in the *fragment access history* (FAH). Since updates are logged by database management systems anyway, e.g., for recovery, this can be seen as an extention of the standard database log. In Section 4.5 we use the FAH to consistently update replicated data at caches.

Given a fragment F of version *ver*, the FAH contains the sequence of updates U that has been applied on F to get the next version $ver + 1$ of F. Therefore we define:

Definition 4.6 (Fragment Access History) *The fragment access history is a system table FAH of records $(id(F), ver, U)$ where $F \in \mathcal{F}$ is a fragment, ver a version number and U a sequence of low-level updates.*

Low-level updates are kept as a sequence, since normally the order of their application is important, e.g., the result of $x = 0$; $x = x + 1$; $x = x \cdot 2$ differs to the result of $x = 0$; $x = x \cdot 2$; $x = x + 1$.

Maintaining the FAH and Fragments in $\mathcal{F}$

We present an algorithm that maintains both the fragment access history and the set of fragments $\mathcal{F}$. Recall that the algorithm is executed within the commit phase of a transaction at the server, but not at the cache. Let

$$seq = (s_1, \#s_1, e_1, val_1), \ldots, (s_n, \#s_n, e_n, val_n)$$

be the resulting execution sequence of a procedure S. The algorithm firstly computes all fragments $F_1, \ldots, F_n$ that are modified by the low-level updates in *seq*, and secondly, it increases the versions for each distinct F_i. Finally, it adds a new tuple for each distinct F_i into the FAH.

Algorithm 4.2 (Maintenance of FAH and $\mathcal{F}$)

```
Syntax:       maintain()
Input:        execution sequence $seq = (s_1, \#s_1, e_1, val_1), \ldots, (s_n, \#s_n, e_n, val_n)$
Output:       none
Shared Data:  $FAH$, $\mathcal{F}$
External:     $fragDef(e)$ with e as a low-level update
Notes:        local variables: fragment $F$, $fragmentList$ as set of identifiers $id(F)$

1. $fragmentList = \emptyset$

2. For each low-level update $e_i$ in seq:
                // Note that queries in seq are ignored by the Algorithm.

   (a) $fragmentList = fragmentList \cup fragID(fragDef(e))$
                // Determine the accessed fragment definition of e and the resulting
                // identifier in $\mathcal{F}$. Note that e has been executed, hence
                // an appropriate entry has been added to $\mathcal{F}$. Note also that
                // $fragID(.)$ returns the identifier of a fragment.

3. For each $F \in$ fragmentList:
                // Note that fragmentList is a set.
                // Since a fragment can be modified by multiple
                // updates $e_i$, $|$fragmentList$| \leq n$ holds.

   (a) Let $ver$ be the current version of $F$ in $\mathcal{F}$.
   (b) Update $ver = ver + 1$ for $F$ in $\mathcal{F}$.
                // $F$ was modified and its version is increased by 1.

   (c) Let $U$ be the sequence of all low-level updates $e_i$ $(1 \leq i \leq n)$ in seq with
       $fragID(fragDef(e_i)) = F$. Let the order of low-level updates in $U$ preserves
       their original order in seq.
                // Derive all low-level updates that access $F$.
   (d) Replace each low-level update $e_i$ in $U$ of the form $insert(R, t)$ by $insert(R, tid, t)$
       with $tid = val_i$.
                // Records of the $FAH$ are also used to synchronize data at caches.
                // The above transformation guarantees that the execution of
                // low-level updates in $U$ will create the same tuple identifier
                // at a cache. Note also that $val_i$ contains the result of a
                // low-level update which, in case of an $insert(R, t)$, is the
                // generated tuple identifier.
   (e) Add the new tuple $(F, ver + 1, U)$ to the $FAH$.
                // Note that in this algorithm $F$ represents the identifier
                // of a fragment in $\mathcal{F}$.
```

Step 2 determines the accessed fragment identifier for each low-level update. We increase the version for all distinct data fragments in Step 3 and add the sequence of low-level updates U that has caused the new version to the FAH. Step 3c computes U by determining the accessed fragment of each low-level update e_i. It calls the function $fragID(.)$ which will always produce a fragment identifier, since the low-level updates e_i have been executed and an appropriate entry had been added to $\mathcal{F}$. The replacement in Step 3d is necessary, since we use the sequences U to update cache data later on. By using low-level updates of the form $insert(R, tid, t)$, the same tuple identifier is used at a cache.

As a result, the FAH contains the current and previous version numbers of all fragments and the updates that have been applied to each version. Further, a version is consistently updated, since it is done within a transaction at the server. In the following section we use the FAH to consistently update cache data.

Transactions and Fragment Versions

The version of a fragment is always increased within the commit phase of a transaction. As stated in Section 4.3, we assume that the server implements a standard concurrency control mechanism. Then, the version number of a fragment and the corresponding data can only be defined within the scope of a transaction.

Definition 4.7 (Fragment Version) *Let a transaction access a fragment F with $(F, R, A, c, ver) \in \mathcal{F}$ and r as an instance of the table R. Then the content of F of the version ver is defined by $\sigma_{A=c}(r)$.*

Hence, each transaction that accesses in its starting phase a version number *ver* of F operates on fragment data $\sigma_{A=c}(r)$. Note that a transaction in its execution phase might perform updates on $\sigma_{A=c}(r)$, such that in this phase the content of a fragment of version *ver* can be different from $\sigma_{A=c}(r)$.

4.5 Fragment Replication and Synchronization

We have shown how server data is fragmented and how for each fragment a version number is maintained. Data is replicated at the level of fragments to caches. The attribute *mode* is used to define the synchronization mode of a replicated fragment that can be either an incremental mode ($mode = inc$), where only updates are propagated to the cache, or a full mode ($mode = full$), where the entire content of a fragment is copied to the cache. For simplicity we also use the notation $mode(F)$. The full mode is necessary, since all data of a fragment must be copied to the cache if the fragment is replicated for the first time. The full mode is also used in case a cache detects an erroneous synchronization, so that fragment data have to be re-set (see below).

On an intuitive level a subset $repl_C \subseteq \mathcal{F}$ defines the fragments that are replicated to a cache C. However, since $\mathcal{F}$ is a system table, the subset cannot be defined as such.

Definition 4.8 (Replicated Fragments) *Let C be a database cache and $\mathcal{F}$ be the fragment definition table at the server. A system table $repl_C$ of tuple $(id, mode)$ logically defines the set of fragments F with $id = id(F)$ replicated to C. We also use the notation $F \in repl_C$.*

Hence, $repl_C$ is a table of two columns with a referential key to the table $\mathcal{F}$ on the first place. The definition as system table allows the system to change the set of replicated fragments at any time. However, for simplicity we also use the notations $repl_C \subseteq \mathcal{F}$ and $F \in repl_C$ in following chapters to denote that $F \in \mathcal{F}$ is replicated at cache C.

Finding the right setting of $repl_C$ is subject to the optimization problem as addressed in Part III. For the remaining part of this Chapter, we assume that for each cache C, the table $repl_C$ is given.

For all fragments $F \in repl_C$, the server is responsible for replicating data in $data(F)$ consistently to the cache C. For this, we introduce an optimistic log-transfer scheme (see [45, 89]) in the next sections that periodically reads low-level updates of the fragment access history and applies them to cache data. It is optimistic, since updates are propagated in an asynchronous manner beyond the boundaries imposed by transactions. As a result, there can be stale data at a cache.

4.5.1 Propagate Updates to Caches

There is a synchronization process running at the server that is responsible for updating fragments in $repl_C$ for all database caches C. Intuitively, it behaves as follows: First, select all new records (F, v, U) from the fragment access history. Second, propagate the sequence U to caches C with $F \in repl_C$ and set the version of data fragments at C accordingly.

Algorithm 4.3 (Synchronization Process at the Server)

```
Input:        none
Output:       none
Shared Data:  $FAH$, $repl_C$ for all $C \in \mathcal{C}$
Note:         the algorithm is a process that runs continuously at the server,
              local variables $F$ and $F'$

  1. Retrieve oldest non-synchronized entry $(F, ver, U)$ in the $FAH$. If none exists, wait.

  2. For all caches $C$ with $F \in repl_C$:

     (a) Let $(F, R, A, c, ver')$ be the corresponding tuple for $F$.
                     // Note that $ver = ver'$ does not necessarily hold, since a
                     // younger entry in the $FAH$ could already have further
                     // increased the version of $F$.
     (b) Let $F' = (F, R, A, c, ver)$ with the version number as in the $FAH$.
     (c) If $mode(F) = full$, send message $(full, F', data(F))$ to $C$.
         Set $mode(F) = inc$.
                     // Synchronize full fragment and switch to incremental mode.
```

```
    (d) If $mode(F) = inc$, send message $(inc, F', U)$ to $C$.
                        // Apply only new updates to the fragment.

3. Start again with 1.
```

The algorithm checks the fragment access history and starts data synchronization as soon as new fragment versions appear. Note that the retrieved tuples are processed in the same order as they appear in the FAH. This is necessary in order to consistently update data according to increasing version numbers.

Step 2 performs a loop over all caches that host a copy of the modified fragment. In Step 2c all tuples of a fragment are send to a cache. This is necessary, if data are replicated for the first time, or if the cache has lost an intermediate update (see next Section). In Step 2d, only low-level updates are propagated to the cache. Note that $data(F)$ and U contain the tuple identifiers. Hence, fragment data is properly replicated to caches.

Both Algorithm 4.2 for maintaining the FAH and Algorithm 4.3 for synchronizing cache data, operate only on the FAH, $repl_C$ and $\mathcal{F}$. Both act consistently, since Algorithm 4.2 only (i) inserts tuples into FAH, (ii) adds fragments to $\mathcal{F}$ and (iii) sets the version of fragments $F \in \mathcal{F}$. Algorithm 4.3 only retrieves tuples from FAH. Hence, they do not interfere in each others operations on commonly used data structures.

4.5.2 Receive Updates from the Server

A database cache continuously awaits synchronization messages from the server and applies them on local data. Similarly to the server, there is also a table $\mathcal{F}$ at the cache, that contains all replicated fragments. There is one difference. The fragment identifiers (`ID` columns of table $\mathcal{F}$) are not set automatically, but correspond to the identifiers of the server. The table is maintained exclusively by the cache by using the information of incoming synchronization messages. Note that all notions that have been defined for the server, e.g., $fragID(.)$, $fragDef(.)$, $data(.)$, etc., also apply to the cache.

Consider a fragment F and its two successive versions ver and $ver' = ver + 1$. As mentioned above, ver' might result from performing multiple low-level updates on ver. We require that Definition 4.7 of a fragment also applies for transactions at the cache. Hence, the synchronization algorithm at a cache has to block transactions at the cache until F is completely updated from ver to ver'.

The following algorithm updates data at a cache according to the two synchronization modes. Furthermore, it implements the removal of a fragment that is used by the optimizer later on to remove data from a cache that is not needed any more.

Algorithm 4.4 (Synchronization Process at the Cache)

Input:	*none*
Output:	*none*
Shared Data:	*system table $\mathcal{F}$ that contains all fragment definitions, tables $\mathcal{R}$*
Note:	*The algorithm is a process that runs continuously at the cache. Recall that the server sends messages of type $message(type, F, data)$ to a cache where data is either a sequence of low-level updates or the content of a fragment.*

```
1. Sleep until a message from the server arrives. Let mode be the type and $F = (id, R, A, c, ver)$
   be the fragment of the message.
2. Start transaction.
3. If mode = $remove$ then
   (a) Delete corresponding fragment $(R, A, c)$ from $\mathcal{F}$.
                  // Maintain local fragment definitions.
   (b) Delete all tuples $t$ in $R$ with $t(A) = c$.
                  // Update content of fragment $F$.
4. If mode = $full$ then
   (a) If exists fragment $(R, A, c)$ in $\mathcal{F}$, set its version to $ver$. Otherwise add $F$ to $\mathcal{F}$.
                  // Maintain local fragment definitions.
   (b) Let data be the third parameter of the message. Delete all tuples $t$ in $R$ with
       $t(A) = c$. Copy all tuples $t \in data$ to $R$.
                  // Update content of fragment $F$.
5. If mode = $inc$
   (a) Let $U$ be the third parameter of the message. Perform the sequence of low-level
       updates $U$ on $R$ and set $ver' = ver' + 1$ for $F$ in $\mathcal{F}$.
                  // Update content of fragment $F$.
6. Commit transaction.
7. Start with 1.
```

After receiving a message from the server, the algorithm checks its type. To remove a fragment from the cache, the server sends a message $(remove, F, .)$ that will remove all tuples that belong to the fragment F. Furthermore, the cache applies all updates within a transaction, to prevent other transactions at the cache from accessing partially updated fragments.

In case of type $full$, all tuples of a fragment are removed and the new ones are inserted. This type is used for the first synchronization of a fragment. In case of type inc, only the delivered updates are applied on the fragment. The version number is set appropriately for both types.

The algorithm exemplifies a possible synchronization scheme for an incremental update. Among others, a robust implementation of the algorithm for real-world systems requires the integration of error control to check whether none of the incremental updates got lost. In such a case, the incremental update would produce erroneous data in the cache. However, we do not address these problems here.

4.5.3 Dynamic Data Replication

The client-server system is able to replicate data dynamically. Given a cache C and the set of its replicated fragments $repl_C$, a fragment F is added to $repl_C$ by the command

> Add tuple $(F, full)$ to the system table $repl_C$. This will cause the synchronization Algorithm 4.3 to replicate F to C.

and it is removed from $repl_C$ by the commands

> Remove tuple $(F, state)$ from the system table $repl_C$ with $state = full$ or $state = inc$. This will cause the synchronization Algorithm 4.3 to stop the replication F to C. Let $F = (id, R, A, c, ver)$ be the corresponding tuple of F in $\mathcal{F}$. The replicated fragment and its data are removed from the cache by sending a message $(remove, F, .)$.

Both commands are used by the optimizer in Part III that dynamically sets replicated fragments at run time.

4.6 Executability of IO Statements

As defined in Section 4.4, the function $fragDef(.)$ returns the fragment definitions of the form (R, A, c) or $(R, A, *)$ for low-level updates and queries. The first case represents access of a fragment and the second access of a full table. However, the above algorithms propagate data at the level of fragments, such that a cache is not able to determine whether all fragments of a table have been replicated or not.

To handle this case properly, we assume that the server is sending a message to each cache C whenever all fragments of a table are replicated. These messages are sent asynchronously beyond transactional boundaries. For this, we use a boolean function $complete(R)$ at the cache that indicates whether all fragments of R are available. The cache only executes a query on a full table R, if $complete(R) = true$, otherwise the query is rejected due to missing data. Note that in Section 5 we verify all query computations anyway, such that query computations at the cache that are based on a wrong $complete(R)$ state will be detected at the server. However, since fragments are not placed and removed frequently, the value of $complete(R)$ is correct in most of the cases.

In the following we define a set-value function $access(e)$ for an IO statement e that first checks whether all data for the execution of e are available, and second returns all required fragments and their version from table $\mathcal{F}$. We use this function intensively in the remaining part of this work to determine the fragment access of an IO statement. The function is used at a cache and the server. Note that the property *complete*(R) is only required for executions at the cache, since the server hosts all data.

Definition 4.9 (Fragment Access) *Given an instantiated IO statement* e*,* $\mathcal{F}$ *and the function access*(e) *is defined as follows:*

```
1. $access = \emptyset$
2. // compute all fragment-version pairs for access of a full table
   For all $(R, A, *)$ in $fragDef(e)$:
      (a) If $access(e)$ is executed at the cache and not $complete(R)$,
          return $undef$, since table data is not completely replicated.
      (b) Else $access = access \cup \{(F, v) \mid \exists F, c, v : (F, R, A, c, v) \in \mathcal{F}\}$
3. // compute all fragment-version pairs for access of a single fragment
   For all $(R, A, c)$ in $fragDef(e)$:
      (a) check whether a tuple $(F, R, A, c, v)$ exists in $\mathcal{F}$ for some $F$ and $v$.
      (b) If it does not exists, return $undef$.
      (c) Else $access = access \cup \{(F, v)\}$
4. Return access
```

When executed at a client, the function returns an *undef* value if a full table (*complete*(R)) or a fragment is required by the execution of e, but not replicated. Executed at the server, the function returns *undef* if (1) e is a low-level insert update that (2) operates on a new fragment and that (3) has not been executed yet. Then, the execution of e has not yet added an appropriate entry into $\mathcal{F}$ (see Section 4.4.3) and in Step 3a no tuple is found. Hence, at the server the execution of *access*(e) is only feasible for already executed low-level insert updates e.

4.7 Twin Transactions

In previous Sections we have defined an incremental data replication to clients that is based on fragments and their versions. In the following, we define twin transactions and their properties that utilize data at a client to execute a stored procedure.

Definition 4.10 (Twin Transactions) *Let* S *be a low-level procedure that is triggered at a client. We call* S *a twin transaction if* S *is executed as a transaction at the client and the server simultaneously. At the end of the transaction at the client, all low-level updates are canceled and not made persistent.*

Note that forwarding the procedure call from the cache to the server takes time, such that the start of the transaction at the server is delayed by the network communication time. However, the focus is on the simultaneous execution of the procedure code. Hence, the execution of S starts running in parallel at the client and the server. The server executes the procedure code and the result of the execution at the client depends on the replicated fragments. At a client, all low-level updates are canceled, since committed updates require an additional synchronization with the server that we do not consider here. However, at the end of Chapter 5 we discuss an extention of our scheme, where consistent updates are committed at the client.

In the following, we look at the problem of global unique tuple identifiers that can be solved by using twin transactions. Furthermore, we define conditions under which the execution at the client produces the same result as the server.

4.7.1 Unique Execution and Tuple Identifiers

Twin transactions are identified by a unique global identifier *eid* that is generated by the client. For this, we assume that each client (including its cache) is identified within the client-server system by a unique client ID $cid \in \mathbf{N}$. Furthermore, each client counts its procedure calls by $n_{cid} \in \mathbf{N}$. Then, the global identifier *eid* is uniquely defined by the tuple (cid, n_{cid}).

Since a twin transaction executes the same procedure code at a client and the server, we require that an operation $tid = insert(R, t)$ creates the same tuple identifier *tid* at a cache and the server. Recall that the synchronization process does not generate new tuple identifiers at the cache, since only operations $insert(R, tid, t)$ are performed. Only the direct code execution at the cache (e.g. the execution of a twin transaction) can perform operations $insert(R, t)$. Unique tuple identifiers are important, since the SQL statements `UPDATE` and `DELETE` utilize these identifiers within their execution. Non-unique tuple identifiers cannot guarantee an equal execution of both SQL statements at a client and the server.

To obtain a globally unique tuple identifier, we use a combination of the execution identifier, the statement identifier and the statement counter. The latter two are defined by execution sequences (see Definition 4.2).

Definition 4.11 (Unique Tuple Identifier) *Let S be a low-level procedure, eid its execution identifier and $seq = (s_1, \#s_1, e_1, val_1), \dots, (s_n, \#s_n, e_n, val_n)$ a resulting execution sequence of its execution. Then, for all $tid = insert(R, t)$ low-level updates e_i in seq the tuple identifier is defined by $tid = (eid, s_i, \#s_i)$.*

From the definition we conclude that different twin transactions always generate different tuple identifiers, since their *eid* is always different. As the tuple $(s_i, \#s_i)$ is unique within an execution of a procedure and the resulting execution sequence, all operations $tid = insert(R, t)$ of a procedure generate different tuple identifiers *tid*. Note also, that the tuple identifier is deterministic as required by Assumption 4.1.

As a result, we conclude that both executions of a twin transaction generate equal tuple identifiers if they produce same sequence $s_1, \ldots, s_n$ of IO statements. Note that the sequence determines the statement counters $\#s_1, \ldots, \#s_n$ and therefore the tuple identifier.

Among others, the chosen tuple identification mechanism is important for another property of IO statements — the locality of their execution. This property is important for the main result below.

Proposition 4.1 (Locality of IO Statements) *Let S be a twin transaction and let eid its execution identifier. Let s be an IO statement in S that is executed with $val = eval_S(s, \#s, e)$ or $val = eval_C(s, \#s, e)$ at the client or the server. Let further be*

$$access(e) = \{(F_1, v_1), \ldots, (F_n, v_n)\} \neq undef$$

the accessed fragments and their versions of e. Then, the content of fragments $F \in \mathcal{F}$ with $F \neq F_i$ for $1 \leq i \leq n$ does not affect the computation of the result val of both function calls.

Proof: The statement s can be a query, a delete or an insert low-level update. In Section 4.4.2 and Definition 4.4 (function $fragDef(.)$) we have defined the set of fragment definitions that are accessed by an IO statement. These definitions represent a fragment in a logical way. In Section 4.6 and Definition 4.9 (function $access(.)$) we have defined how to determine the fragment identifiers and their versions for a given set of fragment definitions. This function checks whether the given fragment definitions are available, thus it checks whether the given IO statement can be executed on local data. Note that all data is available at the server. We use this function also for the server, to compare its fragment access with that at the cache.
Case I: s is a query

> The function $fragDef(e)$ returns a set of fragment definitions $(R, A, *)$ and (R, A, c). The first represents all fragments of a table, the second a single fragment. The table R results from a `FROM` clause of the query and the fragment (R, A, c) is only in the result set if the conjunctive `WHERE` clause of e includes a condition $A = c$. The conjunctive form guarantees that $A = c$ does not appear in a negated context. As a result, the query processor has only to consider tuples t in R with $t(A) = c$. If all these definitions are available, the function $access(e)$ returns a set $\{(F_1, v_1), \ldots, (F_n, v_n)\}$ and $undef$ otherwise.
>
> Clearly, all fragments F that are defined on tables R' that do not appear in the `FROM` clauses of e, will not affect the result val, since by Assumption 4.1 a procedure, and therewith a query, only operates on tables as they appear in the procedure code. If F is defined on a table R that appears in the `FROM` clauses of e, it can only affect val if e reads data from F. Let (R, A, c') be the fragment definition of F. From $F \neq F_i$ for $1 \leq i \leq n$ we conclude that e does not contain a predicate $A = c'$ in one of its conjunctive `WHERE` clauses and that e does not operate on the full table R, but one a fragment F_i of R with a fragment definition, say (R, A, c). Since the query processor only considers tuples t with $t(A) = c$ the

computation of the query result *val* is not affected by tuples t with $t(A) = c'$. Note that we only consider non-overlapping fragments, such that $c \neq c'$ holds.

Case II: s is a delete low-level update

A delete low-level update only removes a tuple from a fragment and has no side-effects. Since only one tuple is processed, only one fragment can be modified. As a result, the content of all other fragments does not affect the delete operation which returns the deleted tuple as the result *val*.

Case III: s is a insert low-level update

An insert operation adds one tuple to one fragment F_1 and returns the tuple identifier as result *val*. It has two side-effects: The first is to maintain the fragment definition table $\mathcal{F}$ if a new fragment is accessed. Since at most a tuple is added to $\mathcal{F}$ the content of fragments F cannot affect the insert into F_1. The second is the creation of the unique tuple identifier. The identifier is only computed from the values *eid*, s, and $\#s$. Hence, it is not affected by the content of other fragments F.

Note that this does not necessarily hold if the tuple identifier is computed on a different basis. Consider the case where a new tuple identifier is computed by taking the maximum identifier of a table increased by one. Consider a fragment $F \notin \{F_1, \ldots, F_n\}$, but that is defined on the same table R as one of the $\{F_1, \ldots, F_n\}$. Assume that S performs an insert on R that generates a new tuple identifier. Then, the generated identifier depends on the number of tuples in F, since each insert would increase the maximum identifier. Hence, the computation of the result *val* would depend on the content of the fragment F.

$\square$

4.7.2 Consistent Twin Transaction

As stated earlier, fragment versions have been introduced for verifying the up-to-dateness of cache data. First, we show that equal version numbers of fragments at the client and the server imply equal fragment content. Second, we show a consistent twin transaction where both executions produce equal execution sequences.

Proposition 4.2 *Let S be a twin transaction. Let both executions operate only on a single fragment F. If the local versions numbers of F are equal, then also the local contents of F are equal in the starting phase of both transactions.*

Proof: Let F_C and F_S denote the fragments at the cache and the server. Let $ver(F_C)$, $ver(F_S)$ denote the versions of the fragments and $data(F_C)$, $data(F_S)$ their content at the starting time of both corresponding transactions.

By Algorithm 4.2 (maintenance of the fragment access history FAH) we conclude that at the server the version $ver(F_S)$ has been set (Step 3b) and that a tuple $(F_S, ver(F_S), U)$ has been put into the FAH (Step 3c). Since this has been done within a transaction, both changes have been committed at the server.

Note that we use a standard concurrency protocol at the server, such that the committed updates are persistent and consistent under other concurrently running transactions at the server. The committed changes allow S to access the fragment F_S with its version $ver(F_S)$ in its starting phase.

Since the cache accesses a version $ver(F_C) = ver(F_S)$, the synchronization Algorithm 4.3 at the server has: (1) processed the tuple $(F_S, ver(F_S), U)$ in the FAH and (2) sent a message $(full, t, data(F_S))$ or (inc, t, U) to the cache. The synchronization Algorithm 4.4 at the cache has (1) performed Step 4 or 5, to set the local version $ver(F_C)$ and to update $data(F_C)$, and has (2) committed these updates.

Again the committed changes allow S to access the fragment F_C with its version number $ver(F_C) = ver(F_S)$. Note that the corresponding execution at the cache of a twin transaction rejects all low-level updates. Hence, the fragment F_C cannot be modified by local transaction, but only by the synchronization Algorithm 4.4.

Hence, we have only to show that $data(F_C) = data(F_S)$ holds after the commit of the transaction of Algorithm 4.4.

Let us take a look at the Steps 4 and 5, and how $data(F_C)$ is set. Recall that the algorithm propagates $insert(R, tid, t)$ and no $insert(R, t)$ operations, such that any insert at the cache creates the same tuple identifier as the server. We consider both Steps independently:

- **Step 4 (full update)**
  The server has sent the full fragment content and the local content of F_C is set to $data(F_C) = data(F_S)$.

- **Step 5 (incremental update)**
  The server has sent a sequence of updates U that is applied on F_C at the cache. Prior to an incremental update the server had sent the full content of F_C. Then, all data at the server has been propagated to the cache. Under the assumption that none of the incremental updates gets lost, the content of F_C at the cache must be equal to F_S. Hence, $data(F_C) = data(F_S)$ holds.

For both Steps $data(F_C) = data(F_S)$ holds which completes the proof. □

Finally, we show that equal fragment versions at cache and server result into a consistent twin transaction, where both executions produce the same execution sequence.

Proposition 4.3 (Consistent Twin Transaction) *Let S be a twin transaction. Let both executions of S operate on fragments of equal versions. That is,*

$$\bigcup_{i=1}^{m} access(e_i) \quad = \quad \bigcup_{i=1}^{k} access(e'_i) \quad = \quad \{(F_1, v_1), \ldots, (F_n, v_n)\} \quad \neq \quad undef$$

holds for all executed (but not committed) IO statements $e_1, \ldots, e_m$ *at the cache and* $e'_1, \ldots, e'_k$ *at the server. Then, both executions of* S *produce the same execution sequence.*

We refer to non-committed IO statements, since in the commit phase of a transaction the version numbers of the modified fragments are increased. Furthermore, we refer to executed IO statements, since only then the function $access(e)$ produces meaningful results (see also Section 4.6). For the same argumentation the set of fragment-version pairs is not $undef$, since this only can be the case for non-executed IO statements at the server.

Proof: Let $R_1, \ldots, R_l$ be the tables as they appear in the procedure code of S. If the set $\{(F_1, v_1), \ldots, (F_n, v_n)\}$ contains all fragments of these tables, we conclude by Proposition 4.2 that the content of the fragments at cache and server is equal and therefore also the instances of the tables $R_1, \ldots, R_l$ are equal. By Assumption 4.1 we conclude that both executions produce the same instance and the same execution sequence which includes the generated tuple identifiers and query results.

If the set $\{(F_1, v_1), \ldots, (F_n, v_n)\}$ does not contain all fragments of the tables $R_1, \ldots, R_l$, there is a fragment F of these tables with $F \notin \{F_1, \ldots, F_n\}$. According to Proposition 4.1, however, the execution of the IO statements $e_1, \ldots, e_m$ and $e'_1, \ldots, e'_k$ is not influenced by the content of such fragments F. If the content of such fragments is irrelevant for the execution of S, we can assume any content for them for showing the Claim. Therefore, assume that such fragments F have the same version number and content at the cache and the server. Then, all fragments of the tables $R_1, \ldots, R_l$ are replicated to the cache and therewith the instances of these tables are equal. Again Assumption 4.1 applies and both executions of S result into the same instances of $R_1, \ldots, R_l$ and produce the same execution sequences. □

Claim 4.3 is used in Chapter 5 to show the correctness of the simultaneous execution scheme.

4.8 Related Work

The concept of stored procedures is very popular and almost every database vendor provides stored procedures for different procedural languages. In the literature, we can find similar considerations. *Knowledge Independence* [20] has been proposed as a new design principle, which implies extracting knowledge from applications and putting it into the schema, in the form of trigger and rules (or stored procedures). In this way, knowledge is centralized and becomes declarative, simplifying application design and evolution. The goal is to factor out as much of the semantic knowledge that is encoded in the applications as possible, and place it under the control of the database system itself, thereby enforcing it across all applications.

Within the pre-compilation of stored procedures, we have demonstrated how a uniqueness constraint can be handled. Modern database systems provide a variety of integrity constraints (e.g. primary key, foreign key, uniqueness, NULL values, default values, etc.) and advanced active concepts, such as triggers and rules. However, adding extra code by pre-compiling it into the procedure code to enforce the integrity constraints is not trivial and leads to well-known problems, such as termination [92, 93, 105], confluence [99], computability [68], etc. These problems have been studied by many authors. An overview is given by [73].

As mentioned in Section 2.2, most of the recent database caching schemes operate on a tuple basis and apply a grouping mechanism. Grouping is used to reduce the maintenance effort for replicating data to caches, and to capture semantic relationships among tuples. In this work, we use fragmentation that does not consider semantic information. This is feasible, since the performance improvement of our simultaneous execution scheme can be shown independently of the underlying grouping mechanism.

In Section 2.4, we have summarized the problem of synchronizing replicated data. As mentioned, we use an optimistic log-transfer protocol that manipulates replicated data with controlled inconsistencies that must be resolved by the system. As shown by Claim 4.3, we detect such inconsistencies by comparing the version number of fragments. Then, equal versions represent the consistent case.

The concept of twin transactions has also been suggested for mobile environments ([26, 86]). However, they use it in the context of performing two similar transactions in a sequential, rather than in a simultaneous manner. Their objective is to first perform a transaction on a mobile device and then repeat its execution at the server later on during the next connection phase of the device.

4.9 Summary and Discussion

We have presented a client-server database system that uses a database cache at each client. The system provides twin transactions and dynamic data replication to caches.

A twin transaction characterizes two equal procedure calls where one is executed at the cache and the other simultaneously at the server. We have shown that both executions lead to the same effect on the underlying tables if both have operated on equal fragment version. To achieve this, we had to adhere to deterministic procedure code and had to introduce globally unique tuple identifiers that guarantee an equal creation of tuple identifiers during the simultaneous execution. The concept of twin transactions is the basis for our simultaneous execution scheme as defined in Chapter 5.

The chosen replication scheme allows us to dynamically replicate data at the level of table fragments. This way, data can be placed at caches specifically to the access behavior of users, and it can be adapted at run time whenever the access behavior changes. These issues are discussed in Part III.

Compile Constraints and Trigger Code into Stored Procedures

Schema constraints (like foreign key on tables) and advanced concepts for integrity enforcement (e.g. trigger) are not taken into account by the client-server system. As mentioned above, the chosen language exemplifies our approach and the restrictions have been made to keep the system simple. However, intuitively we can argue that an integration of these concepts is feasible. Let us consider the trigger mechanism and let us assume that schema constraints are internally mapped to triggers. A trigger is a piece of procedure code that is executed for a certain event, e.g., insert a tuple into a table. A complier at the database level can insert the entire trigger code into the original procedure code, and can further apply translation rules, so that the code only performs low-level updates (see Section 4.2.3). Of course, the compiler has to cope with termination and confluence problems ([92, 93, 105, 99]). Hence, to some extent the advanced concepts can be handled at the compiler level. First steps into this directions have been discussed in [59].

Sub-Procedure Calls and Internal Functions

Our language also imposes a restriction on sub-procedure calls and internal functions. As long these occur in expressions of low-level updates, assignments and return operations, they can be handled by the above mentioned compiler. If they occur within queries, e.g. in PostgreSQL, a stored procedure can be used as a table in the `FROM` clause of a query. This is more difficult, since they affect the query processor which includes query planing and optimization.

Still, a subject of future work is the problem of integrating query execution plans partially into the low-level procedure code. Our scheme could handle internal functions and sub-procedure calls in queries. As a result, the low-level procedure contains the computation of sub-queries, joins, etc. explicitly, which enables our scheme to handle partial query evaluation at caches. However, this is only possible if intermediate query results, e.g., result of a sub-query, are small enough for transferring them from the cache to the server. It is out of question that the intermediate result of a cross join of big tables should not be transfered via a network.

Fragment Access Detection

To exemplify our approach, we have introduced a primitive algorithm for detecting the fragment access of an IO statement. It operates on a syntactical basis and detects only full table access and the access of a single fragment for each table in the `FROM` clause of a query. Hence, more work has to be done in providing techniques for an efficient and precise fragment access detection, as shown by Example 4.1.

Chapter 5

A Simultaneous Execution Scheme for Split Procedure Code

In Chapter 4 we introduced a client-server database system with data replication to caches at clients and the concept of twin transaction for a simultaneous executing of a transaction at a cache and the central server. Based on this, we define the concept of *split twin transactions* in this Chapter. The main idea is to logically split the procedure code into two parts and to assign one to the cache and the other to the server. When a twin transaction is executed, the cache and server only execute their assigned parts of the procedure code. This way the execution of a procedure call is distributed among cache and server in a parallel fashion. As shown at Figure 5.1, we extend the database cache by a *partial execution* and the server by a *query result cache*, a *Result Verification* and a component for *Static Code Analysis*. As

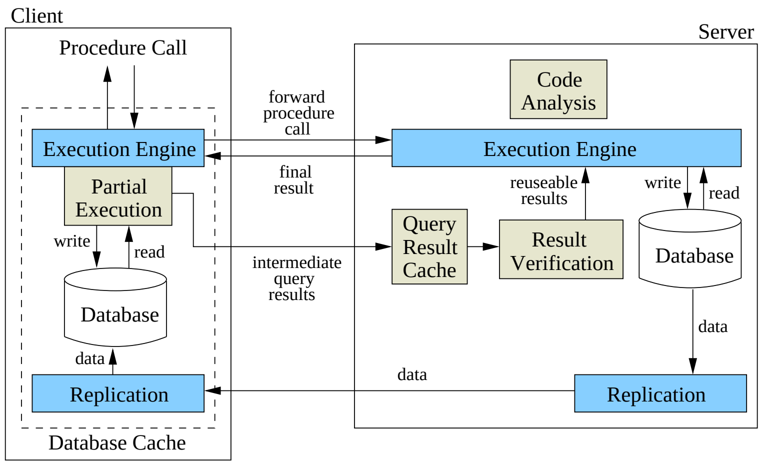


Figure 5.1: Architecture of a Database Cache.

we will show in Chapter 6, our scheme further improves the performance w.r.t. traditional execution for database caching.

5.1 Split Twin Transactions — An Overview

We demonstrate the concept of split twin transactions by an example. Consider again the `AddCart` procedure in Example 5.1. The procedure adds a given number of product items to the user's shopping cart if available, otherwise an error message is returned to the user.

Example 5.1 (The Stored Procedure `AddCart`) *The input parameters are the users session identifier* `sessID`*, the product identifier* `prodID` *and the* `amount` *of requested items. The procedure uses the local variables* `session` *and* `stock`*. Symbols depicted at the right side represent the SQL queries and updates of the corresponding line of code (query = rectangle, update = circle). They are used in figures and text that illustrate the execution of the procedure.*

```
PROCEDURE AddCart(sessID,prodID,amount)
DECLARE session, stock AS integer;
BEGIN
  1  UPDATE Statistic SET requests=requests+amount WHERE pid=prodID;      $\textcircled{U}$
  2  session := (SELECT * FROM Session WHERE id=sessID);                   $\boxed{S}$
  3  IF session is null THEN
  4      ...
  5      RETURN Error;
  6  inStock := (SELECT storage >= amount FROM Stock WHERE pid=prodID);    $\boxed{A}$
  7  IF NOT inStock THEN
  8      ...
  9      RETURN Error;
 10  INSERT INTO Cart VALUES(sessID,prodID,amount);                        $\textcircled{I}$
 11  RETURN SuccessCode
END
```

In the following we only consider the main execution sequence of the procedure which is $\boxed{U}$ $\textcircled{S}$ $\textcircled{A}$ $\boxed{I}$. We neglect the other possibilities, since they only rarely occur at run time. Figure 5.2 depicts the resulting split twin transaction. In twin transactions, cache and server still execute the same low-level procedure code, but the cache can 'ignore' certain statements and the server can use query results from the server instead of performing them by itself. In the example the database cache does not execute the code of line 2 to 5 and sends the result of query $\boxed{A}$ to the server. Note that the code in 1 to 5 does not affect query $\boxed{A}$. The server executes the whole procedure code, but uses the query result of the cache in line 6 to assign the local variable `inStock`. Finally, the query $\boxed{A}$ is executed simultaneously by the cache while the server is executing the lines 1-5.

Motivated by the above example, we define split twin transaction.

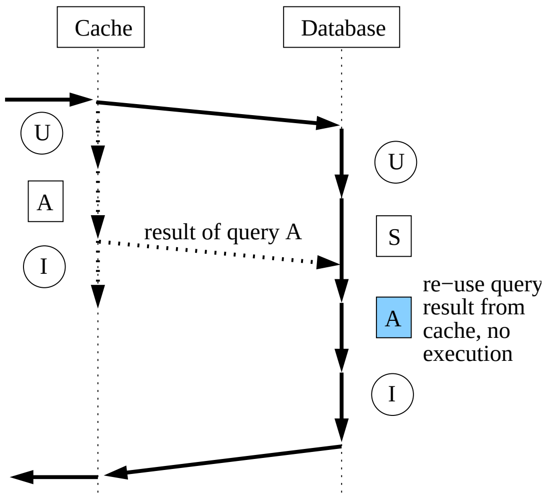


Figure 5.2: Split Twin Transactions with passing a query result to the server.

Split Twin Transactions

Recall that the procedure code contains IO statements $s_1, s_2, \ldots$ that are uniquely identified by number $s_i \in ID(S)$ with $i = 1, 2, \ldots$ (see Section 4.3). Based on twin transactions (see Definition 4.10) a *split twin transaction* is defined in the following by a subset $split \subseteq ID(S)$ of the IO statements (low-level updates and queries) of S that precisely defines which IO statements are allowed to be executed at cache and server.

Definition 5.1 (Split Twin Transactions) *Let S be a twin transaction and $split \subseteq ID(S)$ a subset of IO statements. We call S a split twin transaction if the cache executes at most all statements in split, and sends the result of each query $q \in split$ to the server. When the server needs to process q while executing S, the delivered result of q is reused, if it is correct, otherwise q is executed by the server.*

The procedure code is logically split into two parts. At the cache, only the IO statements in *split* are allowed to be executed, and all other statements are not. At run time, not all IO statements in *split* are necessarily executed, since the cache might not reach the entire procedure code. Consider the above example, where the line of code represents the statement identifier. Assume line $6 \in$ *split*. If the `RETURN` in line 5 is executed, the query in line 6 is not executed. Accordingly, the server tries to use a delivered result. Therefore it does not necessarily execute all queries in a split.

In the remaining part of the Chapter, we consider only split twin transaction for one cache and a central server and assume that for each procedure a split parameter is given. We also write $split(S)$, to denote the correspondence to a procedure S. In Part III we define a run time optimizer that identifies *split* parameters with a high performance, e.g., execution time.

The Simultaneous Execution Protocol

The definition of split twin transactions let arise further questions that will be elaborated while sketching the underlying execution protocol for split twin transactions. Note that the synchronization process at the server continuously updates replicated data at the cache. For a given $split \subseteq ID(S)$, a procedure call of S is executed as follows:

1. A client initiates a procedure call to the local database cache, which immediately passes the call to the central server. The call is uniquely identified by the execution identifier *eid*.
2. Directly after passing the call, without awaiting a response from the server, the cache starts to execute the low-level procedure code of S. Analogously, the server starts to execute the same procedure code.
3. The execution at the cache:
    (a) The cache processes the code until the next IO statement s. In case $s \in split$ the statement is executed, otherwise it is not executed on data and ignored. We propose a partial execution scheme that defines how to further process the code once statements have been ignored.
    (b) The result of each query is sent to the server. To handle these results for multiple concurrent executions of the client-server system, we use a *Query Result Cache* (QRC) at the server for keeping results until they are needed by the code execution there. Note that the server might wait for a query result, or the result might be delivered, before the server actually processes the appropriate query.
    (c) The cache handles all following IO statements analogously and aborts if it receives a completion message from the server. Thereby the server has already committed the transaction and no more query results are needed from the cache.
    (d) Once the execution at the cache has been aborted or completed, all computed updates are undone and the cache awaits the next procedure call. We have chosen this strategy, since a cache might have performed stale updates, and since the synchronization process at the server propagates updates from the server to the cache anyway.
4. The execution at the server:
    (a) The server executes the procedure code and checks for every query in *split*, if there is an appropriate result in the query result cache. If so, then the result is verified and re-used within the execution. In that case, the query is not executed on server data.

        Not all delivered results can be used by the server, since an execution on stale data at a cache might produce erroneous results. For this, we propose a verification scheme that takes into account the version number of fragments and intermediate updates that have been performed by the cache and the server.

(b) The server completes the execution and commits all updates on server data. Finally, it passes the result of the computation to the cache which undoes all updates on cache data.

5. The result of the execution is sent to the client application that initiated the request.

As we want a simple integration of the new scheme into the execution engine of a database system, we propose the following interfaces:

- The function $eval(.)$ of the client-server system is replaced by the functions $eval_C(.)$ (cache) and $eval_S(.)$ (server) to implement the specific behavior of the cache and the server.

- Once a cache ignores the execution of an IO statement, the remaining code could possibly no longer be processed properly, e.g., a local variable is not assigned. Therefore, we define a set of rules that extend the execution engine to cope with partial execution.

- The commit phase at the server is equal to those of twin transactions and the commit at a cache is extended by the undo of local updates.

In Section 5.2 we define the query result cache. The behavior of caches and the server is defined in the Sections 5.3 and 5.4. In Section 5.5 we analyse the procedure code, identify independent queries and discuss useful split parameters. We show the correctness of our scheme in Section 5.6 and finally point to related work and discuss premisses, drawbacks and possible extentions of our approach.

5.2 The Query Result Cache

Consider, for example, Figure 5.2. There, the result of query $\boxed{\text{A}}$ is delivered while the server is executing query $\boxed{\text{S}}$. For this, every query result is put into the servers query result cache (QRC) and the server is accessing the cache whenever needed, e.g., after completing query $\boxed{\text{S}}$ and proceeding with the verification of query $\boxed{\text{A}}$. Of course, the delivery time matters and the server might have to wait for a result.

Once a result is put into the QRC, the server must be able to precisely recognize its origin. For this, the QRC has to include at least the unique execution identifier *eid* and the identifier of an executed query statement $s \in split$. However, this information is not sufficient, since a query might be executed multiple times within a loop, and since a query might be differently instantiated by local variables. As defined by execution sequences, we also add the instantiated IO statement e and the statement counter $\#s$ into the QRC. The following example demonstrates the necessity of the counter.

Example 5.2 *Consider the following piece of procedure code. Again lines of code represent the statement identifiers.*

```
1 FOR row IN SELECT * FROM R1 LOOP
2    IF row.A1>row.A2 THEN
3       INSERT INTO R2 VALUES (row.A1);
4       var := SELECT AVG(A1) FROM R2;
5    END LOOP
6 END FOR
```

The loop adds new tuples into table `R2` *and computes the average of each run over the column* `A1`*. At run time the execution results in a sequence* $1, 3, 4, .., 3, 4, .., 3, 4$*. Then the corresponding execution sequence contains (among others) multiple tuples*

$(4,\ i,$ `SELECT AVG(A1) FROM R2`$,\ val)$

where 4 *is the statement identifier,* i *the counter starting at 1 and* val *the average value assigned to the local variable* `var`*. Hence, the same query is executed multiple times and may result in a different result* val *each time. The only chance to distinguish between single executions is the statement counter* i*.* □

Since the server has to verify a delivered query result (see Section 5.4), we additionally put information about fragment versions and applied updates into the cache.

Definition 5.2 (Query Result Cache (QRC)) *Let* eid *and* seq *result from an execution of a low-level procedure* S *at the cache. The query result cache consists of tuples*

$$(eid, s, \#s, e, val, F, seq, state)$$

with $s \in ID(S)$ *as a query,* $(s, \#s, e, val) \in seq$*,* F *as the result of* $access(e)$ *at the cache and* seq *as an execution sequence of low-level updates. The* $state$ *contains additional information about the execution of* e *at the cache.*

The values eid, s and $\#s$ uniquely identify a tuple in the QRC. The read and write access on the QRC and the attribute $state$ are elaborated in subsequent sections. We restrict seq to low-level updates only, since the sequence is used to capture executed updates at the cache. As we show in Section 5.4, F and seq are sufficient to verify a query result at the server.

5.3 Execution at the Cache

In this section we define the behavior of a cache during the execution of split twin transactions. This covers the partial execution model, the final commit of transactions and the $eval_C(.)$ function that replaces the original function $eval(.)$ of the system.

5.3.1 The Partial Execution Model

Given a procedure S and $split \in ID(S)$, the basic idea of our scheme is that the function $eval_C(s, \#s, e)$ returns an $undef$ value and does not execute the IO statement e on data for all $s \notin split$. We use the same mechanism for handling missing data at a cache. That is, the function $eval_C(s, \#s, e)$ returns $undef$, if the statement e could not be executed due to missing fragments.

Clearly, the result delivered by $eval(.)$ might be further used in the procedure code which possibly prevents other IO statements from being executed. Our procedure language is affected as follows:

- If the query at the right side of the assignment statement `<VAR> := (SELECT .. FROM .. WHERE ..)` returns an $undef$, the local variable `<var>` cannot be assigned.
- If the query of a loop statement `FOR <var> IN LOOP <query> LOOP <stmts> END LOOP` is not executed, the entire loop cannot be executed. This affects loops in the original code as well as the loops that are used by the pre-compilation of `INSERT`, `DELETE` and `UPDATE` statements.

Hence, a local variable might be of value $undef$ or a loop statement is executed for an $undef$ value. Both cases might cause further conflicts in the code execution. For example, an $undef$ value is used within a query or low-level update `SELECT * FROM R WHERE A=`$undef$ or $insert(R, (1, undef))$ which is not defined on data. These and other conflicts will be handled by our partial execution scheme.

In the following we define the partial execution by a set of rules that apply for the execution of certain statements of the procedural language. We assume that the execution engine of a cache implements these rules properly. Given a low-level procedure S and a parameter $split \subseteq ID(S)$, the rules are defined as follows:

Rule 1: The function $eval_C(s, \#s, e)$ returns $undef$ if the IO statement e cannot be executed, due to missing data in the cache.

Rule 2: For all IO statements $s \notin split$ the function $eval_C(s, \#s, e)$ returns $undef$ and e is not executed on data.

Rule 3: If the instantiated expression of an IO statement contains $undef$, it can not be executed on data. For this, the function $eval(s, \#s, e)$ returns $undef$ and does not perform e on data.

Rule 4: An assigned variable `<var>` is of value $undef$, if the assigned expression contains an $undef$ value. This affects `<var> := (SELECT .. FROM .. WHERE ..)` and `<var>:=<expr>`.

Rule 5: Once the condition of an `IF <cond> THEN .. ELSE .. END IF` contains an $undef$ value, it can not be evaluated at run time and the program path of the `IF` statement cannot be determined. All variables that are assigned in the branches of the `IF` statement are set to $undef$ and the `IF` statement, including its branches, is not executed. The execution continues with the next statement after `END IF`.

Rule 6: If the query expression of a loop statement yields *undef*, the body of the loop is not executed. All variables that are assigned in the body are set to *undef* and the engine continues with the next statement after `END LOOP`.

Rule 7: All statements `RAISE EXCEPTION` and `RETURN` in the original code are ignored.

Rule 8: Whenever the execution engine jumps over an `IF` or `LOOP` statement that contains a query $s \in split$, a notification with the state *undef* is put into the query result cache (QRC) for s.

The rules cover all expressions `<expr>` and conditions `<cond>` of the procedure language, such that all $split \in 2^{ID(S)}$ can be handled by the partial execution scheme. Furthermore, they are very liberal and cause the engine to execute as much of the procedure code as possible. Rules 1 to 3 have to be implemented by the $eval(.)$ function and and rules 4 to 7 by the underlying execution engine. Clearly, a partial execution might perform inconsistent updates on data while operating on inconsistent data. Such cases are handled at committing time where all local updates are rejected.

Note that the rules operate on a syntactical basis and do not consider semantic dependencies in the procedure code. We demonstrate the issue with the following example.

Example 5.3 (Application of Rules and Code Dependencies) *Consider the following snippet of procedure code. Each table has one column, denoted by* `A`*.*

```
1 INSERT INTO R2 VALUES (x);
2 var1 := SELECT count(*) FROM R1 WHERE TRUE;
3 var2 := var1 + 1;
4 var3 := SELECT count(*) FROM R2 WHERE A<var2;
```

Let the line of code represent the identifier of IO statements. Consider an execution of the procedure with split $= \{1, 4\}$*. Then, the first* `SELECT` *is not executed and by* **Rule 2**, *undef is assigned to the variable* `var1`*. By* **Rule 4**, *undef is also assigned to* `var2`*. By* **Rule 3**, *the execution engine does not perform the second* `SELECT`*, since its instantiated expression contains an undef value and the query is not defined.*

If we consider an execution with split $= \{2, 4\}$*, both* `SELECT` *statements are executed by the engine and the variable* `var2` *is assigned. However, in this case the result of the second* `SELECT` *on table* `R2` *would be possibly erroneous, since the* `INSERT` *has not been executed on table* `R2`*.* □

The example shows the necessity of verifying query results at the server which is discussed in detail in Section 5.4. Semantic dependencies within the procedure code are discussed in Section 5.5.

5.3.2 Handling IO Statements

The $eval_C(.)$ function at the cache is responsible for:

- Handling $undef$ values that result from the partial execution rules.
- Sending intermediate notifications that inform the server of currently executed IO statements. Such notifications are also put into the QRC.
- Putting query results into the QRC.

Assume the cache has partially executed the procedure code, thereby producing a sequence

$$seq = (s_1, \#s_1, e_1, val_1), \ldots, (s_{n-1}, \#s_{n-1}, e_{n-1}, val_{n-1})$$

where each element represents a call of $val_i = eval_C(s_i, \#s_i, e_i)$. Let $eval_C(s_n, \#s_n, e_n)$ be the next call that is performed by the execution engine. Then, the function $eval_C(.)$ is defined as follows. The definition uses the function $access(e)$ (see Definition 4.9).

Definition 5.3 (The $eval_C(.)$ Function at the Cache) *Let the notions be as above. Then, $eval_C(s_n, \#s_n, e_n)$ is defined by:*

1. *If $s_n \notin split$, return $undef$ (Rule 2).*
2. *If s_n is a low-level update*
   (a) *If e_n contains $undef$, return $undef$ (Rule 3).*
   (b) *If $access(e_n) = undef$, return $undef$ (Rule 1).*
   (c) *Execute $val = eval(s_n, \#s_n, e_n)$.*
3. *If s_n is a query*
   (a) *If e_n contains $undef$, put $(eid, s_n, \#s_n, e_n, \emptyset, \emptyset, \emptyset, undef)$ into the QRC and return $undef$ (Rule 3).*
   (b) *If $access(e_n) = undef$, put $(eid, s_n, \#s_n, e_n, \emptyset, \emptyset, \emptyset, undef)$ into the QRC and return $undef$ (Rule 1).*
   (c) *Let $seq' = seq$. Remove elements $(s, \#, e, val)$ from seq' with (1) e as a query or (2) e as a low-level update with $access(e) \cap access(e_n) = \emptyset$.*
   // Note: As a result, only the low-level updates in seq'' operate on the same
   // fragments as the query e_n. Hence, they affect the computation of e_n.
   (d) *Put $t = (eid, s_n, \#s_n, e_n, undef, access(e_n), seq', note)$ into the QRC*
   // Note: Informs the server that the cache is currently executing the query.
   (e) *Execute $val = eval(s_n, \#s_n, e_n)$.*
   (f) *Update t in QRC. Replace $undef$ by val and $note$ by $exec$.*
4. *return val*

Step 1 implements **Rule 2** of the partial execution model which requires that statements in $ID(S) - split$ are not executed at the cache. In Step 2 a low-level update is only executed if its expression does not contain $undef$ values (**Rule 3**) and all data are available (**Rule 1**). Recall that the Algorithm calls the original $eval(.)$ function of the underlying system in Step 2c.

Step 3 handles queries as follows. The Steps 3a and 3b correspond to the Steps 2a and 2b, but additionally send a notification to the server that the cache could not execute the query. For this, it puts a tuple into the QRC that contains the state $undef$. The execution sequence seq' (Step 3c) contains all low-level updates that affect the same fragments as query e_n. These values are essential for the servers result verification (see Section 5.4.3). In Step 3d the server is informed that the cache has started the execution of the query. Fragments, its versions and the sequence of executed low-level updates are also put into the QRC. If the query has been computed, the result is put in to the QRC (Step 3f).

Note again that a cache at run time does not necessarily execute all $s \in split$, e.g., `IF <cond> THEN` s_1 `ELSE` s_2 `END IF` with $s_1, s_2 \in split$ executes either s_1 or s_2.

5.3.3 Completing an Execution

Within a split twin transaction, identified by *eid*, the cache and the server simultaneously execute a procedure call. After the server has finished its execution, it sends the result r to the cache which might either be a value (single value, a tuple or a table) or an error message, in case the transaction at the server has been rejected.

While receiving this message, the cache might be either still executing the procedure code or has already finished its execution. Independently from the execution at the cache, the incoming commit of the server causes the execution at the cache to stop. Furthermore, none of the low-level updates at the cache are committed. As a result, the execution at the cache has not modified cache data.

In case the execution at the cache terminates and no commit has been received from the server, the cache sends a completion message to the server. For this, the QRC is used again and a message $(eid, \ldots, end)$ is put into the QRC. The message contains only the execution identifier *eid* and a status *end* which indicates that no more query results are delivered by the cache.

5.4 Execution at the Server

This Section defines the behavior of the server which is again integrated into the $eval(.)$ function of the system that is replaced by $eval_S(.)$ at the server.

5.4.1 Retrieving Query Results from the QRC

Each IO statement at the server is executed by the $eval_S(s, \#s, e)$ function of the underlying execution engine. For a query s that has been computed by a cache ($s \in split$), the server checks the QRC for an appropriate result, thereby possibly waiting for a result to appear. If a result was delivered, it is verified and, if valid, reused by the server and not executed. If the result is invalid the server has to execute the query.

Before we define the function $eval_S(s, \#s, e)$, we summarize its behavior on an intuitive level. We distinguish between *normal mode* and *shared mode*. The first represents the case where the server executes the procedure code without using query results from the cache, and the second represents the continuous reuse of query results from the cache. A split twin transaction is always started in shared mode. Figure 5.3 depicts the process of retrieving,

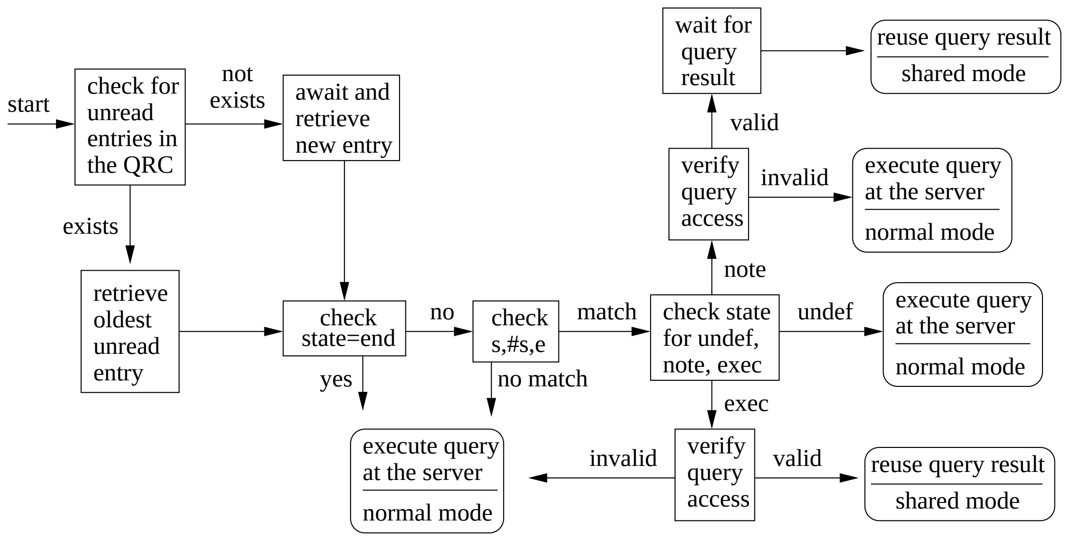


Figure 5.3: Retrieve, verify and reuse query result from the query result cache handled by the function $eval_S(.)$.

verifying and reusing a query result from the query result cache. Furthermore, it shows all cases where the system switches to the normal mode. Intuitively, a query e is handled by $eval_S(.)$ as follows:

1. Check whether there is an entry in the QRC that has not been retrieved yet.

   (a) If none exists, wait until the cache adds a new entry and retrieve it.

   (b) If an entry exists, retrieve the oldest unread entry. The oldest is important, since a cache might already have performed multiple queries and thus might already have put multiple entries into the QRC. Furthermore, the cache executes the same low-level procedure code, hence it executes IO statements in the same order as the server. Even if the cache does not execute some of the statements, the original order is preserved. Therefore, the server always checks for the oldest entry.

2. The retrieved entry is in the form of $(eid, s', \#s', e', val, access, seq, state)$. If $state = end$, the cache has already finished its computation and no more query results are delivered, allthough the server expects them. This can be the case if a cache followed a

different execution path in the procedure code than the server. In this case the server executes e by itself and continues in normal mode.

3. In order to reuse a result, the cache must have performed the same query in the procedure code. This is the case if $s = s'$ (the statement identifier of the procedure code is equal), $\#s = \#s'$ (the cache has performed the same number of repetitions of s) and $e = e'$ (the cache has performed syntactically the same query expression). If these values do not match, the cache has followed another path in the code. Hence, it is not useful to consider more results from the cache. The server executes e by itself and continues in normal mode.

4. Once the cache has processed the same query, the final step is to figure out its execution state and to determine the reuse case.

   (a) If $state = undef$, the cache could not execute the query due to missing data or an $undef$ value in the query expression. The server executes e by itself and continues in normal mode.

   (b) If $state = note$, the cache has started but not completed with the execution of the query e ($val = undef$). However, as we show below, the cache has already submitted all information that is required to verify the incoming query result. Hence, the server can, in parallel to the cache, verify the incoming result.

      i. If the incoming result is correct, the server awaits the result to appear in the QRC, reuses it, completes the execution of $eval_S(s, \#s, e)$ and continues in shared mode.

      ii. Otherwise the server executes e by itself and continues in normal mode.

   (c) If $state = exec$, the cache has finished the execution of the query and has put the result val into the QRC.

      i. If the result is correct, the server reuses it, completes the execution of $eval_S(s, \#s, e)$ and continues in shared mode.

      ii. Otherwise the server executes e by itself and continues in normal mode.

There are two wait phases (Step 1a and 4bi) that might cause a starvation. If a cache performs properly, neither of both will occur. In Step 1a, there will be at least one entry with $state = end$ that is sent if the cache has finished. In Step 4bi the cache is already executing the query. Hence, only a crash of a cache can cause starvation. In the following, we assume a proper cache and do not further discuss starvation.

The above scheme only continues in shared mode if the delivered result was correct. This results from the observation that an invalid result can cause more invalid results, due to stale data, missing data or code dependencies in the procedure code. In Section 5.8 we discuss alternatives to continue in shared mode even for invalid queries or syntactically different queries.

5.4.2 Handling IO Statements

Based on the above process, we define the function $eval_S(.)$. For this, we assume that the server has already executed the procedure code thereby producing an execution sequence

$$seq = (s_1, \#s_1, e_1, val_1), \ldots, (s_{n-1}, \#s_{n-1}, e_{n-1}, val_{n-1})$$

where each element represents a call of $val_i = eval_S(s_i, \#s_i, e_i)$. Let $eval_S(s_n, \#s_n, e_n)$ be the next call that is performed by the execution engine. Then, the function $eval_S(.)$ is defined as follows. Again, the definition uses the function $access(e)$ (see Definition 4.9).

Definition 5.4 (The $eval_S(.)$ Function at the Server) *Let the notions be as above. Then, $eval_S(s_n, \#s_n, e_n)$ is defined by:*

1. *If $s_n \in split$ and e_n is a query and shared mode*
   - (a) *Check for unread tuples with eid in the QRC. If none exists, wait.*
     Retrieve oldest unread entry $(eid, s_n, \#s_n, e, val, access, seq', state)$.
   - (b) *If $state = end$, set normal mode and jump to 2.*
   - (c) *If not ($s_n = s$ and $\#s_n = \#s$ and $e_n = e$), set normal mode and jump to 2.*
   - (d) *If $state = undef$, set normal mode and jump to 2.*
   - (e) *Let $seq'' = seq$. Remove elements $(s, \#, e, val)$ from seq'' with (1) e as a query or (2) e as a low-level update with $access(e) \cap access(e_n) = \emptyset$.*
     // Note: As a result, only the low-level updates in seq'' operate on the same
     // fragments as the query e_n. Hence, they affect the computation of e_n.
   - (f) *If $state = note$,*
     - i. *If $access = access(e_n)$ and $seq' = seq''$,*
       wait for result val to appear in the QRC. Jump to 3.
     - ii. *Else set normal mode and jump to 2.*
   - (g) *If $state = exec$,*
     - i. *If $access = access(e_n)$ and $seq' = seq''$, jump to 3.*
     - ii. *Else set normal mode and jump to 2.*
2. *Execute $val = eval(s_n, \#s_n, e_n)$*
3. *return val*

Step 1 implements processes at sketch at Figure 5.3. As in the function $eval_C(.)$, the execution sequence seq'' (Step 1e) contains all low-level updates that affect the same fragments as the query e_n. The verification is done in the Steps 1.e.i and 1.f.i which is explained in the next Section.

We assume that a garbage collector is running in the background, cleaning the QRC. That is, all entries are removed for a completed execution *eid*.

5.4.3 Verifying Query Results

Given an entry $(eid, s, \#s, e, val, access, seq', state)$ and the function call $eval_S(s_n, \#s_n, e_n)$, we explain the correctness of the delivered result val. The full proof is given in Section 5.6. As explained in Section 5.4.1, we apply the verification only to syntactical equal queries, that is $e_n = e$ holds.

The data access at the cache, while executing the query e, is determined by the value $access = \{(F_1, v_1), \ldots, (F_i, v_i)\}$ $(i > 0)$ which contains the fragments F_j and their version v_j. Furthermore, the execution at the cache has produced a sequence seq' of low-level updates that affect the fragments in $access$. Note that by Definition 4.9, all fragments F_j are replicated at the cache.

The data access at the server is determined *before* the query e_n is processed. Analogously to the cache, it is computed by the function $access(e_n)$ which analyses the query expression e_n on a syntactical basis. Furthermore, the server might have performed a sequence seq'' of low-level updates during eid which affects the fragments in $access(e_n)$.

Equal Fragment and Version Access

If $access = access(e_n)$, then the server would access the same fragments and versions in the starting phase of the transaction to execute e_n on server data. Since equal versions imply equal data (see Proposition 4.2), the server would execute e_n on the same data as the cache. Since e_n is deterministic (see Assumption 4.1), the execution of e_n would yield the same result at the server.

Equal Sequence of Performed Low-Level Updates

However, both executions might have performed updates that can change the content of the fragments in $access$. If cache and server perform different updates, fragment data would be changed differently and we cannot necessarily conclude that the query was executed on equal data. However, if both have performed the same modifications on data, denoted by $seq' = seq''$, data has been equally modified and cache and server would compute the same result for e_n. Note that seq' and seq'' also contain the generated tuple identifiers. Hence, both sequences have also generated equal identifiers.

5.4.4 Completing an Execution

After the completion of a procedure execution, the server enters its commit phase. Recall from Section 4.4.4 that the Algorithm 4.2 is performed within the commit phase of each transaction at the server. The algorithm increments the version of each modified fragment and puts executed low-level updates into the *fragment access history* that is the base for the synchronization scheme.

Finally, the server sends the result of the execution to the cache and the execution is completed. The result is either a value (single value, a tuple or a table) or an error message in case the transaction has been rejected.

5.5 Identify Dependencies within the Procedure Code

As pointed out in Section 5.3.1, the partial execution at the cache is very liberal and tries to execute as many of the statements of a given *split* parameter as possible. The scheme is able to handle all $split \subseteq ID(S)$ parameters. However, due to dependencies in the procedure code, some of the *split* parameters may cause $undef$ values to be assigned to local variables which can prevent queries in *split* from being executed. In these cases, no query result is delivered to the server, although expected. Furthermore, such dependencies could cause invalid query results to be sent to the server. The following examples show such dependencies.

Consider for example the `AddCart` procedure from our web shopping application. The procedure increases the request counter for a product, checks the session and aborts the execution in case of invalid sessions. We add some extra code that counts the clicks for prime users (e.g., that might get a special discount for the number of clicks per month). The available amount of the product is checked and, if available, the product is added to the shopping cart. Otherwise the procedure aborts.

```
PROCEDURE AddCart(sessID,prodID,amount)

UPDATE Statistic SET requests=requests+amount WHERE pid=prodID;
session := (SELECT * FROM Session WHERE sid=sessID);
IF session is null THEN RETURN Error;
IF session.userType='PrimeUser' THEN
   UPDATE User SET clicks=clicks+1 WHERE uid=session.userID;
inStock := (SELECT storage >= amount FROM Stock WHERE pid=prodID);
IF NOT inStock THEN RETURN Error;
INSERT INTO Cart VALUES(sessID,prodID,amount);
RETURN SuccessCode
```

We discuss the dependencies of the procedure code on the basis of the resulting low-level procedure where `INSERT`, `DELETE` and `UPDATE` statements have been replaced accordingly (see Section 4.2.3).

```
PROCEDURE AddCart(sessID,prodID,amount)
```

```
 1  FOR row IN SELECT ID,* FROM Statistic WHERE pid=prodID LOOP
 2     row.requests=row.requests+1;
 3     delete(Statistic,row.ID);
 4     insert(Statistic,row.ID,row);
 5  END FOR;
 6  session := (SELECT * FROM Session WHERE sid=sessID);
 7  IF session is null THEN RETURN Error;
 8  IF session.userType='PrimeUser' THEN
 9     FOR row IN SELECT ID,* FROM User WHERE uid=session.UserID LOOP
10        row.clicks=row.clicks+1;
11        delete(User,row.ID);
12        insert(User,row.ID,row);
13     END FOR;
14  END IF
15  inStock := (SELECT storage >= amount FROM Stock WHERE pid=prodID);
16  IF NOT inStock THEN RETURN Error;
17  insert(Cart,(sessID,prodID,amount));
18  RETURN SuccessCode
```

In total there are 9 IO statements. Again, we use the line of code to identify them. Hence, `ID(AddCart)`$= \{1, 3, 4, 6, 9, 11, 12, 15, 17\}$. A partial execution at the cache is defined by any *split* $\subseteq$ `ID(AddCart)`. We discuss the most representative settings of the *split* parameter and the resulting problems with code dependencies.

- *split* $= \{11\}$: The cache only has to execute 11. However, 11 is never executed, since the 6 and 9 are not executed. If 6 is not executed, $undef$ is assigned to the variable `session` and the entire `IF` statement (line 8-14) will be not executed by the engine. If 9 is not executed, the entire loop (line 8-13) is not executed. Hence, all *split* parameters that include 11, but not 6 and 9 are equal to *split* $= \emptyset$.

- *split* $= \{12\}$: Analogously to *split* $= \{11\}$ but with an additional data dependency, since 12 operates on the same table (possibly fragment) as 11.

- *split* $= \{6, 9, 11, 12\}$: No conflicts. However, the execution might already be aborted at line 7. Since the cache ignores these aborts (see **Rule 7** in Section 5.3.1), it might perform erroneous updates on local data in line 11 and 12. Recall that those conflicts are detected by the verification scheme that takes performed low-level updates at cache and server into account. Furthermore, after an execution completes at a cache, all updates are undone. Hence, none of the erroneous updates are applied to cache data.

- *split* $= \{1, 6, 9, 15\}$: The cache would execute all queries and send the results to the server. Note that the result of 9 and 15 are not considered by the server, if the execution is aborted at line 7.

- *split* $= \{15\}$, *split* $= \{6, 9, 15\}$: Analogously to *split* $= \{1, 6, 9, 15\}$.

We are interested in those *split* parameters which cause the smallest possible number of $undef$ values in the procedure code and that execute queries in *split* at the cache. Note that we cannot determine the precise number of $undef$ values at run time, since their occurrence also depends on the user input (input parameters of stored procedures) as shown by the following piece of procedure code:

```
1 IF <cond1> THEN
2       SELECT ID INTO var1 FROM R1 WHERE <cond2>;
3 ELSE
4       SELECT ID INTO var1 FROM R2 WHERE <cond3>;
5 END IF;
6 SELECT ID INTO var2 FROM R3 WHERE var1=<expr>;
```

The `IF` statement assigns each branch a local variable `var1` that is used by the query in line 6. Again the IO statements are identified by the line of code they appear in. Consider $split = \{2, 6\}$. Clearly, whenever `<cond1>` returns true, the query in line 6 is properly executed and no $undef$ value is assigned to `var1`. However, in case `<cond1>` returns false, the query in line 4 is not executed and $undef$ is assigned to `var1`. Hence, for $split = \{2, 6\}$ the number of $undef$ values at line 6 depends on the evaluation of `<cond1>` at run time. Since the evaluation of `<cond1>` can also depend on the input parameters of a procedure call, the number cannot be predicted in advance. Hence, the split $\{2, 6\}$ is only useful if the first branch of the `IF` statement is executed frequently at run time.

The example has shown code dependencies and their impact on $undef$ values at run time. However, there are also dependencies among the IO statements (low-level updates and queries). The above example with $split = \{12\}$ demonstrates such a case, where the statement 12 depends on 11, since both operate on the same table and possibly on the same fragment (depends on the chosen fragment column). However, the following example demonstrates more precisely how these dependencies can cause invalid query results. Consider

```
1 insert(R,(var1));
2 insert(R,(var2));
3 SELECT count(*) INTO var3 FROM R;
```

which inserts the tuples `(var1)` and `(var2)` into the table `R` and finally counts all tuples in `R`. Query 3 never causes an $undef$ value, since it is neither related to input parameters nor to local variables. However the result of query 3 is only correct if the inserts 1 and 2 are performed. That is, the parameters $split = \{1, 3\}$ and $split = \{2, 3\}$ will always produce an erroneous query result at the cache which will be detected by the servers $eval_S(.)$ function in the Steps 1.f.i and 1.g.i. In these Steps, the server checks whether a cache has also performed all low-level updates that influence the result of the query ($seq' = seq''$). For the parameters $split = \{1, 3\}$ and $split = \{2, 3\}$, this will not be the case, since the delivered result is always invalid at the server. Hence, the split parameters $\{1, 3\}$ and $\{2, 3\}$ are not useful.

Possibilities for Detecting Code and Data Dependencies

As the examples have shown, not all split parameters in $2^{ID(S)}$ are useful for split twin transactions. We are looking for a set $Z \subseteq 2^{ID(S)}$ of split parameters that causes a small number of $undef$ values and erroneous queries. In general there are three options to determine the set Z. These are:

1. Use a given set Z that is provided by the administrator and developers.

2. Derive Z from a static analysis of the procedure code.

3. Obtain Z by a run time analysis.

Clearly, the second requires a semantic analysis of the procedure code which is almost impossible due to the complexity of current stored procedure and query languages. Furthermore, input parameters of procedures cannot be predicted. In the following, we show how to implement a run time analysis.

Run Time Analysis

Intuitively, a run time analysis can be performed as follows: Given a procedure S and the set $2^{ID(S)}$ of all its split parametes, S is executed multiple times for each $split \in 2^{ID(S)}$ and for different input parameters. For each $split$ we observe the number of $undef$ values (Steps 2a and 3a of $eval_C(.)$) and invalid queries (Steps 1gi and 1fi of $eval_S(.)$) at run time. Finally, we filter out all these split parameters with a high number of $undef$ values and invalid queries.

To perform this analysis, the cache must be set to a testing mode (non-production mode) where:

- All fragments are replicated in a transactional manner to the cache. Hence, all data is available and up-to-date. No $undef$ values occur at the cache that result from missing data and no invalid queries occur that result from stale data.

- The normal daily work load is applied to the cache, such that all procedure calls are performed with real parameters. As shown by above examples, the input parameter can affect the resulting number of $undef$ values. Furthermore, the repeated execution of procedures captures the different execution paths at run time, e.g., branches of `IF` statements, number of loop repetitions.

While running the testing mode, no optimization is applied. Note that the testing mode does not affect the execution at the server as defined in Section 5.4.

Assume that such a test has been performed and each procedure has been executed multiple times for all possible $split$ parameters. For each procedure S and each $split \subseteq ID(S)$ we obtain a function $f_{S,split} : split \mapsto [0, 1]$ that assigns each IO statement $s \in split$ a frequency of $undef$ values, and a function $g_{S,split} : split \mapsto [0, 1]$ that assigns each query in $s \in split$ a frequency of invalid query results.

Definition 5.5 *Let S be a procedure and a $split \subseteq ID(S)$ parameter. Let S be executed multiple times for different input parameters. Let $s \in split$ be an IO statement and $n_s > 0$ be the resulting total number of functions calls $eval_C(s, \#s, e)$ for some $\#s$ and e. Let further be $0 \leq n'_s \leq n_s$ be the number of calls where $eval_C(s, \#s, e)$ returns $undef$ in Step 2a or 3a. Then,*

$$f_{S,split}(s) \quad = \quad \frac{n'_s}{n_s}$$

and $f_{S,split}(s) = 1$ for $n_s = 0$.

The case $n_s = 0$ means that s has never been executed during the whole test. One reason for this might be that the path with s has never been followed through the procedure code or s is part of an `IF` or `LOOP` statement which was not executed due to previous *undef* values. In the first case, s has no impact on the systems performance and the second case is caused by *undef* values. Hence, for $n_s = 0$ we set $f_{S,split}(s) = 1$ which means that s is treated as an *undef*-causing statement which therefore is not desired.

Consider again the example at the beginning of this Section. For $split = \{11\}$ we get $f_{S,split}(s) = 1$, since 11 cannot be executed due to an *undef* value in the condition of line 8. For $split = \{15\}$ we expect $f_{S,split}(s) = 0$, since the query can always be executed independently from the previously executed code.

Definition 5.6 *Let S be a procedure and a $split \subseteq ID(S)$ parameter. Let S be executed multiple times for different input parameters. Let $s \in split$ be a query and $n_s > 0$ be the resulting total number of functions calls $eval_S(s, \#s, e)$ for some $\#s$ and e, where the Steps 1fi or 1gi are performed. Let furthermore $0 \leq n'_s \leq n_s$ be the number of calls where $eval_S(s, \#s, e)$ determines an invalid query result in these Steps. Then,*

$$g_{S,split}(s) \quad = \quad \frac{n'_s}{n_s}$$

and $g_{S,split}(s) = 1$ for $n_s = 0$.

The definition only considers the Steps 1fi or 1gi of $eval_S(s, \#s, e)$, since they determine the correctness of a query result. Note that a call of $eval_S(s, \#s, e)$ does not necessarily execute these Steps in case of *undef* values. The transactional data replication guarantees that each query at the cache is executed on actual data, hence $access = access(e_n)$ holds in both Steps 1fi or 1gi. The correctness of a query is only determined by the evaluation of $seq' = seq''$. Again, we treat queries that have not been executed as invalid queries, since such queries have no impact on the system's performance. Consider again the above example with the split parameters $\{1, 3\}$ and $\{2, 3\}$. For both we expect $g_{S,split}(s) = 1$ for the query 3.

Filtering the Set of Possible Split Parameters

Based on the observed run time statistics, we define the set of relevant split parameters of a procedure.

Definition 5.7 (Relevant Set of Split Parameters) *Let S be a procedure and $f_{S,split}$, $g_{S,split}$ the functions as defined above. Then, the set of relevant splits of S is defined by*

$$\begin{aligned} Z(S) \quad = \quad \{split \mid split \subseteq ID(S) \wedge \\ \forall s \in split : f_{S,split}(s) < \alpha \wedge \\ \forall s \in split : s \text{ is a query} \Rightarrow g_{S,split}(s) < \beta\} \end{aligned}$$

with α and β as a user-defined upper bounds for the frequencies of $undef$ values and invalid queries.

Note that we use this filter as a preselection for dropping useless split parameters which are mainly those with $f_{S,split}(s) \approx 1$ and $g_{S,split}(s) \approx 1$. For this a range of $(0.9, 1)$ is sufficient for setting the bounds. In Part III we show how to select split parameters out of $Z(S)$ that yield optimal run time performance.

5.6 Correctness of Split Twin Transactions

In Section 4.7, we have shown that a twin transaction produces equal execution sequences on cache and server, if both executions operate on equal fragment versions. Since a twin transaction executes the entire procedure code, we cannot directly apply the resulting Propositions for a split twin transaction that partially executes the code at the cache.

However, on an intuitive level, the execution scheme is correct if an execution with a database cache produces the same effect on server data as an execution without a database cache. Without our scheme, the server executes a function $eval(.)$ that processes individual IO statements. We have replaced this function by $eval_S(.)$ which adds additional behavior during the execution of a low-level procedure. Since the additional behavior in $eval_S(.)$ does not change the content of the user tables $R_1, \ldots, R_n$, we only have to show that $eval_S(.)$ produces the same result as $eval(.)$.

Proposition 5.1 (Correctness) *Let $eval_S(s_1, \#s_1, e_1), \ldots, eval_S(s_n, \#s_n, e_n)$ be all function calls at the server within a split twin transaction. Then, $eval_S(s_i, \#s_i, e_i) = eval(s_i, \#s_i, e_i)$ holds for all $1 \leq i \leq n$.*

Proof: Given a call $eval_S(s_i, \#s_i, e_i)$ with $1 \leq i \leq n$ (see also Section 5.4.2), we distinguish between the following two cases: In the first case, $eval_S(s_i, \#s_i, e_i)$ performs neither Step 1.f.i nor 1.g.i. Then, no query result is reused and $eval_S(s_i, \#s_i, e_i)$ returns the result of $eval(s_i, \#s_i, e_i)$. Hence, $eval_S(s_i, \#s_i, e_i) = eval(s_i, \#s_i, e_i)$ holds.

In the second case, $eval_S(s_i, \#s_i, e_i)$ performs either Step 1.f.i or 1.g.i. Then, the query e_i is executed by the cache, its result is reused by the server and $access = access(e_i)$ and $seq' = seq''$ holds. We use the notations seq' and seq'' as they appear in the function $eval_S(.)$. In the remaining part of the proof we show that indeed $eval_S(s_i, \#s_i, e_i) = eval(s_i, \#s_i, e_i)$ also holds for the second case.

The condition $access = access(e_i)$ states that the query e_i at the cache has been executed on the same fragments and their versions, say $\{(F_1, v_1), \ldots, (F_m, v_m)\}$, as the server is accessing in its execution of $eval_S(.)$. Hence, if the server would have executed the query e_i, it would have executed e_i on the same data. The condition $seq' = seq''$ states that the cache and the server have executed the same low-level updates on the fragment $\{F_1, \ldots, F_m\}$ during the twin transaction before e_i was executed.

We conclude that both executions of the twin transaction have been started on the fragments and their versions $\{(F_1, v_1), \ldots, (F_m, v_m)\}$ and that the cache and the server applied the same low-level updates on these fragments. Hence, on an intuitive level cache and server execute the query e_i on identical data and the result will be equal.

To show this property by using Proposition 4.3, we construct the following low-level procedure S (we omit the declaration and head part):

```
BEGIN
      $u_1$;
      ..
      $u_k$;
      FOR row IN $e_i$ LOOP
           ..
      END LOOP
END
```

It consists of low-level updates $u_1, \ldots, u_k$ that have been applied by the cache and the server and the query e_i that is executed by the cache, but not by the server. Note that the sequence of low-level updates $u_1, \ldots, u_k$ represents the partial execution at the cache.

By Proposition 4.3 we conclude that the execution of S as a twin transaction yields equal execution sequences at the cache and the server. Since these sequences also include the result of the query e_i, both executions of S must have produced the same result for e_i. Hence, the query result that is delivered by the cache corresponds to the execution of the function $eval(.)$ at the server and $eval_S(s_i, \#s_i, e_i) = eval(s_i, \#s_i, e_i)$ holds for the second case. $\Box$

5.7 Related Work

Simply speaking, the cache delivers a query expression e and a result val for e, and the server has to verify if val is a correct result for e at shared server data. Among other tests, the function $eval_S(.)$ also checks if cache and server have executed the same low-level updates on data. These updates are captured by the execution sequence seq'' (see Step 1e in $eval_S(.)$). The more precise the impact of these updates on the result val is known, the less noise appears in seq'' and the more query results can be reused by the server.

This problem is well-known in the field of *materialized view maintenance*, wherein e is a view definition and val represents the materialized view. For a given set of updates on data, the problem is to determine whether val still represents the result of e or if val has to be recomputed. A simple mechanism is to check table access. If the updates have been applied on tables that are not in the view definition e, the result val is still correct. However, as summarized by [48], a semantic analysis of e and the updates further improves this check. For example, an `UPDATE` on table `R` is independent of the query `SELECT count(*) FROM R`, allthough both operate on the same table. `INSERT` and `DELETE` statements however are not. These techniques usually provide different solutions for different types of queries, since in general, not all types can be handled correctly, e.g., aggregations and sub-queries impose limitations.

Our verification scheme can be further improved by integrating those techniques in Step 1e in $eval_S(.)$ where the updates seq'' are computed that affect the result (or view).

We have chosen a partial execution scheme that relies on a split of the procedure code. Hence, our scheme does not require transforming the original procedure code. However, in literature, similar problems have been studied in the field of parallel processing for multi processor systems and compiler construction, e.g., [3, 7, 103]. These methods could be used for improving the partial execution and for determining relevant split parameters. In the field of rule triggering systems we have discussed in [57] independencies within the trigger code.

5.8 Summary and Discussion

On top of the client-server database system, we have developed split twin transactions and a simultaneous execution scheme for its proper execution. The scheme partially executes a low-level procedure at a cache and the server simultaneously, whereby the cache sends the result of each query to the server. The server tries to re-use these results instead of executing them by itself. Thus, the execution of these queries has been shifted from the server to the cache. The amount of computation at a cache is controlled by the *split* parameters for each procedure. Thus, allowing to dynamically balance the load at run time by varying the setting of the *split* parameters. In Section 6 we study the impact of different settings on the performance of the client-server system. In Part III we define the performance and propose a run time optimizer that identifies the optimal setting of the parameters.

In the following we discuss the open issues that arise in the course of this Chapter.

Early Notifications

The cache puts a notification into the QRC before a query is executed. Based on this notification, the server verifies the incoming result, thus being able to decide whether to use the incoming result or to execute the query by itself. Since the server itself might wait for a notification to appear in the QRC, it also waits for a possible wrong result. In that case, the resulting idle time would be a waste of resources at the server.

Hence, the earlier a notification is sent, the earlier the server can determine the correctness of an incoming query result. Consider again the low-level procedure in the beginning of Section 5.5. As we have argued, the query on table `stock` in line 15 only depends on the input parameters. Hence, the execution at the cache can already send the notification of that query at the beginning of the execution. To determine the fragment access, it only has to instantiate the query expression. As a result, the notification is put into the QRC at the beginning of an execution and the server does not have to wait for it. The reduction of idle time at the server further improves the total execution time. An implementation of this technique requires performing look-aheads while executing the procedure code.

Normal Mode versus Shared Mode

A split twin transaction starts in shared mode that causes the server to consider delivered query results from the cache. Furthermore, setting a new version requires all updates on a fragment to be equal to the servers execution. However, we switch to normal mode if an invalid result has been delivered. As a result, the server does not consider any more delivered results until the split twin transaction is completed. We have chosen this behavior, since an invalid query result, that results from missing or stale data at the cache, can cause more invalid query results to be delivered to the server.

Alternatively, the server can always run in shared mode, so that for each query the QRC is checked for an appropriate result. However, the server then has probably to wait for notifications and results even if they are invalid. To consider alternative settings of the shared mode, we would have to investigate the probability of valid query results that are delivered after an invalid query has already been delivered. Again, this depends on the dependencies in the procedure code. Since we have not performed such an analysis, we stick to the above setting of the shared mode.

Commit or Reject Updates at the Cache?

Our scheme undoes all updates that have been made during an execution at a cache. This is useful, since a cache might have performed erroneous updates which can cause inconsistencies at the cache. However, if a cache has operated on the same fragment version as the server and has performed exactly the same updates as the server in its execution, these updates can be committed and cache data is consistently updated.

Such a scenario requires an extension of the commit phases at cache and server, as well as the synchronization process at the cache. The commit at the server has to send all applied updates to the cache, so that the cache is able to compare these updates to its performed updates. Furthermore, the synchronization process at the server must be aware of these applied updates, so that once these updates are propagated by the server, they are not applied at the cache again. This extention can be easily integrated into our scheme. The commit at the cache would increase the local version numbers as done by the server and the synchronization process at the cache only has to check the local version of a fragment before updates are applied. If the version at a cache is already up-to-date, updates are not applied.

This commit possibly further improves the up-to-datedness of cache data, since then propagated updates are not delayed by the optimistic synchronization scheme for correct transactions.

Keep Query Results in the QRC

After a split twin transaction terminates, all corresponding entries are removed from the QRC. Furthermore, an execution at the cache only considers query results from the corresponding twin transaction. A possible extention of our scheme can consider the QRC in a wider scope. Hence, executions at the server might also use query results from previous executions, if correct, or even results that have been put by other caches in the meantime. Of course, this would increase the search time for valid results in the QRC, but the number of reuse cases could potentially be increased.

Effect on the Procedure Code

An interesting question is whether our approach affects the programming of the procedure code by developers. To support our scheme, developers can code a procedure in two sequential parts, say A and B. Part B is designed, so that it does not depend on part A, hence can be computed independently by a cache. Furthermore, developers can provide possible split parameters.

Since our scheme detects fragment access on a syntactical basis, all queries in the procedure code, if possible, should contain the fragmentation attribute of the involved tables. Consider the following query on a table `R` with a fragmentation column `productGroup`: `SELECT productName INTO <var> FROM R WHERE productID=1`. Since the query does not contain the fragmentation column, our scheme assumes full table access. However, a single tuple in `R` always determines the value, say x, for the fragmentation column, such that fragment access can be improved by adding `productGroup=x` into the `WHERE` clause.

Chapter 6

Case Study — The ONE-System

The goal of this Chapter is to show that a simultaneous execution scheme yields a better performance than the traditional detection-based protocol. The ONE-System is a prototype implementation of the simultaneous execution scheme. We run various experiments that analyse the relationship between the configuration (split) parameters and the response time and throughput of the system.

This Chapter is structured into four parts. First, we present the architecture of the ONE-System and explain how it is implemented. Second, we define the experimental setup which includes the execution of experiments. Third, we compare both schemes for a one-client and a four-client system by analysing several experiments. Finally, we discuss the performance of a slow database cache and the performance of a modified version of our scheme that does not switch to normal mode for invalid queries.

6.1 Architecture

The ONE-System is a minimal system, since it allows only *one* stored procedure. We show how the detection-based and the simultaneous execution protocol are implemented and give details of the query result verification technique and data replication.

6.1.1 Database Schema and Procedure Code

The database system consists of a single table R which contains tuples `(objID,data,version)`, with `objID` as a primary key, `data` as a binary representation of the object and `version` as the version number of the object starting with 0. Each update of an object will increase the version number by one. R is split into fragments where each fragment contains exactly one object. Hence, fragment access is detected in terms of object identifiers. For R we define the following parameters:

1. `objectSize` - size of an object (column "data") in Byte

2. `tableSize` - number of tuples in R

For simplicity we assume that the identifiers of the objects are in a range from 1 to `tableSize`. In experiments we vary this parameters to study its impact on the systems performance.

There is one stored procedure S that operates on R. It consists of 20 sequential steps. According to the input parameter, each step selects an object ID and retrieves the binary object from R. Finally, S passes back all 20 binary objects. To simulate an expensive query, we perform a join on R. The procedure code is:

```
CREATE PROCEDURE S(o1,..,o20) RETURNS ROW AS
DECLARE o1,..,o20 AS integer;
DECLARE d1,..,d20 AS blob;
BEGIN
      SELECT t1.data INTO d1 FROM R AS t1, R AS t2
        WHERE t1.objID=t2.objID=o1;
      SELECT t1.data INTO d2 FROM R AS t1, R AS t2
        WHERE t1.objID=t2.objID=o2;
      ..
      SELECT t1.data INTO d20 FROM R AS t1, R AS t2
        WHERE t1.objID=t2.objID=o20;
      RETURN d1,..,d20;
END
```

The code contains 20 statements (queries) that are enumerated from 1 to 20. Hence $ID(S) = \{1, \ldots, 20\}$. Obviously, all queries are independent and can be executed in any order without spoiling the final result. Therefore, S can be executed for all possible combinations given by $split \subseteq ID(S)$.

6.1.2 The Client-Server Database System

The server is connected with up to four clients, each running a database cache. Procedure calls are exclusively triggered at clients. The server is responsible for modifying and replicating objects in R.

Data Modification at the Server

There is a special server process that constantly generates updates on R. We use the parameter `updateLoad` to simulate a predefined frequency of updates. It defines the sleep time in milliseconds between two updates of R.

This process picks randomly an object in R, modifies its binary representation and increases the version number by one. Each update u is registered in the table LOG by adding a tuple

$$(objID, u)$$

Note that we neglect the syntax of updates since it does not affect the experimental analysis. This log is also accessed by the synchronization process and the query result verification.

Procedure Calls at a Client

Each client is a single user machine that constantly initiates the procedure S in a sequential manner. To vary the load of the system, we use a parameter `execLoad` that specifies the idle time in milliseconds between two procedure calls. The database cache at a client executes S either with a traditional or the simultaneous execution protocol.

Data Replication

The ONE-System is primarily used to investigate the impact of different split parameters on the system's performance. Therefore, all data is replicated, such that each procedure call can be executed on its own cache data.

For the experiments in this Chapter, we use a modified version of the synchronization process. In contrast to the original process, as proposed in Chapter 4, updates are not immediately propagated, but are delayed for a given period. We use this modification to vary the number of inconsistent objects at a cache. The larger the delay, the higher is the number of inconsistent objects at the cache. The smaller the delay, the faster are updates propagated.

To control the frequency of the synchronization and the resulting impact on the consistency of cache data, we use the parameter `updateDelay` that defines the sleep time in milliseconds between the synchronization phases of the process. A phase is defined as follows. First, it reads and removes all tuples from the log. If such tuples exist, the corresponding updates are applied to the cache data in the same order as they appear in the log. After a period of `updateDelay`, the next phase is triggered. Depending on the setting of `updateLoad`, new tuples will be added by the load-generating process at the server.

Hence, a low value indicates frequent synchronization with a low number of updates to propagate and a high value indicates less frequent synchronization with a high number of updates. Note that updates do not reside for long in the log and that the size of the log is rather small. Further, we do not need to run a transaction for operations on the log, since the load process only adds tuples and the synchronization process only removes tuples. Hence, there can be no conflicts. As a result, we expect fast operations on the log table.

6.1.3 A Traditional Execution Scheme

In Chapter 3 we have introduced two traditional execution schemes for database caching. In the following, we present a primitive implementation of a detection-based protocol. Such protocols execute the entire code sequential at the cache and decide for each query, if it is executed locally or send to the server. Since we use a detection-based protocol, each computed query at the cache is verified by the server.

Such a protocol is implemented by the following procedure code for a given a *split* parameter. We use the notation `< EXECUTE ON SERVER: SELECT .. INTO` x `FROM .. >` to denote that a cache sends a query for execution to the server and that the assigned variable x is send back to the cache. We present only the code for one step out of the 20. The others follow analogously. The variable i runs for 1 to 20 and *split* $\subseteq \{1, \ldots, 20\}$, such that the code block between the `..` is repeated 20 times.

```
CREATE PROCEDURE S(o1,..,o20) RETURNS ROW AS
          // Define local variables.
DECLARE o1,..,o20,counter AS integer;
DECLARE d1,..,d20 AS blob;
DECLARE mode AS {shared,normal};
BEGIN
     mode=shared;
     ..
     IF $i \in split$ AND mode=shared THEN
       SELECT t1.data INTO d$i$ FROM $R$ AS t1, $R$ AS t2
         WHERE t1.objID=t2.objID=o$i$;
       EXECUTE ON SERVER: // Retrieve number of updates on the object.
         SELECT count(*) INTO counter FROM $LOG$ WHERE objID=o$i$;
       IF counter>0 THEN
         // When the object was updated, data at the cache are not
         // consistent and the query is re-executed at the server.
         EXECUTE ON SERVER:
           SELECT t1.data INTO d$i$ FROM $R$ AS t1, $R$ AS t2
           WHERE t1.objID=t2.objID=o$i$;
         mode=normal;
         // When a re-execution has been taken place, the execution
         // mode is changed to "normal mode" where the server does
         // not reuse query results from the cache within the current
         // execution of the procedure $S$.
       END IF
     ELSE
       EXECUTE ON SERVER:
         SELECT t1.data INTO d$i$ FROM $R$ AS t1, $R$ AS t2
         WHERE t1.objID=t2.objID=o$i$;
     END IF;
     ..
     RETURN d1,..,d20
END
```

When a query i appears in *split*, it is executed at the cache. The result is verified by checking whether the log table contains an update (which has not been propagated yet) that has modified the data of object oi. If such an update exists, cache data is inconsistent and the query is sent to the server for execution. If a query does not appear in *split*, it is also executed by the server.

Note that our simultaneous execution scheme distinguishes between shared and normal mode (see Section 5.4.1). The testing mode is not considered here, since it only affects the run time optimization as explained in Part III. A server in shared mode awaits query results from the cache and in normal mode no more results are considered by the server. We have chosen this strategy, since in general an invalid query result at the cache can cause more invalid results.

We have added this behavior to the above procedure code, to make the traditional and our scheme comparable. Then, both schemes behave similarly on invalid queries. However, as we will discuss in Section 6.4 we achieve the same experimental results if the schemes do not switch to normal mode.

6.1.4 The Simultaneous Execution Scheme

In Chapter 5 we have defined the simultaneous execution scheme where cache and server execute a procedure in parallel. Given the parameter *split*, the cache executes only queries in *split* and passes the result of the computation to the server which reuses the result instead of computing it.

For the experiments we use a simple version of our scheme that does not use notifications to indicate the start of a query execution at a cache. This is feasible, since the given procedure executes all queries in a sequential manner in any case, and since this implementation is sufficient to outperform the traditional scheme. Note that we expect a better performance with notifications, since the server can start the verification before the result has been delivered to the server. Hence, the server can already re-execute the query, while the cache is still executing the comming invalid result. However, we further discuss this issue in Section 6.4.

We use the notation `< SEND TO SERVER:` $(i, oid, data)$ `>` where $i \in ID(S)$, *oid* an object ID and *data* the result of a query. Note, that we add the client identification number to the message if more than one client is attached to the server. However, we provide the procedure code for one client only.

Again, we present the code for one step i out of 20 and omit the declaration part that is equal to that of the traditional scheme. A procedure code at the cache is defined as

```
BEGIN
        ..
        IF $i \in split$ THEN
           SELECT t1.data INTO d$i$ FROM $R$ AS t1, $R$ AS t2
              WHERE t1.objID=t2.objID=o$i$;
           SEND TO SERVER: ($i$,o$i$,d$i$);
        END IF;
        ..
        RETURN
END
```

The server checks for each query in *split* whether an appropriate result has been delivered by the cache. If not, it waits for the result. The result is verified and reused if correct. Otherwise the query is re-executed at the server.

```
BEGIN
        mode=shared;
        ..
        IF $i \in split$ and mode=shared THEN
           WAIT FOR MESSAGE ($i$,o$i$,d$i$) FROM CACHE;
           SELECT count(*) INTO counter FROM $LOG$ WHERE objID=o$i$;
           IF counter>0 THEN
              SELECT t1.data INTO d$i$ FROM $R$ AS t1, $R$ AS t2
                 WHERE t1.objID=t2.objID=o$i$;
              mode=normal;
           END IF;
        END IF;
        ..
        RETURN d1,..,d20
END
```

The verification is performed analogously to the traditional scheme by checking the log table for updates. Again, we use the switch to the normal mode if an invalid query has been delivered to the server.

6.2 Experimental Setup

This section defines the experimental setup which includes the used hard- and software, the setting of parameters, the execution of experiments, and the performance measures of an experiment.

6.2.1 Used Hard- and Software

As we choose a machine with 2x800MHz CPU and 1024 MB main memory. The experiments use one or four clients which are also machines with up to 1GHz CPU and up to 1024MB main memory. All machines are connected by a 100MBit network and run with Linux. As a database system, we have tested DB2, Postgres and MySQL. We will not compare the performance of these systems. Rather, we present only experimental results for one of these systems in subsequent sections. The other systems yield similar results as presented in this chapter.

For implementing stored procedures on top of the database system, we have used PHP script that performs the SQL statements at the database. Note that PHP is only used to perform these SQL statements, to start and commit transactions, and to measure the performance, e.g., processing time of a procedure. Within the execution of a single procedure we have observed a maximal overhead of 2-4ms for executing the PHP code.

6.2.2 Parameters of Experiments

In Section 6.1 we have defined parameters for configuring the ONE-System. During the execution of experiments, we distinguish between two types of parameters. Static parameters are initially set at the beginning and not changed during an experiment. Run time parameters are changed during the experiment, since we want to observe their impact on the systems performance.

Static Parameters

The parameters `objectSize`, `tableSize`, `updateLoad` and `execLoad` are fixed. Alltogether we have performed over 100 experiments with different settings of these parameters, see Table 6.1. In subsequent experiments we present only two of these combinations.

Parameter	#	Values
`objectSize`	8	1, 5, 10, 50, 100, 500, 1000, 5000 KB
`tableSize`	8	1000, 2000, 5000, 10000, 50000, 100000, 500000, 1000000 records
`updateLoad`	5	0 ms, 200ms, 500ms, 1000 ms, 2000 ms
`execLoad`	4	0 ms, 50 ms, 100 ms, 500 ms
`waitingTime`	1	5 sec
`triggerPeriod`	1	80 sec

Table 6.1: Range (Domain) of static parameters.

Note that it is not the scope of these experiments to identify situations where database caching is useful. Rather, we consider situations where the traditional scheme improves the performance and compare the resulting performance to our scheme. However, as expected, database caching is not useful for small values of `updateLoad` and less complex queries (e.g. small table size). In the first case there is a high synchronization effort and a high number of invalid queries and in the second case the server can always perform such queries very efficiently.

Run Time Parameters

In the experiments we analyse the impact of two run time parameters on the system's performance. As mentioned in previous chapters, the main parameters of the simultaneous execution scheme is the split of the procedure code that is captured by $split \subseteq ID(S)$ of a low-level procedure S, and the amount of replicated data that is captured by a set of fragments *repl*. However, for the ONE-System we only analyse the impact of the *split* parameters. We distinguish between two groups of *split* parameters — without and with independent queries.

The first group is used for testing the traditional and our scheme for a sequential computation of the procedure code. Hence, the cache cannot 'jump' over queries and has to execute all queries as they appear in the procedure code. For this we will consider *split* parameters of $\emptyset$, $\{1\}, \{1, \dots, i\}$ with $2 \leq i \leq 20$.

The second group is only used for our scheme with independent queries. Hence, the cache can 'jump' over certain queries of the procedure code. For this we use $\emptyset$, $\{20\}, \{i, \dots, 20\}$ with $1 \leq i \leq 19$. Consider for example $split = \{18, 19, 20\}$. The cache only executes the queries 18,19,20 and sends their results to the server. The server executes queries 1 to 17 in parallel to the cache and reuses the results of 18,19,20 after it has completed the queries 1 to 17.

Additionally, we use `updateDelay` as run time parameter. The parameter is used for varying the amount of valid query results that are delivered to the server. Recall that data replication within the simultaneous execution scheme is asynchronous. Hence, updates are not performed within transactional boundaries and as a consequence a cache can operate on stale data. The amount of valid query results is controlled by 13 possible levels of `updateDelay`. These are: 100ms, 150ms, 200ms, 300ms, 500ms, 750ms, 1sec, 2.5sec, 5sec, 10sec, 15sec, 20sec, 25sec. The time specifies the idle time between synchronization phases. For high delays we expect a low number of valid results and for low delays a high number of valid results.

As explained in the next section, we measure the systems behavior for each combination of *split* and `updateDelay`. Alltogether we have 273 possible combinations for the first and second group.

6.2.3 The Execution of Experiments

All experiments are executed under equal conditions: The machines exclusively run the experiments and there is no relevant traffic on the network. The server constantly performs updates with a delay of `updateLoad`. The synchronization process periodically propagates all new updates to caches. Each experiment is executed as follows:

1. Choose a *split* parameters out of the 21 possible levels.
2. Choose an `updateDelay` parameter out of the 13 possible levels.
3. Wait `waitingTime` (5 seconds).
4. Each cache triggers the stored procedure $S(v_1, \dots, v_{20})$ sequentially over a period of `triggerPeriod` (80 seconds). Between each call there is a pause of `execLoad`. For each call, the values $v_1, \dots, v_{20}$ are randomly computed with $1 \leq v_i \leq$ `tableSize`.
5. Goto 1, as long there is a combination of *split* and `updateDelay` that has not been tested yet.

The break of 5 seconds is used to allow the synchronization process for small values of `updateDelay` to perform intermediate synchronizations with the new parameter. This is necessary, since a high value might cause a high number of inconsistencies that would be taken over into the next test and lead to wrong experimental results.

Performance Characteristics

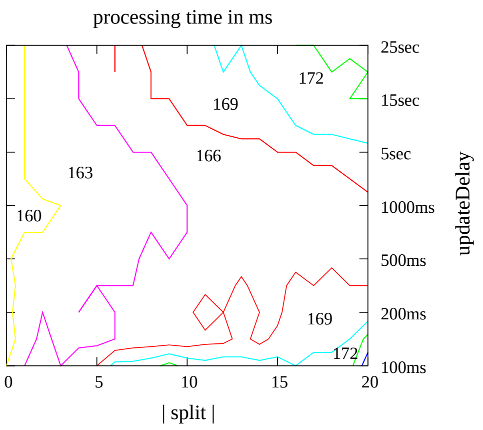
For each combination of *split* and `updateDelay`, we observe the following values in Step 4 within the trigger period of 80 seconds:

- The *processing time* of S is defined as the average time between the initiation of a procedure call and arrival of its final result at the cache.
- The *cumulated verification time* is defined as the average execution time of the query `SELECT count(*) INTO c`i `FROM` LOG `WHERE objID=o`i. In the following experiments we only use the cumulated verification time which is the sum of all query verifications during the execution of a procedure.
- The *query time* is defined as the average execution time of a single query `SELECT t1.data INTO d`i `FROM` R `AS t1,` R `AS t2 WHERE t1.objID=t2.objID=o`i at a cache or the server.
- The split parameter defines which queries are executed at the cache. However, the server re-executes these queries if the delivered result was invalid, or if the server has switched to normal mode where no more delivered results are used. The *re-execution time* is defined as the average execution time of all such re-executions at the server during an execution of S.
- The *throughput* is defined as the number of calls of S during the test (80 seconds) of a combination of *split* and `updateDelay`.
- The *idle time* is defined as the average time the server has to wait during the execution of a procedure. This includes all `WAIT` statements.
- The *reuse rate* is defined as the percentage of valid queries of an execution of S. 100% states that all results of queries in *split* have been reused at the server, and 0% states that none of the results could be reused. The higher the number of consistent objects at the cache, the higher the reuse rate.

Illustration of Plottings

In the subsequent experiments we show how the performance depends on the run time parameters. We use a 3-dimensional diagram as depicted by the right side of Figure 6.1. The X-axis represents the different settings of the *split* parameter, or more precisely, the number $|split|$ of queries that are executed by the cache. The Y-axis represents the settings of the `updateDelay` parameter as defined above. The Z-axis represents the observed performance value. The buttom of the diagram shows the contour lines and the appropriate value of the Z-axis. More precise, denote the maximal value between two contour lines.

The left side of Figure 6.1 depicts the same data, but only the area with the contour line rotated by 90°. To improve the readability, we only use such diagrams in the remainding part of this work where we name the name of the performance value (Z-axis) in the top of the diagram.



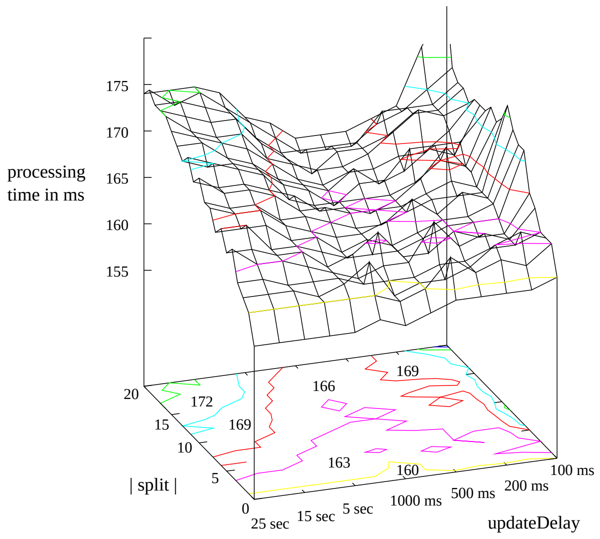


Figure 6.1: Diagrams for run time parameters and observed values.

6.3 Comparison of the Execution Schemes

We perform three types of experiments that show the behavior of (1) the traditional execution scheme and the simultaneous scheme (2) without independent queries and (3) with independent queries. For each type we show the performance behavior for a one- and a four-client system. The configuration of all tested systems is shown by Table 6.2.

Parameter	Values
`objectSize`	100 KB
`tableSize`	5000 records
`updateLoad`	1000 ms
`execLoad`	100 ms
`waitingTime`	5 sec
`triggerPeriod`	80 sec

Table 6.2: Configuration of the tested client-server systems.

The goal of the section is not to investigate different setups where database caching improves the performance. Rather, we start with setups where the traditional caching improves the performance and show that for such situations the simultaneous scheme yields equal or even better performance.

6.3.1 The Traditional Execution Scheme

We measure the performance of the traditional scheme for a one- and a four-client system.

Low Load at Server and One Client

A query of the procedure code is executed at the cache in about 7.5ms and the server in about 7.7ms. The resulting processing time is depicted at Figure 6.2. As we can see, the minimum

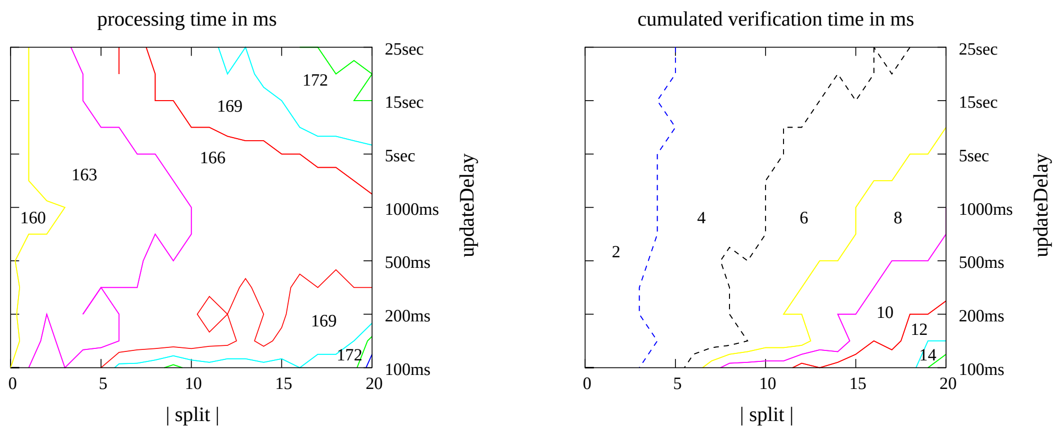


Figure 6.2: ONE-System with one client and traditional execution scheme. Processing time (left) and cumulated verification time (right) in ms.

processing time is about 160ms which represents 20 query executions of 7.7ms each at the server (denoted t_{server}), the network communication between cache and server (0.2ms for both directions, denoted t_{net}) and the overhead of executing the PHP script (about 2-4ms, in average 3ms, denoted t_{php}). Figure 6.3 depicts the resulting execution scheme where the

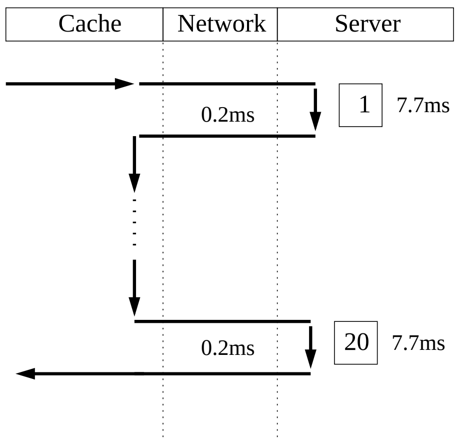


Figure 6.3: ONE-System with one client and traditional execution scheme. Execution scheme for the maximal processing time.

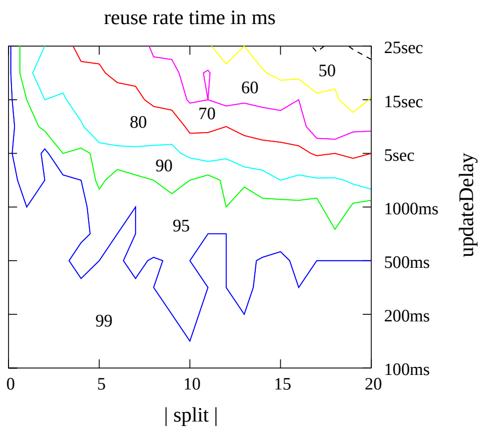
box represent the execution of a query. The processing time is computed by the formula

$$\begin{aligned} t_{total} &\approx t_{cache} + t_{net} + t_{verify} + t_{re} + t_{server} + t_{php} \\ &= 0 + 20 \cdot 0.2\text{ms} + 0 + 0 + 20 \cdot 7.7\text{ms} + 3\text{ms} \\ &= 161\text{ms} \end{aligned}$$

where t_{cache} is the processing time at the cache, t_{verify} the cumulated verification time and t_{re} the re-execution time at the server which in all cases is 0, since all computations take place at the server.

Note that the computation is based on values taken from diagrams, such that the resulting processing time represent an approximate value.

The processing time is maximal in the upper and lower right with about 172ms. Let us first explain the upper one. Consider Figure 6.4 which depicts the reuse rate and the corresponding re-execution time. The higher the number of queries in *split* and the higher



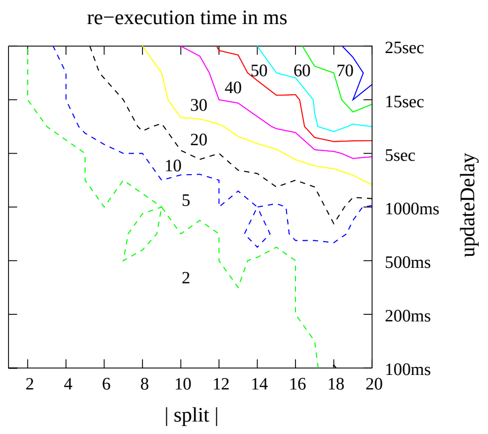


Figure 6.4: ONE-System with one client and traditional execution scheme. Reuse rate in percent (left) and re-execution time (right) in ms.

the value of `updateDelay`, the lower is the reuse rate. Recall that the server switches to normal mode in case of an invalid query, hence, no more query results are then considered by the server which further decreases the reuse rate. The upper maximum can be roughly approximated as follows: The reuse rate is about 55%, hence the cache deliverers about 11 (55% of 20) valid results and the 12-th result is invalid and re-executed by the server that also executes the remaining 9 queries. The 172ms result from $12 \cdot 7.5 = 90$ms (queries at the cache), 4ms for all network communication, about 5ms of verifying the 12 results (see Figure 6.2), a re-execution time of 70ms (see Figure 6.4) and the PHP overhead of 2-4ms. This is,

$$\begin{aligned} t_{total} &\approx t_{cache} + t_{net} + t_{verify} + (t_{re} + t_{server}) + t_{php} \\ &= 12 \cdot 7.5\text{ms} + 20 \cdot 0.2\text{ms} + 5\text{ms} + (70\text{ms}) + 3\text{ms} \\ &= 172\text{ms} \end{aligned}$$

where the 70ms re-execution time corresponds to 11 query executions a 7.7ms at the server.

The maximum of the lower right results from the high verification time (see Figure 6.2). For a reuse rate of 99% all results are verified and almost all of them are valid. The lower the reuse rate, the fewer queries are verified due to the switch to normal mode. The 172ms result from 20 queries at 7.5ms at the cache (due to 99% reuse rate), 4ms network communication for all 20 queries, 12ms of verification, about 3ms of re-execution time, and the PHP overhead of 2-4ms. This is,

$$\begin{aligned} t_{total} &\approx t_{cache} + t_{net} + t_{verify} + +t_{re} + t_{server} + t_{php} \\ &= 20 \cdot 7.5\text{ms} + 20 \cdot 0.2\text{ms} + 12\text{ms} + 3\text{ms} + 0 + 3\text{ms} \\ &= 172\text{ms}. \end{aligned}$$

We observe that database caching has almost no effect for a low-loaded system.

High Load at the Server and Multiple Clients

In the following we show how our implementation of a traditional scheme behaves in case of a high-loaded four-client system. A query at the cache is executed within 5.8ms and at the server in a range of 6-16ms, depending on its load. Note that all clients run the same combination of the *split* and `updateDelay` parameters. We did not evaluate settings with different combinations among clients, since it would not influence our observations. Therefore, we only show the behavior of one of the four caches.

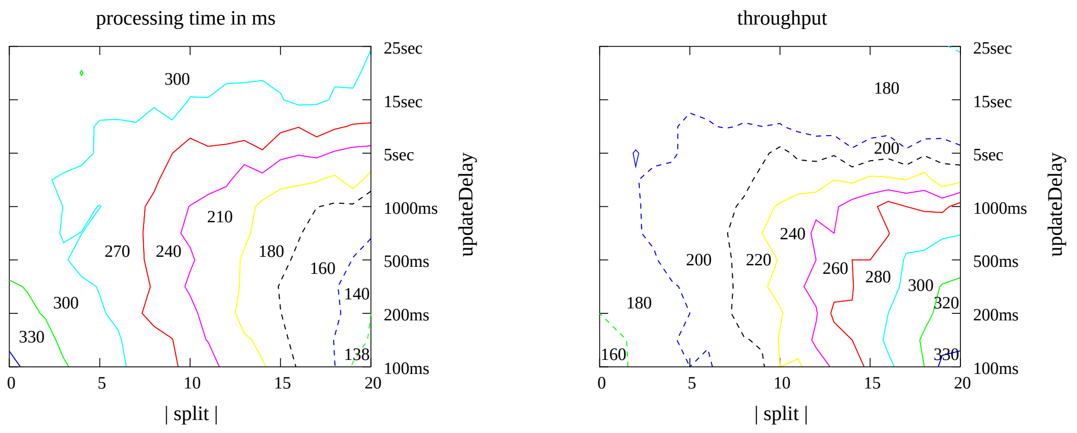
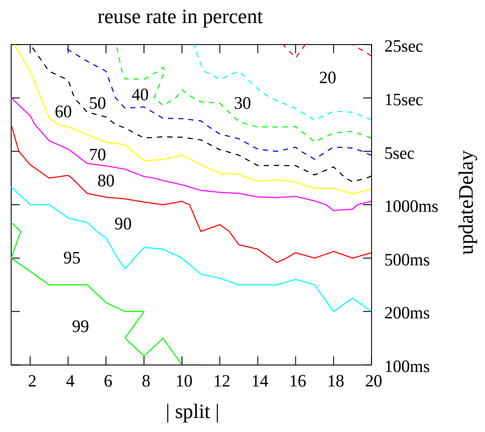


Figure 6.5: ONE-System with four clients and traditional execution scheme. Processing time in ms (left) and throughput (right) of one cache as number of calls.

Figure 6.5 shows the resulting execution time and throughput. Recall that the throughput shows the number of procedure calls that have been triggered at one client in a test phase of 80 seconds. The faster an execution at the cache, the earlier the next procedure call can be triggered. Hence, the maximum throughput is achieved for the lowest processing time. As



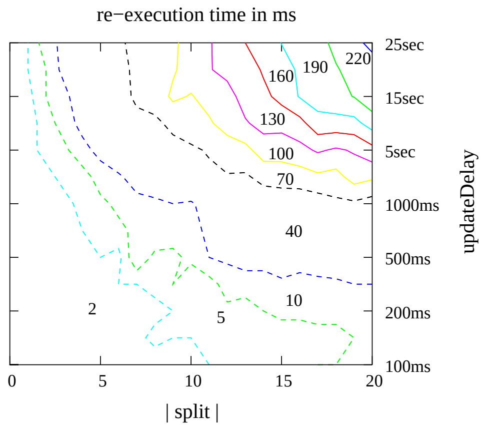
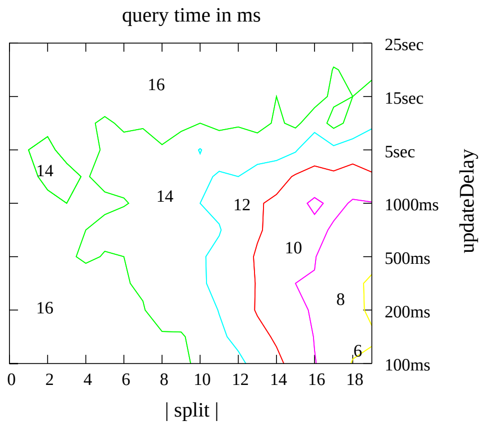


Figure 6.6: ONE-System with four clients and traditional execution scheme. Reuse rate in percent(left) and re-execution time (right) of one cache.

we can see, the processing time is minimal with about 138ms, if all queries are executed at the cache.

Figure 6.6 depicts the reuse rate and the resulting re-execution time. Due to the higher processing time, we observe a lower reuse rate as in the one-client system. The higher the processing time, the more delayed updates can affect the processing and therewith cause more invalid queries. Additionally, the synchronization of four caches further delays the optimistic replication scheme.



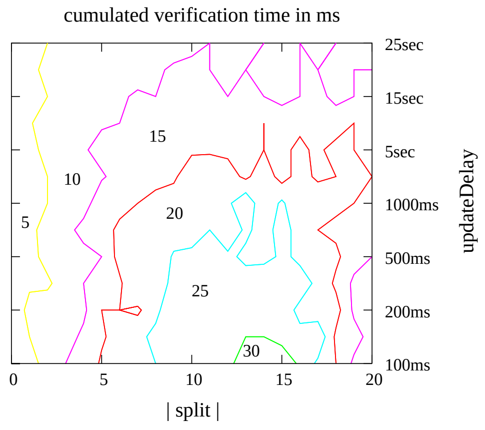


Figure 6.7: ONE-System with four clients and traditional execution scheme. Query time in ms (left) and cumulated verification time (right) of one cache.

Figure 6.7 shows the resulting query and verification time. Since queries are more expensive at the server, each execution of S with a high number of query executions at the cache yields a better processing time. The more queries are executed at the cache, the more load is taken from the server which results in a decreasing query time at the server. Analogously,

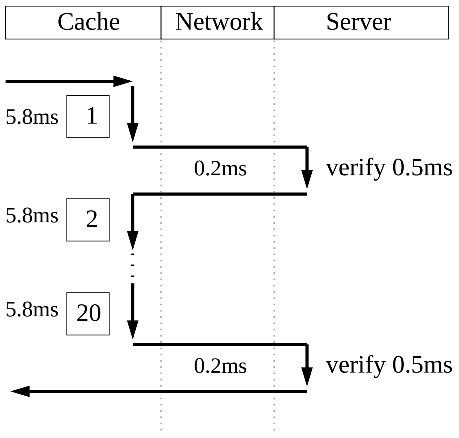


Figure 6.8: ONE-System with four clients and traditional execution scheme. Execution scheme for the maximal processing time.

the verification time decreases for an increasing number of valid queries. However, for a low reuse rate the server still has to perform most of the queries, such that there is almost no improvement for high `updateDelay` values.

The appropriate execution scheme for the minimal processing time of 138ms is depicted by Figure 6.8. This is,

$$\begin{aligned} t_{total} &\approx t_{cache} + t_{net} + t_{verify} + +t_{re} + t_{server} + t_{php} \\ &= 20 \cdot 5.8\text{ms} + 20 \cdot 0.2\text{ms} + 8\text{ms} + 6\text{ms} + 0 + 3\text{ms} \\ &= 137\text{ms}. \end{aligned}$$

The processing time of 330ms in the lower left results from 20 queries at 16ms, network communication of 4ms and the PHP overhead of 2-4ms. This is,

$$\begin{aligned} t_{total} &\approx t_{cache} + t_{net} + t_{verify} + +t_{re} + t_{server} + t_{php} \\ &= 0 + 20 \cdot 0.2\text{ms} + 0\text{ms} + 0\text{ms} + 20 \cdot 16\text{ms} + 3\text{ms} \\ &= 323\text{ms}. \end{aligned}$$

We observe that database caching doubles the throughput for the four-client system.

6.3.2 The Simultaneous Execution Scheme With Dependent Queries

We repeat the experiments from Section 6.3.1 for the simultaneous execution scheme without independent queries. Hence, all queries are executed in the order as they appear in the procedure code. The difference to the traditional scheme is that the server performs queries and the verification in parallel to the computations at the cache, hence reducing the processing time.

Low Load at Server and One Client

Figure 6.9 shows the resulting execution and idle time. Again, a query is executed in 7.5ms at the cache and in 7.7ms at the server. We observe a processing time in a small range of 154-158ms. However, the more queries are executed at the cache, the more the server has to wait for the results. As we can see, the total idle time increases with an increasing number of queries at the cache.

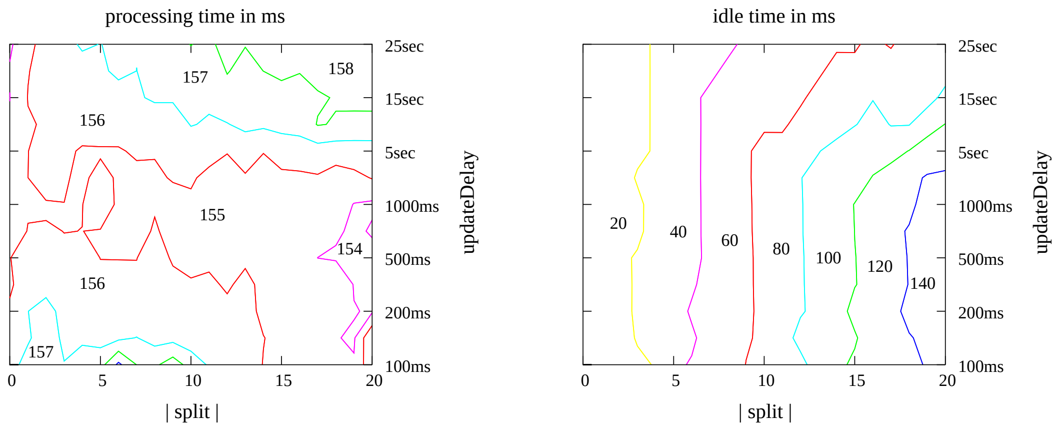


Figure 6.9: ONE-System with one client and simultaneous execution scheme without independent queries. Processing time in ms (left) and idle time (right).

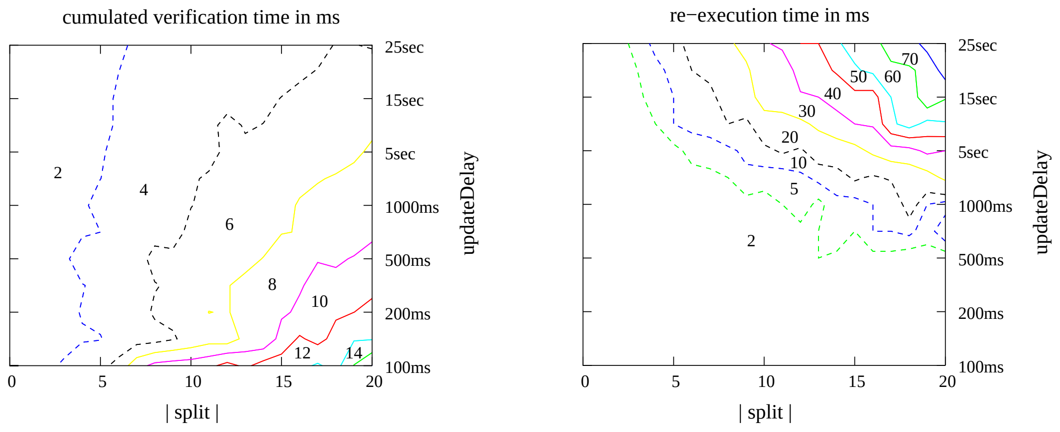


Figure 6.10: ONE-System with one client and simultaneous execution scheme without independent queries. Cumulated verification time in ms (left) and re-execution time (right).

Figure 6.10 shows the verification and re-execution time that are almost equal to the traditional scheme. Recall from Chapter 5 that a cache sends a query result asynchronously to the server, without waiting for a response. Instead the cache continues with the execution of a stored procedure. Therefore, the network communication time, the verification time and the re-execution time does not necessarily affect the total processing time.

Figure 6.11 depicts the minimal processing time (154ms) that has been observed for 18-20 queries at the cache and an `updateDelay` of 100-1000ms. The cache performs 20 queries at

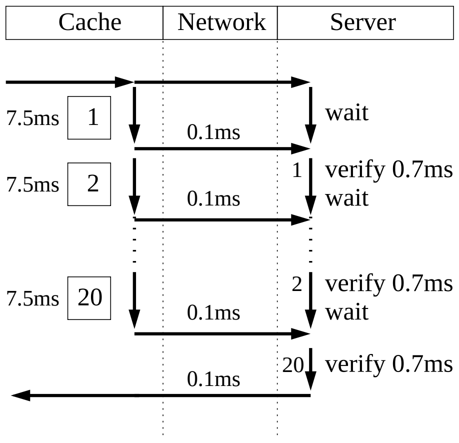


Figure 6.11: ONE-System with one client and simultaneous execution scheme without independent queries. Execution scheme for the minimal processing time.

7.5ms. Sending the last result to the cache costs about 0.1ms. When the last result arrives, the server has already verified all previous results. The verification is about 0.7ms and sending the final result back to the cache is about 0.1ms. The re-execution time is about 1ms. This is,

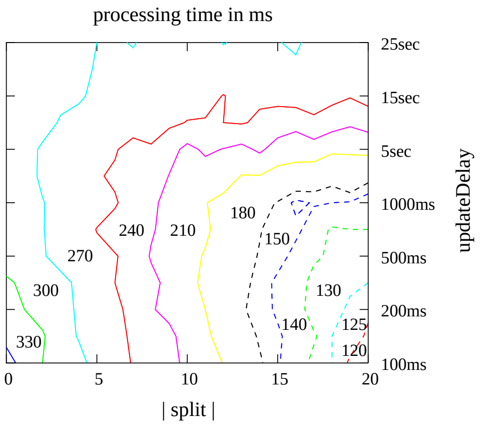
$$\begin{aligned} t_{total} &\approx t_{cache} + t_{net} + t_{verify} + +t_{re} + t_{server} + t_{php} \\ &= 20 \cdot 7.5\text{ms} + 2 \cdot 0.1\text{ms} + 1 \cdot 0.7\text{ms} + 1\text{ms} + 0 + 3\text{ms} \\ &= 154,9\text{ms}. \end{aligned}$$

Note that the time for sending intermediate results as well as their verification (about 14ms in total) does not affect this computation, since these are done in parallel to the computations at the cache.

As a result, we observe that the processing time in the minimum case is almost equal to the traditional scheme. However, the simultaneous scheme also reaches this minimum, even if all queries are executed at the cache.

High Load at the Server and Multiple Clients

For one of the four clients the resulting processing time and throughput is depicted at Figure 6.12. Analogously, the cache performs a query with 5.8ms and the server in a range of 6-16ms. Compared to the traditional scheme the processing time reduces from 138ms to 120ms. By the achieved reduction of the processing time, we observe a throughput of about 360 procedure calls which improves the traditional scheme by 30 calls.



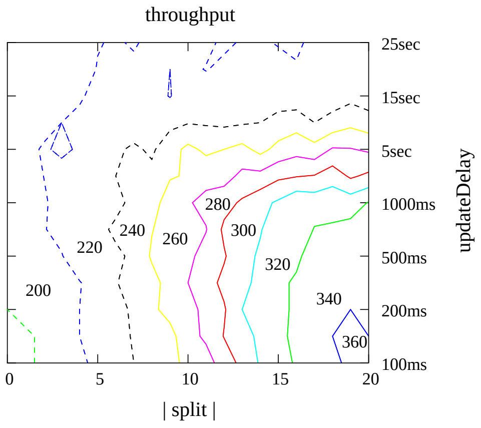


Figure 6.12: ONE-System with four clients and simultaneous execution scheme without independent queries. Processing time in ms (left) and throughput (right) of one cache.

In Figure 6.13 we show the resulting verification, idle, re-execution and query time. For some of the graphics we have skipped the observed values, since they are similar to previous figures or are not related to the maximum case. We observe the same effect on the query

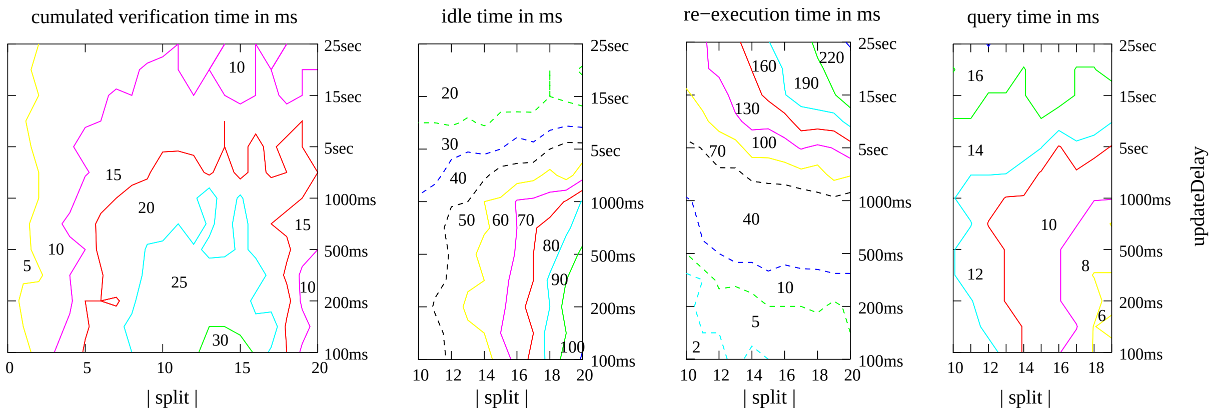


Figure 6.13: ONE-System with four clients and simultaneous execution scheme without independent queries. From left to right: cumulated verification, idle, re-execution and query time.

and verification times as for the traditional scheme. The more load taken from the server, the faster the execution of queries and the verification of results.

Figure 6.14 shows the resulting execution scheme of the minimal processing time. The value of 120ms results from 20 queries at 5.8ms at the cache, 0.2ms of network communication (0.1ms for delivering the last result and 0.1ms for passing back the final result of the server), and 0.5ms for verifying the last delivered result. This is,

$$\begin{aligned} t_{total} &\approx t_{cache} + t_{net} + t_{verify} + +t_{re} + t_{server} + t_{php} \\ &= 20 \cdot 5.8\text{ms} + 2 \cdot 0.1\text{ms} + 1 \cdot 0.5\text{ms} + 2\text{ms} + 0 + 3\text{ms} \\ &= 121,7\text{ms}. \end{aligned}$$

Again, the verification of previous results does not affect the processing time. Taking the

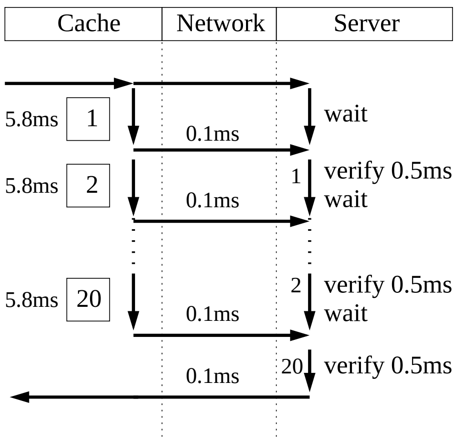


Figure 6.14: ONE-System with four clients and simultaneous execution scheme without independent queries. Execution scheme for the minimal processing time.

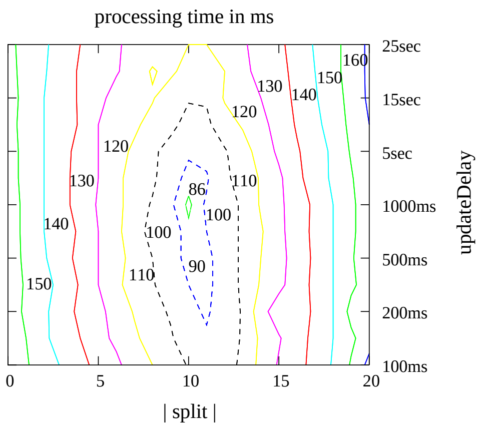
100ms idle time (see Figure 6.13), there are about 16ms of computations left at the server. These are about 10ms for the verification and the PHP overhead of 2-4ms.

6.3.3 The Simultaneous Execution Scheme With Independent Queries

Again, we repeat the experiments from Sections 6.3.1 and 6.3.2, but also allow independent queries. Instead of the sequential split parameters $\emptyset$, $\{1\}, \{1, \dots, i\}$ with $2 \leq i \leq 20$, we use parameters with independent queries $\emptyset$, $\{20\}, \{i, \dots, 20\}$ with $1 \leq i \leq 19$. Hence, the cache can 'jump over' the first queries in S. The X-axis of subsequent figures depicts the number of query executions at the cache. Intuitively, it can be read as follows. Let $i = 0, \dots, 20$ be a value of the X-axis. Then, the server executes the queries 1 to $20 - i$ of the sequence and the cache $20 - i + 1$ to 20. Clearly, when the server has executed its queries, some results of the cache have already been delivered, such that the server does not have to wait for them.

Low Load at Server and One Client

As shown by Figure 6.15, the processing time decreases to a minimum of 85ms for computing 10 queries at the cache and the remaining ones at the server. The speedup is mainly obtained by executing the queries at cache and server in parallel. There are no idle times until the



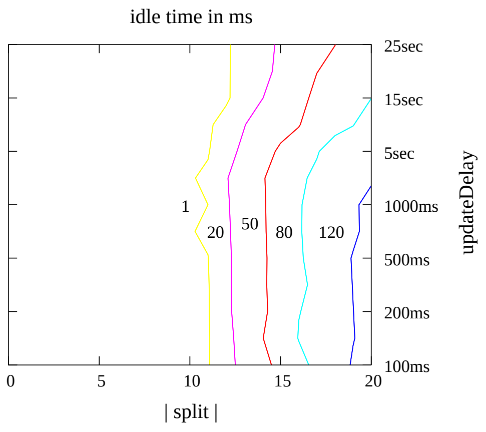
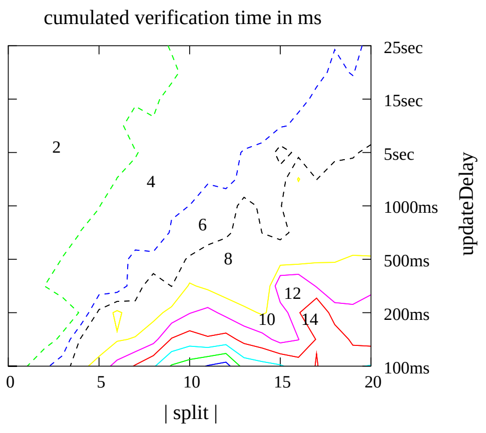


Figure 6.15: ONE-System with one client and simultaneous execution scheme with independent queries. Processing time (left) and idle time (right).

value 10 at the X-axis, since the server executes more queries than the cache. As a result, all results have been delivered, when the server has completed its queries.

As show by Figure 6.16, the verification and re-execution time is again similar to the traditional and simultaneous scheme without independent queries.



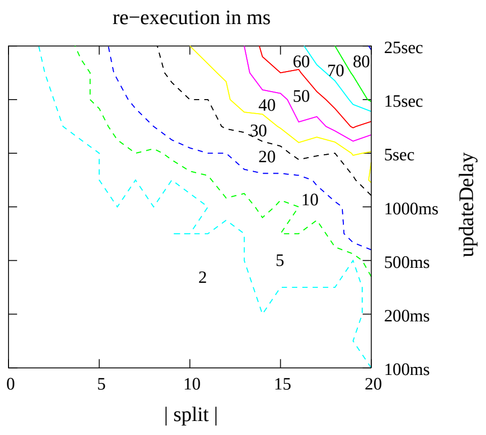


Figure 6.16: ONE-System with one client and simultaneous execution scheme with independent queries. Cumulated verification time (left) and re-execution time (right).

Figure 6.17 shows the execution for the minimum of about 85ms. By taking Figure 6.16 into account, it results from 10 queries at 7.7ms at the server, 0.2ms network communication and 4ms verification of 10 results (0.4ms each). This is,

$$\begin{aligned} t_{total} &\approx t_{cache} + t_{net} + t_{verify} + + t_{re} + t_{server} + t_{php} \\ &= 10 \cdot 7.5\text{ms} + 2 \cdot 0.1\text{ms} + 10 \cdot 0.4\text{ms} + 3\text{ms} + 0 + 3\text{ms} \\ &= 85.2\text{ms}. \end{aligned}$$

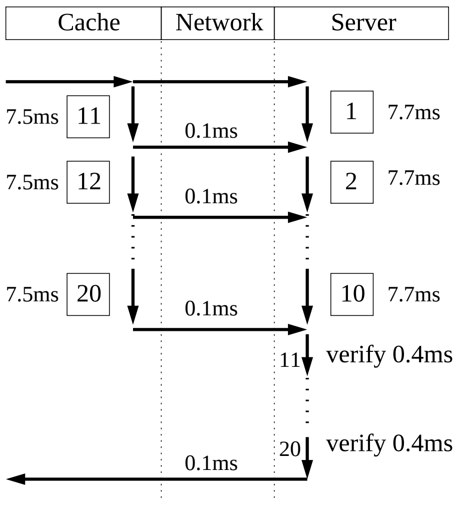


Figure 6.17: ONE-System with one client and simultaneous execution scheme with independent queries. Execution scheme for the minimal processing time.

As expected by the parallel execution of 10 queries, the processing time is about half as much as the processing time of the minimal processing time of the traditional scheme.

High Load at the Server and Multiple Clients

By the results of the one-client experiment, we also except an improvement for the four-client experiment. For one of the four clients the resulting processing time and throughput is depicted at Figure 6.18. Analogously, the cache performs a query with 5.8ms and the server in a range of 6-16ms. The minimum is about 95ms, which improves the minimum of the

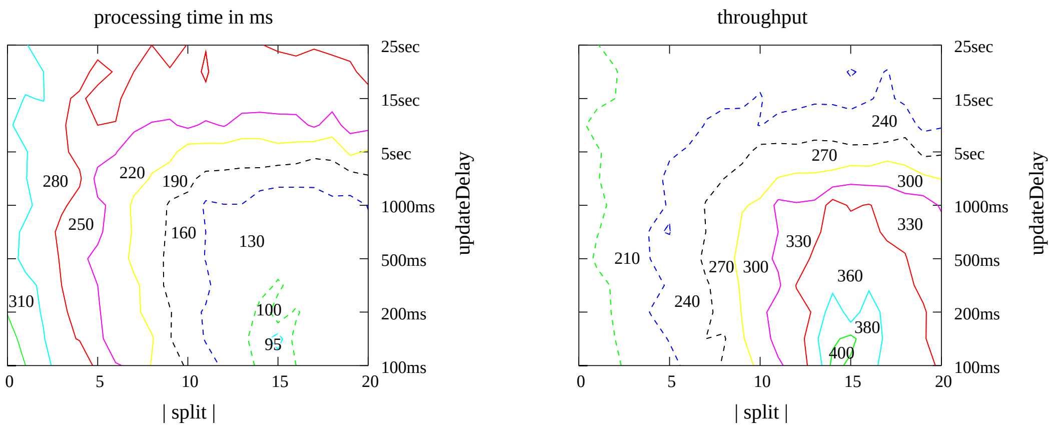


Figure 6.18: ONE-System with four clients and simultaneous execution scheme with independent queries. Processing time (left) and throughput (right).

traditional scheme by about 40ms and the simultaneous scheme without independent queries by 25ms. The throughput could be increased by 70 compared to the traditional execution scheme and increased by 40 compared to our scheme without independent queries.

Figure 6.19 depicts the verification, idle, re-execution and query time which is again

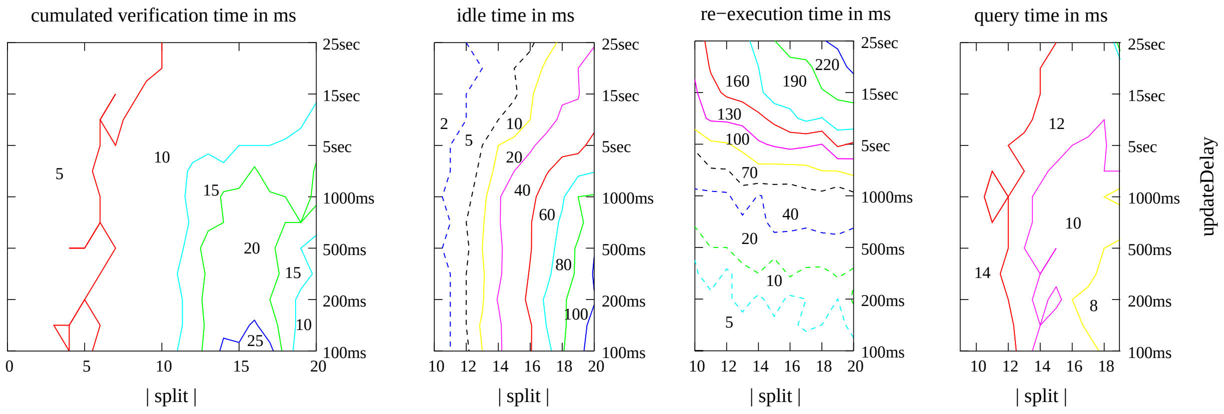


Figure 6.19: ONE-System with four clients and simultaneous execution scheme with independent queries. From left to right: verification, idle, re-execution and query time.

similar to the traditional scheme and the simultaneous scheme without independent queries. Note that due to the parallel execution at cache and server, there are no idle times for less than 10 queries at the cache.

Figure 6.20 depicts the execution scheme for the minimum of 95ms. It results from 15 queries at 5.8ms at the cache, 0.2ms of network communication and 1.5ms of verification for the last delivered result. This is,

$$\begin{aligned} t_{total} &\approx t_{cache} + t_{net} + t_{verify} + +t_{re} + t_{server} + t_{php} \\ &= 15 \cdot 5.8\text{ms} + 2 \cdot 0.1\text{ms} + 1 \cdot 1.5\text{ms} + 3\text{ms} + 0 + 3\text{ms} \\ &= 94.7\text{ms}. \end{aligned}$$

Since the server has to wait for some of the results, it can perform the verification in parallel to the cache, such that these overhead does not affect the processing time.

As a result, we conclude that the best performance results from an optimal balancing of the procedure code that is influenced by the query time, the verification time, the re-execution time, and the existence of independent queries in the procedure code.

6.4 Other Related Experimental Results

In the following we investigate two more experiments. First, we show the effect of a slow database cache. Second, we discuss a modification of the simultaneous execution scheme which continuously uses the shared mode.

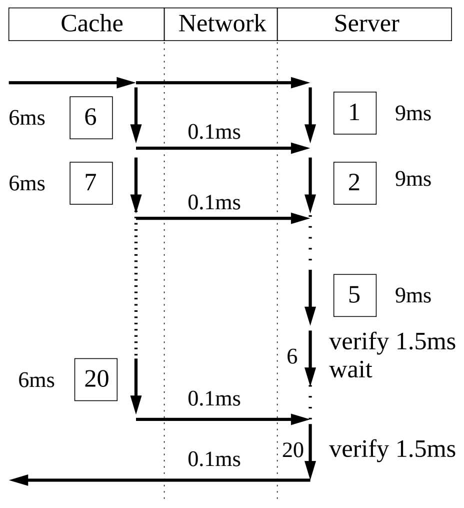


Figure 6.20: ONE-System with four clients and simultaneous execution scheme with independent queries. Execution scheme for the minimal processing time.

6.4.1 Performance of Slow Database Caches

We repeat the four-client experiment with one slow database cache. The following figures refer to the observed performance of procedures that have been triggered at this cache. Again, the server performs its queries in a range of 6-16ms. However, the cache executes a query in 12ms. Figure 6.21 shows the resulting performance with dependent queries and Figure 6.22

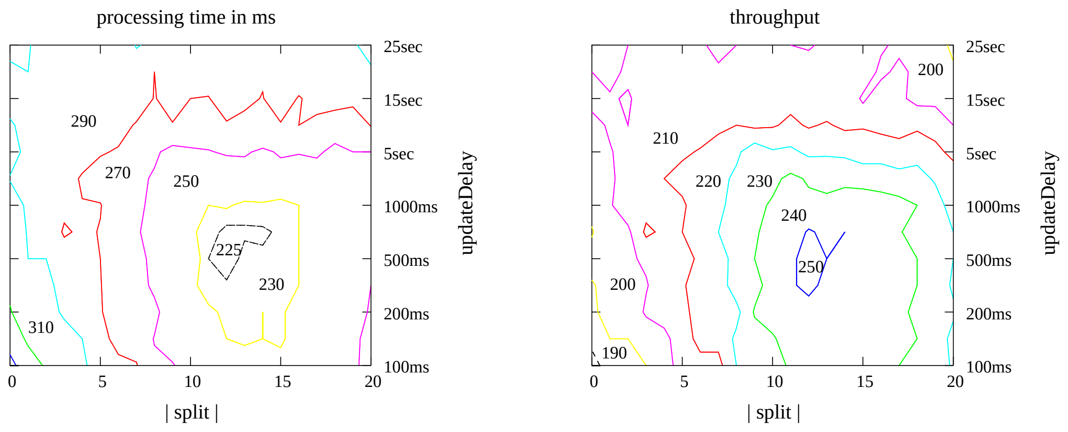
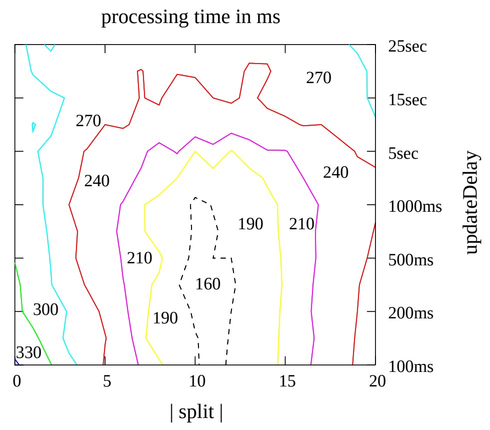


Figure 6.21: ONE-System with four clients and simultaneous execution scheme with dependent queries. Processing time (left) and throughput (right) for a slow cache.

the performance for independent queries.

As expected, the processing time and the throughput decrease and the cache performs less queries as its fast conterparts (see Sections 6.3.2 and 6.3.3). Without independent queries,



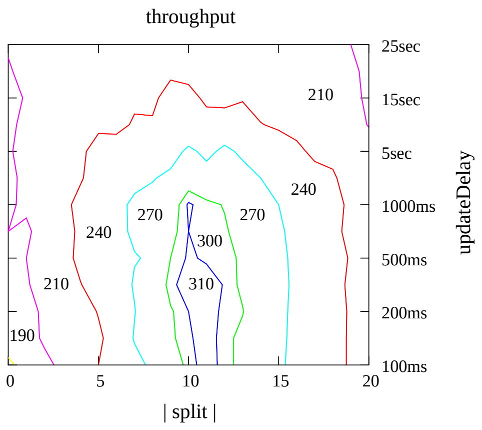


Figure 6.22: ONE-System with four clients and simultaneous execution scheme with independent queries. Processing time (left) and throughput (right) for a slow cache.

the caches performs about 8 queries less and with independent queries, about 5 queries less. Clearly, the slower the cache, the more queries have to be executed at the server, to obtain a minimal processing time.

We conclude that the simultaneous execution scheme also benefits from slow database caches. More generally, a database cache is useful as long as query executions at the cache improve the processing time.

6.4.2 Shared vs. Normal Mode

As mentioned at the beginning of this Chapter, we discuss a modification of our scheme that now always runs in shared mode. Hence, the server does not switch to normal mode and tries to use the delivered results independently of previous invalid results. First, we look at the pure sequential case where no independent queries exists in the procedure code, and second, we discuss the modification for independent queries.

Shared Mode and No Independent Queries

Let us consider the ONE-System and a cache that has to execute the queries $q_1, \ldots, q_n$ sequentially and to deliver the results to the server. If the result of a query q_i with $1 \leq i \leq n$ is invalid, the results of $q_{i+1}, \ldots, q_n$ are also invalid due to the non-existence of independent queries. If the server does not switch to normal mode, it would still wait for these invalid results, verify, and re-execute them. As a result, the server performs additional idle times and verifications that would not have been performed by switching to the normal mode. Hence, the modified scheme would yield a lower performance.

However, the original scheme as defined in Section 5 additionally uses notifications to inform the server that the cache is currently executing a query. Therefore, the server can already perform the verification and execute the query by itself before the cache has even delivered the invalid result. Still, the server performs additional waiting phases (for notifications) and verifications that would not have been performed by switching to the normal mode. Again, the modified scheme would yield a lower performance.

Shared Mode and Independent Queries

The picture is different for independent queries in the procedure code. In the same situation, one or multiple results of the queries $q_{i+1}, \ldots, q_n$ can still be valid, since they do not depend on $q_1, \ldots, q_i$. By switching to normal mode, the server might reject valid results. When the server keeps the shared mode, instead of switching to normal mode, it can possibly use more delivered results, such that there are less re-executions of queries at the server. As a result, the performance can be further improved as underlined by the following experiment.

We run the four-client experiment with independent queries for the modified scheme. The resulting processing time and throughput are depicted at Figure 6.23. Compared to the results of the original scheme (see Figure 6.18), we obtain a similar minimal processing time (about 95ms) and a similar throughput (about 400 procedure calls in 80 seconds). However, the optima within the diagrams cover a larger area. Consider the region of 130ms and 330 procedure calls. Both are twice as big as for the original scheme that switches to normal mode.

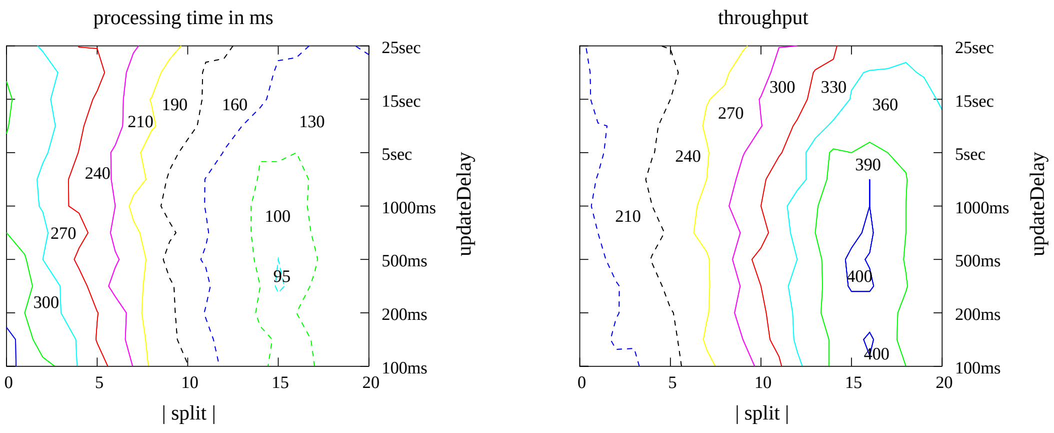
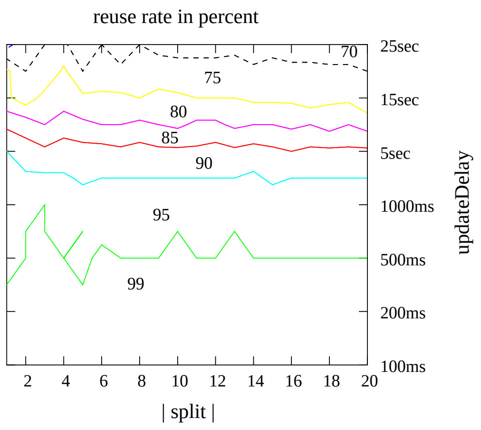


Figure 6.23: ONE-System with four clients and modified simultaneous execution scheme with independent queries. Only shared mode. Processing time (left) and throughput (right).

This effect can be explained as follows. The synchronization process at the server propagates updates in a delayed manner. The delay is specified by the parameter `updateDelay`. A given value of `updateDelay` causes a certain number of inconsistent data objects at the cache that is independent of the used *split* parameter. While performing the modified scheme, this independence of the *split* parameter must be reflected by the reuse rate, since each delivered result is verified by the server.

Figure 6.24 depicts the resulting reuse rate and re-execution time. As expected, the reuse rate that is almost constant for a fixed value of `updateDelay`.



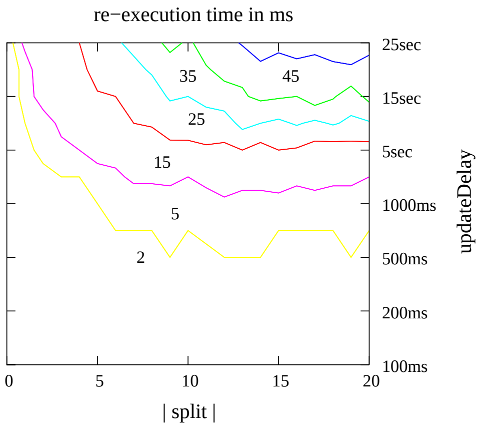


Figure 6.24: ONE-System with four clients and modified simultaneous execution scheme with independent queries. Only shared mode. Reuse rate (left) and re-execution time (right).

The higher reuse rate is causing a larger optimum within the diagrams. Consider again the processing time of 130ms. In Figure 6.24 it corresponds to a reuse rate of about 70%. If we look at the processing time of the original scheme at Figure 6.18 and its reuse rate at Figure 6.6, we realize that the 130ms of the original scheme also correspond to a reuse rate of about 70%. The increasing reuse rate also decreases the re-execution time as shown by the Figures 6.24 and 6.19.

From the experiments with the modified scheme, we conclude that depending on the amount of independent queries in the procedure code, the performance either gets worse (in case of dependent queries) or improves (in case of independent queries). However, we consider the proper handling of shared and normal mode as fine-tuning of our approach which would also require a precise detection of independent queries. Since we show only the existence of such queries, we push this problem to further extentions of our scheme and consider instead the primarily proposed scheme that switches to normal mode whenever an invalid queries occurs.

Shared Mode and the Traditional Protocol

For this experimental analysis we have modified the traditional protocol to switch to normal mode in case of invalid queries. The modification was necessary to make both protocols compatible. However, as shown by the experiments in this chapter, the improvement of the performance results mainly from parallel performed verifications, less network communication and independent queries. Hence, our scheme outperforms the original one if the shared mode is used.

6.5 Summary and Discussion

To analyse the performance of the simultaneous execution scheme, we have defined the ONE-System that executes a single stored procedure related to one table. As we have shown, our implementation of the simultaneous scheme outperforms the detection-based traditional scheme. Figure 6.25 summarizes the comparison of the traditional and our novel scheme. It shows the throughput and processing time for one of the four clients. For our specific

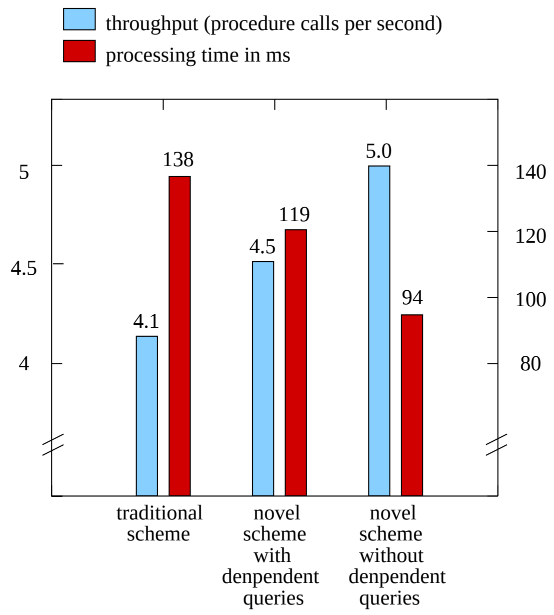


Figure 6.25: Throughput and processing time for the ONE-System with four clients and three different execution schemes.

example the throughput could be improved by about 10% with dependent queries and by about 22% by taking independent queries into account. The increasing throughput results from a faster processing time of procedures that allows the computation of more procedure calls per time unit. The processing time could be reduced by (1) performing the verification in parallel, (2) by reducing network communication, and (3) by taking dependent queries within the procedure code into account.

Further, we have shown that independent queries and the correct handling of the shared mode further improves the performance. For all experiments we could show that small update delays (range of 100ms to 1000ms of the parameter `updateDelay`) have only a minor impact on the maximal performance. Hence, the use of an optimistic synchronization protocol for propagating updates is feasible.

However, the experiments have been performed in a narrow scope and a complete comparison of both schemes requires further:

- to embed and test the scheme into the execution engine of a real DBMS, e.g., an open-source DBMS, such as Postgres,

- to study the performance for a high number of connected caches and determine scalability limits,
- to analyse experimentally the best performing switching rules for the shared and normal mode, and
- to implement and test the full scheme that includes the concept of notifications.

Open Questions

We have used the `updateDelay` parameter for simulating the amount of data inconsistencies at the cache. In all experiments the server performs one update per second (the parameter `updateLoad` is always set to 1000ms), such that we expect a high performance for all settings of `updateDelay` that are smaller than 1000ms or slightly above that value. This has been confirmed by all experiments.

Intuitively, we expect the best performance for `updateDelay`=100ms, since then the probability of data inconsistencies is very low at the cache. However, for some of the experiments the maximal performance is located above that value (e.g. Figures 6.15 and 6.23). As we can see from the reuse rate of these experiments, the number of data inconsistencies is higher for these maxima than for the setting of `updateDelay`=100ms. This means that the number of data inconsistencies is not minimal for the best performance, allthough a higher number of inconsistencies causes more overhead for verification and re-execution.

Finally, we could not clarify whether this phenomenon results from hidden overhead of our synchronization process or from measurement errors. We ignore this problem in the following, since these displacements of extreme values do not influence the major result of the experiments.

Part III

Self-Adaptive Load Balancing between Cache and Server

Chapter 7

Dynamics, Performance and the Optimization

Performance is key issue for client-server database systems. Beside hardware equipment, user activities and amount of data, etc., it also depends on the optimal configuration of the database management system itself.

In Part II we have defined a client-server database system with a database cache at each client and a simultaneous execution scheme that allows us to split the procedure code into two parts and to execute both simultaneously at cache and server. At a cache, the scheme is controlled by two configuration parameters: (1) the size of data that is replicated at a cache and (2) the split of the procedure code. In various experiments we have shown the impact of these parameters on the systems performance.

As mentioned in Part I, one of the major requirements of a database cache is its capability to autonomously adapt the caching to load conditions and user activities changing over time. Therefore, a cache runs an optimizer that is responsible for maximizing cache performance by continously adjusting the configuration parameters during run time.

In this Section, we define the optimization problem which the optimizer has to solve. We look at the dynamics of a client-server database system and show that various influencing factors cause a time-varying optimum. Furthermore, we define a set of performance measures that are used to define the objective function and additional important restrictions.

7.1 The Dynamics of Client-Server Database Systems

First, we motivate the optimization problem by looking at the dynamics of a client-server database system caused by various time-dependent factors. From the view of a single cache, we observe the following factors:

- **Requests**
  At the client site, users and application programs initiate procedure calls that are simultaneously executed by the cache and the central server. The frequency and type of procedure calls have a significant impact on the system's performance. In general,

the work load profile of such systems is not predictable. There are cyclic access patterns over time that represent the daily work load. The number of users is not constant. The behavior of individual users can differ. Users change their access behavior over time, e.g., they complete projects and start new ones, etc. Furthermore, procedures might be changed, added or removed by administrators.

- **Resources**

  If a cache is running on an autonomous machine, it normally has full access to the computational and storage resources. However, a cache at a client or application server does not have full access. Concurrent applications can further occupy computation, storage and network resources that cannot be utilized by the cache. These applications again depend on user activities and their precise impact on cache performance cannot be predicted in advance.

  Another important resource is the central server. From a cache point of view, it is a computational resource that accepts query results, computes requests and notifies the cache about the final result of a procedure call. Alltogether, the performance of the server depends on (1) triggered procedure calls from all caches and clients, and (2) the amount of code that is assigned by the settings of split parameters.

  Resources are also subject to change. Over the life time of a client-server system, hardware is replaced or upgraded, for example disk space at clients and application servers, main memory, etc. However, such a change can even not necessarily detected by a cache, since a faster performing server can either result from a lower load or a hardware upgrade.

Hence, the frequency and type of requests, as well as the availability and amount of resources for its execution vary over time.

Time-Varying Optimum

The time-dependent influencing factors may cause a non-static optimum of the performance. This has been confirmed by the experimental analysis of Chapter 6 where we have investigated the performance of the ONE-System for different setups. In Figure 7.1 we present again some of the experimental results. The X-axis represents the split parameter, i.e. the amount of queries executed at the cache. The Y-axis represents the synchronization frequency of cache data. The lower the frequency, the higher the error rate and number of re-executions. The left diagram represents a system with a low-loaded server where a procedure call achieves a minimal processing time of 85ms for 10 queries. The diagram at the middle shows the same situation for a high-loaded server which results into 95ms for 15 queries. A slow database cache is depicted at the right which achieves 225ms for 12 and 13 queries. Note that a slow cache may either result from poor hardware equipment or from other applications that utilize the client and thus the database cache.

From these observations we conclude that the optimizer of a database cache must revise decisions. That is to say, it has to cope with a moving optimum at run time.

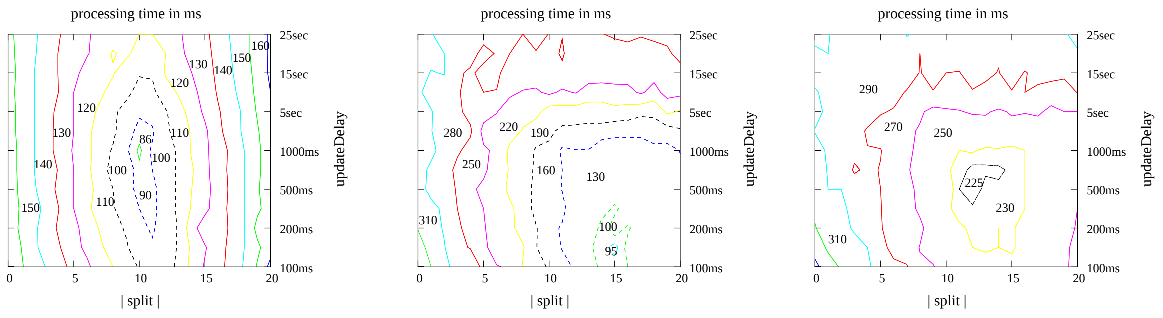


Figure 7.1: Minimal processing time of the ONE-System for different load conditions and types of requests (left: low-loaded server, middle: high-loaded server, right:slow database cache).

Capture Run Time Behavior of a Cache

When we have to deal with a time-varying problem, we have to chose an appropriate model for capturing time. We observe the behavior of a cache at time points

$$t_1, \ldots, t_i, t_{i+1}, \ldots$$

selected with equal length $|t_i - t_{i-1}|$, $i > 0$. Each t_i is a point in time and the i-th stage is defined from t_{i-1} to t_i. In the following we use t_i to denote the i-th stage and t to denote an arbitrary stage. If we consider the 24-hour cycle of most client-server database applications, the length of a stage should be in a range of 1-10 minutes. For shorter sampling intervals, the stage management can possibly cause a bottle neck and longer stages provide only a few possibilities to change the systems configuration. For example, a length of 1 hour allows 24 changes of the system's configuration. In our experiments we have chosen a sampling interval of 80 seconds.

A database cache as a time-varying system can be described by a model

$$y_t \quad = \quad f(x_t, \theta_t, t) + \phi_t$$

where

- x_t is a vector of configuration parameters, e.g., split and replication parameters,
- θ_t is a time-varying "environmental" parameter vector, e.g., user behavior, availability of resources, etc.,
- ϕ_t is the noise and
- y_t is a system behavior vector that characterizes the run time behavior during stage t, e.g., response time of procedure, throughout, etc.

However, it is almost impossible to develop a static model that characterizes the function f, since we neither are able to capture all influencing factors in θ_t, nor its variability over time. Thus we view the behavior of a cache as a black box system with an input and an output. Then, the behavior of a specific cache C is represented by a function

$$y_t \quad = \quad f_C(x_t, t)$$

where x_t is the input, y_t is the output and t is a stage. Again, x_t denotes the vector of configuration parameters and y_t the vector of values that reflect the behavior of C in t. Hence the function value of $f_C(x_t, t)$ represents a measurement of the cache behavior in t.

Therefore, we define in the Sections 7.2 and 7.3 the observed cache behavior in y by a set of measures. All of these are denoted in the form

$$mLabel$$

where the letter m indicates a measure and $Label$ its name. To underline the membership of a measure $mLabel$ to a function value $y = f_C(x, t)$, we either write $y.mLabel$ or $mLabel_t$.

Objective Function to be Minimized

Given a stage t with $y_t = f_C(x_t, t)$, we define in Section 7.4 the objective function

$$g(x_t, y_t)$$

that values the configuration x against the behavior of the cache in stage t. The performance in stage t is optimal iff $g(x_t, y_t)$ is minimal.

The optimizer of a cache observes the behavior $y_{t_1}, \dots, y_{t_i}$ and sets the configuration $x_{t_{i+1}}$ based on these observations, i.e.

$$x_{t_{i+1}} \quad = \quad x_{t_{i+1}}(y_{t_1}, y_{t_2}, \dots, y_{t_i}).$$

Consequently, it has to apply an appropriate on-line (or real-time) optimization technique that continuously analyses the observed behavior and tries to identify an optimal configuration for t_{i+1}.

The underlying assumption is that the behavior during a stage is almost constant and that the behavior of the stage t_{i+1} is mostly "similar" to previous stages. Otherwise the observations of previous stages could be applied to t_{i+1}. Under this Markov assumption the configuration function reduces to

$$x_{t_{i+1}} \quad = \quad x_{t_{i+1}}(y_{t_i}).$$

Continuous Optimization

Continuous optimization problems have been well studied in literature and have their origins in on-line or real-time optimization of industrial processes, e.g., chemical processes which depend on time-varying process conditions. Thus, the goal is not longer to find the precise optima, but to track their movement through the space as closely as possible. In Section 7.5 we give an overview on the main research directions. For a cache the appropriate continuous optimization problem is defined in Section 7.4.

7.2 Cache Execution Performance

In this Section we define the execution-performance of a cache C which captures the efficiency of executing procedure code by the simultaneous execution scheme. In Section 7.3 we define the fragment-performance which captures the impact of replicated fragments on the performance of a cache, hence defining fragments that are suitably for replication.

As shown by the experiments in Chapter 6, the processing time of a procedure is a feasible measure of the system's performance. We shall define two alternative measures for the execution-performance that aim at a maximal cache utilization and a maximal balance of code execution among cache and server respectively.

7.2.1 Idle and Execution Time of IO statements

As defined in Part II, we capture the execution of a procedure by execution sequences (see also see Definition 4.2). We define an extended version of execution sequences that are used to capture the execution time and resulting idle times of IO statements as well as their mode of execution. That is,

$$seq \quad = \quad (s_1, \#s_1, e_1, mode_1), \ldots, (s_n, \#s_n, e_n, mode_n)$$

with

- $s_i \in ID(S)$ as the statement identifier,
- e_i the statement expression,
- $\#s_i$ as the number of IO statements s_j with $1 \leq i \leq n$ and $j \leq i$, $s_i = s_j$ and
- $mode_i$ as the mode of execution that can take the values $undef$, $exec$, $invalid$ and $valid$.

Note that during a simultaneous execution, the executions at cache and server normally result in different sequences. We explain the mode for both cases.

Consider a procedure S, the parameter $split \subseteq ID(S)$ that defines its simultaneous execution and an IO statement s_i of the above sequence seq. The possible modes at the server are:

- **Mode** *exec***:**
  The mode of s_i is $exec$ if s_i is a low-level update, or if s_i is a query that is not indented to be executed at the cache, e.g., $s_i \notin split$.

- **Mode** *invalid***:**
  The mode of a query $s_i \in split$ is $invalid$ if the function $eval_S(s_i, \#s_i, e_i)$ did execute e_i on server data and hence did not receive or reuse the result delivered by the cache. Recall from Chapter 5 that the server can switch to normal mode where also no more query results are considered. Thus, all following results are also treated as invalid.

- **Mode** *valid***:**
  Similar to *invalid*, but with a correct query result that the server could use instead of computing it.

The possible modes at the cache are:

- **Mode** *exec***:**
  The mode is *exec* if s_i is a low-level update that has been successfully executed on cache data. Note that all low-level updates are rejected at a cache during the final commit. Cache data is exclusively updated by the synchronization process. However, the cache still has to perform these intermediate updates to guarantee a consistent execution of queries in the procedure code.

- **Mode** *invalid***:**
  The mode of a query $s_i \in split$ is *invalid* if the function $eval_C(s_i, \#s_i, e_i)$ has delivered a notification and a result to the server, but the server did not reuse the result. Note that due to stale data, a cache can follow a different path in the procedure code than the server. Hence, for a function call $eval_C(s_i, \#s_i, e_i)$, there is not necessarily a corresponding $eval_S(s', \#s', e')$ with $s' = s_i$ and $\#s = \#s_i$.

- **Mode** *valid***:**
  Similar to *invalid*, but with a correct query result that the server could use instead of computing it.

- **Mode** *undef***:**
  For all remaining cases the mode of s_i is $undef$. This mainly includes cases where the $eval_C(.)$ function at the cache returns $undef$.

In the following we only refer to the cases *invalid* and *valid*. Note that for both cases the corresponding query has been executed on cache and/or server data, and hence has occupied computational resources.

For a query s with a resulting execution mode of *invalid* or *valid*, we define its execution and idle time as follows.

1. The execution time of s is denoted $pt_C(s)$ at the cache and $pt_S(s)$ at the server. Both are defined by the execution time of the system function $eval()$, which is used by the underlying execution engines at the server and caches to execute s, pore precise: $eval_C()$ the cache and $eval_S()$ by the server, see also Sections 5.3 and 5.4.

2. The idle time of s is denoted $idle(s)$. At the cache the idle time is always 0, as a cache during execution never waits for the server. At the server the idle time summarizes the waiting time for an entry in the query result cache (QRC) and for the result to appear in the QRC. This concerns the Steps 1a and 1fi in the function $eval_S(.)$ in Section 5.4.2.

In the following we assume that for all completed executions both execution sequences and the corresponding execution and idle times of their statements are available.

7.2.2 Minimal Processing Time

In Chapter 6 we have shown that a minimal processing time of a procedure indicates a performance with minimal response time and maximal throughput. The first measure for the cache-execution performance aims at minimizing the total processing time of stored procedures.

The processing time $PT(SC)$ of a procedure call SC is measured at the database cache and defined as the period between its initial call and the arrival of the final result that has been sent to the server. Let t be a stage, S a procedure and $SC_1^{(t)}, \ldots, SC_n^{(t)}$ all calls of S that have been triggered at the cache during t, where $n \in \mathbf{N}$ represents the number of procedure calls. The first measure for the execution-performance is based on the average processing time of a procedure S in t. It is defined by

$$mPT_t(S) \quad = \quad \frac{1}{n} \cdot \sum_{i=1}^{n} PT(SC_i^{(t)}) \tag{7.1}$$

The goal is to minimize $mPT_t(S)$ for all procedures S.

7.2.3 Maximal Cache Utilization

The second measure aims at a maximal cache utilization. That is, during the simultaneous execution of a procedure, the idle time at the cache should be minimal. We first define the idle time, then the corrected idle time that does not take invalid queries into account and finally, give an alternative definition for the execution-performance.

Consider a simultaneous execution of a procedure call SC and its termination at a cache. The execution is either interrupted by the server or puts a notification into the QRC to indicate its termination. In the first case there will be no idle time at the cache and in the second the cache is idle until the server sends its completion message. Then,

$$idleC(SC) \quad \geq \quad 0 \tag{7.2}$$

represents the idle time at the cache during the execution of the procedure call. More precise: $idle_C(.)$ represents the time between the initiation of a procedure call at the cache until the server passes back its final result.

The goal is to maximize the cache utilization, and hence to minimize the idle time at the cache. However, a low idle time can be easily achieved by setting $split = ID(S)$. Thus, all the procedure code is executed at the cache independently of valid or invalid query results.

Therefore we correct the idle time by putting a penalty for each executed query with mode *invalid*. Let SC be a procedure call of type S and

$$seq \quad = \quad (s_1, \#s_1, e_1, mode_1), \ldots, (s_n, \#s_n, e_n, mode_n)$$

the resulting sequence at the cache. Furthermore, let $Q_{invalid}$ be all pairs $(s_i, \#s_i)$ with $mode_i = invalid$. We use pairs, since due to loops an IO statement can occur multiple times in seq. A pair $(s_i, \#s_i)$ uniquely identifies an element in seq. The corrected idle time is defined as

$$idleC^{+}(SC) \;=\; idleC(SC) \;+ \sum_{(s_i,\#s_i)\in Q_{invalid}} pt_C(s_i) \tag{7.3}$$

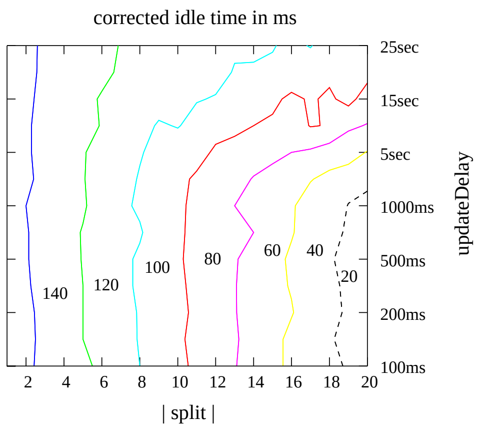
which adds the execution time of all queries that did not produce a valid result. Hence, it does treat these queries as unexecuted at the cache. A value of $idleC^{+}(SC) \approx 0$ indicates that the cache during the entire execution of the procedure call is either executing low-level updates or queries that produce a valid result. However, a value of $idleC^{+}(SC) = 0$ is not possible if the cache ends its execution with a query. Then, the result has to be delivered, verified and the server has to commit the transactions. During this period the cache is idle.

Then, the second measure for the execution-performance is defined by

$$mIdleC_t^{+}(S) \;=\; \frac{1}{n} \cdot \sum_{i=1}^{n} idleC^{+}(SC_i^{(t)}) \tag{7.4}$$

for a procedure S, a stage t and the procedure calls $SC_1, \ldots, SC_n$ of type S during t. Again, the goal is to minimize $mIdleC_t^{+}(S)$ for all procedures S.

We compare the minima of $mPT_t(S)$ and $mIdleC_t^{+}(S)$ based on the experiments that have been made in Chapter 6. For dependent queries Figure 7.2 shows the corrected idle



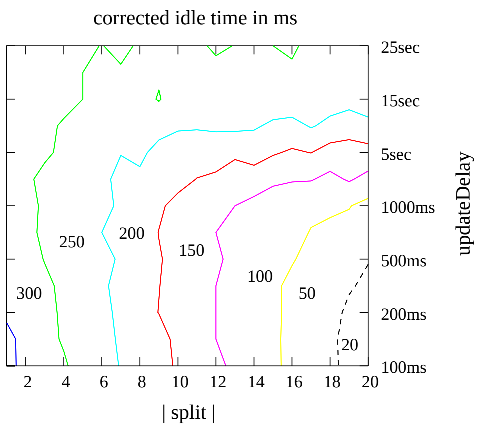


Figure 7.2: Corrected idle time for a one (left) and four client (right) system with dependent queries.

time $mIdleC_t^{+}(S)$. Clearly, the corrected idle time $mIdleC_t^{+}(S)$ aims at a maximal cache utilization, such that its value decreases for an increasing number of valid queries. As we can see, the measure correlates to the processing time $mPT_t(S)$ as shown at Firure 7.3 according to our observations in Section 6.3.2. The minima of both measures are almost identical.

The situation is different if we look at independent queries. Figure 7.4 shows the corresponding corrected idle time $mIdleC_t^{+}(S)$. According the the figure the performance is maximal if a split of size 13 (one client system) or of size 19 (four client system) repsectively

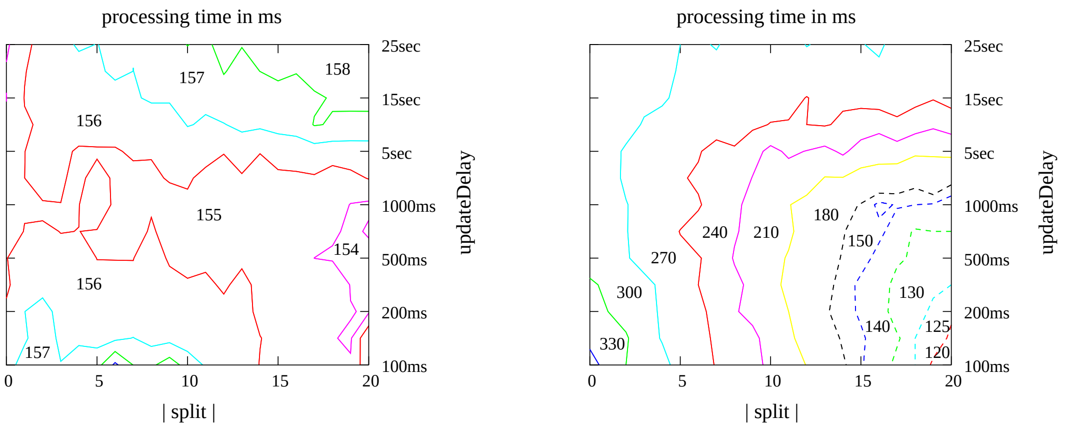


Figure 7.3: Processing time for a one (left) and four client (right) system with dependent queries.

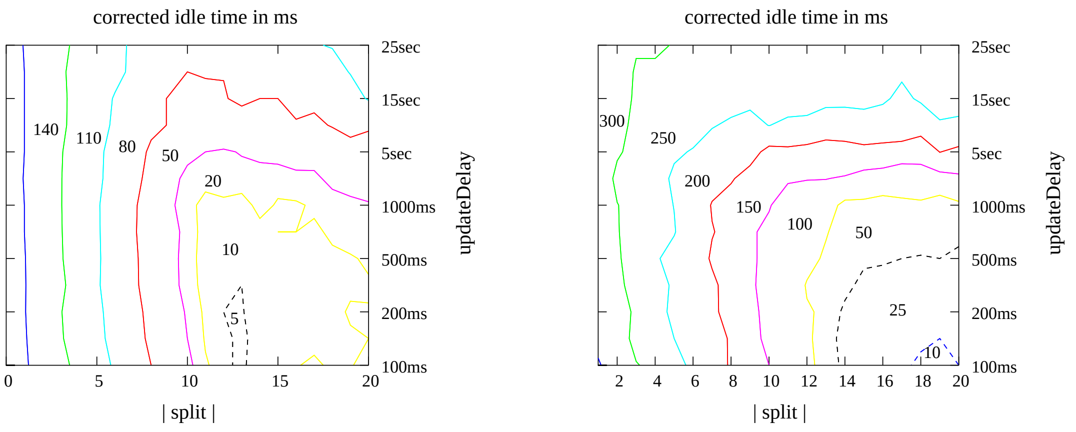


Figure 7.4: Corrected idle time for a one (left) and four client (right) system with independent queries.

is used. As a result, a minimal corrected processing time does not necessarily indicate a minimal processing time for independent queries. The main reason is that the idle time at the server is not considered by the measure as we will see within the next section.

7.2.4 Maximal Code Balancing

The third measure for the execution-performance respects the very nature of the simultaneous execution scheme by balancing the code execution equally among cache and server. The main goal is to minimize the idle time at cache and server during a simultaneous execution.

The total idle time at the server

$$idleS(SC) \geq 0$$

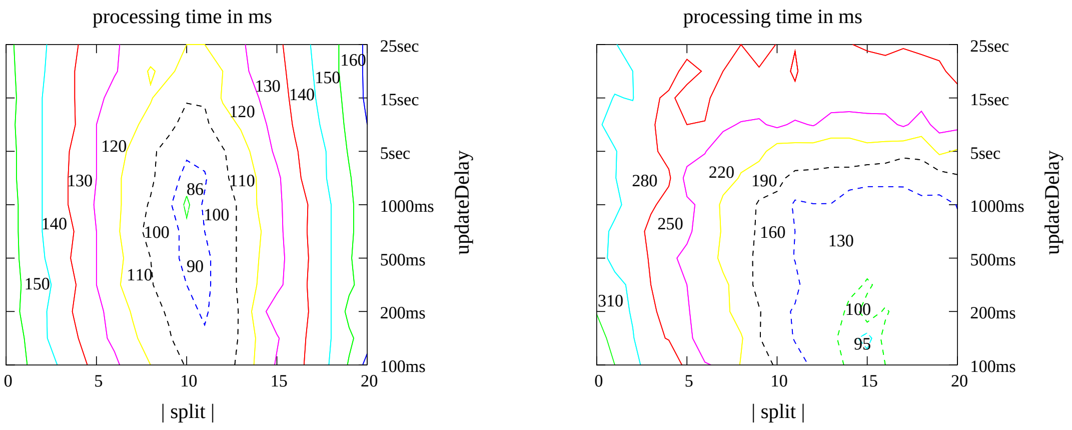


Figure 7.5: Processing time for a one (left) and four client (right) system with independent queries.

for a procedure call SC is defined by the sum of $idle(s)$ for all invalid and valid queries s. Since a high number of re-executions causes almost no idle time at the server, we again put a penalty on invalid queries.

The corrected idle time at the server is defined by

$$idleS^{+}(SC) \quad = \quad idleS(SC) + \sum_{(s_i,\#s_i)\in Q_{invalid}} pt_S(s_i) \tag{7.5}$$

and the code execution balance of a procedure call SC by

$$idleCS(SC) \quad = \quad idleC^{+}(SC) + idleS^{+}(SC) \tag{7.6}$$

The minimum of $idles(SC) \approx 0$ represents the situation where both the cache and the server almost produce no idle times. Hence, all query results at the cache are valid and reused, such that there are no re-executions at the server. Further, both executions are almost of equal length, since otherwise one has to wait for the other. Hence, the code is optimal balanced among cache and server.

The third measure for the execution-performance is defined by the average code balance

$$mIdle_t(S) \quad = \quad \frac{1}{n} \cdot \sum_{i=1}^{n} idleCS(SC_i^{(t)}) \tag{7.7}$$

for a procedure S, a stage t and the procedure calls $SC_1^{(t)}, \ldots, SC_n^{(t)}$ of type S in t. Again, the goal is to minimize $middle_t(S)$ for all procedures S.

Again we have compared the minimal code balancing with the minimal processing time. Figure 7.6 shows the code balancing with dependent queries and Figure 7.7 for independent queries As we observe the code balancing measure does respect more the execution of independent queries, since it overlaps with the minimal processing time (see Figures 7.3 and 7.5). Furthermore, it is very close to the minimal processing time of the sequential case where no independent queries are executed.

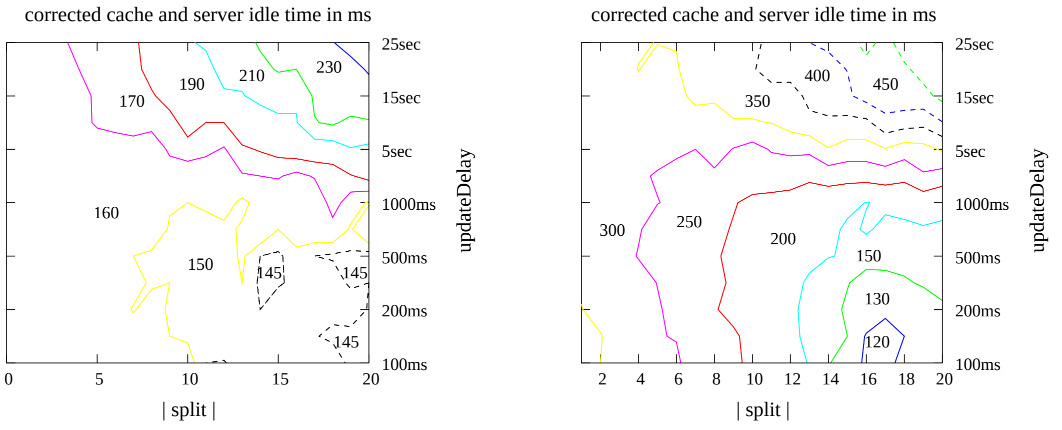


Figure 7.6: Code balance (corrected cache and server idle time) for a one (left) and four client (right) system with dependent queries.

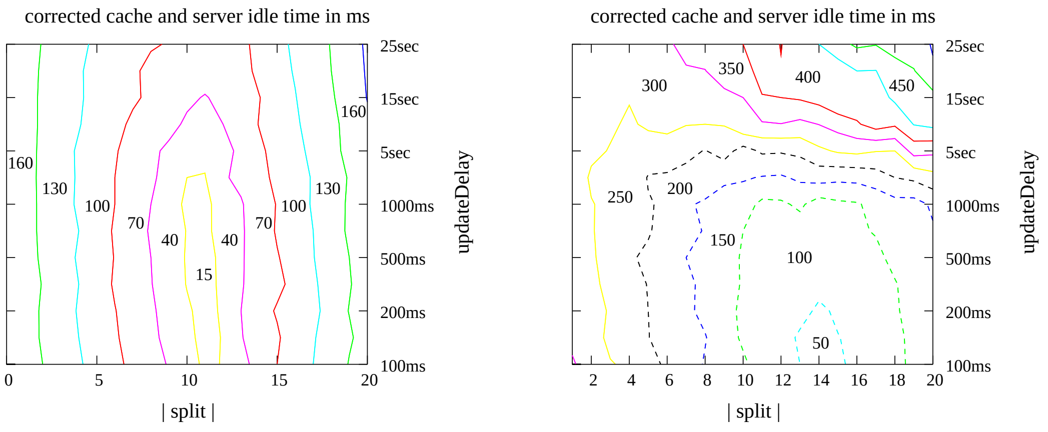


Figure 7.7: Code balance (corrected cache and server idle time) for a one (left) and four client (right) system with independent queries.

7.3 Cache Fragment Performance

The execution-performance alone is not sufficient to capture all performance issues of a cache. To maximize the cache performance, we also have to consider the performance of fragment placements. We introduce two types of measures. The first captures the access frequency of a fragment. It is used to identify fragments for replication, e.g., a fragment that is less frequently written, but frequently read. The second type is used to validate an already replicated fragment, or in other words, to measure its effectiveness. Intuitively, a fragment F should only be replicated if executions on F at the cache yield valid query results that can be reused by the server. For the second, we carefully investigate dependencies of the procedure code, since, for example, a query computation on a fragment F' can access a local variable that was earlier computed on another fragment F. Then, the computation on F' somehow has to be considered by the effectiveness of F. We elaborate this example in Section 7.3.2.

7.3.1 Fragment Access Frequency

We capture the access frequency by two measures — the read and write frequency of a fragment. Consider a cache and all procedure calls at stage t. To process these calls, the $eval_S(s, \#s, e)$ function at the server is called in total n times. Let $t' > 0$ be the length of a stage in seconds and $m \leq n$ be the number of calls that operate on a fragment F. According to the $eval_S(s, \#s, e)$ function as defined in Section 5.4.2, fragment access is detected by $F \in access(e)$. Then, we define

$$mRead_t(F) \quad = \quad \frac{r}{t'} \tag{7.8}$$

as the read frequency of F in t where $r \leq m$ is the number of calls of $eval_S(s, \#s, e)$ on F where e is a query, and

$$mWrite_t(F) \quad = \quad \frac{w}{t'} \tag{7.9}$$

as the write frequency of F in t where $w \leq m$ is the number of calls of $eval_S(s, \#s, e)$ on F where e is a low-level update.

In Section 7.4 we restrict fragment replication by providing upper and lower bounds for both measures.

7.3.2 Dependent Queries in the Procedure Code

Before we define the efficiency of a fragment, we analyse and define dependent queries in the procedure code.

Assume we want to collect the fragment-efficiency as the total execution time of all valid queries that have been executed on a fragment. The following example demonstrates the impact of code dependencies on this collection.

Example 7.1 *Consider the following piece of procedure code. All identifiers starting with* `sp` *are local variables and* `this_month()` *is a function defined on date.*

```
..
SELECT * INTO sp_customer FROM customer WHERE customer_id=sp_id;
IF sp_customer.total_sales>100.000 THEN
   SELECT * INTO sp_benefit FROM extras WHERE month=this_month();
   UPDATE customer_benefit SET benefit=sp_benefit
     WHERE customer_id=sp_id;
END IF;
..
```

The code retrieves some customer data and if the customer has a certain amount of total sales, he gets some benefit, e.g., a gift coupon.

As the example shows, all statements inside the `IF` *statement depend on the first query. If a cache is configured not to execute the first query, the second query can never be executed.*

Let us assume that the first query operates on fragment F_1 and the second on F_2 and that both have been replicated at the cache. Assume further that the execution times of the queries are pt_1 and pt_2 respectively. To respect the code dependencies, we set the fragment-efficiency of F_1 to $pt_1 + pt_2$ and of F_2 to t_2. Thus, we add the execution time of all queries (here the second one) that benefit from another query (here the first one). □

Hence, the proper computation of the fragment-efficiency requires the knowledge of all queries q' in the procedure code that depend on a given query q.

Consider again the run time analysis in Section 5.5. Given a stored procedure S, we have computed a set $Z(S)$ that contains all relevant splits of S. These are all those splits that cause "fewer" $undef$ values and invalid query results at run time. To compute $Z(S)$, we had to execute S on real data for different input parameters. Let SEQ_S be the set of all execution sequences at the server that result from the run time analysis of a procedure S.

First, we define a partial ordering on top of SEQ_S for all queries in the procedure code. The partial ordering defines which query is execution "after" another query. For simplicity we assume that all sequences in SEQ_S are a list $s_1, \ldots, s_n$ of identifiers $s_i \in ID(S)$.

Definition 7.1 (Partial Order on Queries) *Let S be a procedure, SEQ_S the set of all execution sequences and $s, s' \in ID(S)$ two queries. The partial order is defined by $(SEQ_S, <_E)$, where $s <_E s'$ holds if there exists a sequence $s_1, \ldots, s_n$ in SEQ_S with $s_i = s$, $s_j = s'$ and $1 \leq i < j \leq n$.*

The order is partial, since two queries of the procedure code are not necessarily executed sequentially at the server. Consider an `IF` statement with a query in each branch. Then, at run time only one of the queries is executed, such that no ordering exists. Note that due to loops in the procedure code, a query s can be executed multiple times. Hence, $s <_E s$ and $s <_E s' \land s' <_E s$ occur naturally. However, a query s' can only depend on a query s, if s is executed before s', denoted by $s <_E s'$.

The second step for capturing dependent queries is filtering out all independent queries. Given a query s, we want to figure out all queries s' that can be executed independently from s and that are executed after s, hence $s <_E s'$ holds. We take the set $Z(S)$ of "relevant" splits into account.

Definition 7.2 (Independent Queries) *Let S be a procedure, $Z(S)$ all relevant splits of S and $s \in ID(S)$ be a query. The set of independent queries of s is defined by*

$$\begin{aligned} IND(s) \quad = \quad \bigcup \quad & \{s' \mid s' \in split \land split \in Z(S) \land \\ & \qquad s <_E s' \land s \notin split\} \end{aligned}$$

Independent queries of s occur in $split \in Z(S)$ parameters that themselves do not contain s ($s \notin split$). Further, the splits in $Z(S)$ contain queries s' that are executed *after* s ($s <_E s'$). As all splits were executed by the run time analysis, indeed such queries s' exist.

Based on the partial order $(SEQ_S, <_E)$ and the set of independent queries $IND(s)$, we define the set of dependent queries. Given a query s, all queries s' that are executed after s and that are not independent are called dependent queries.

Definition 7.3 (Dependent Queries) *Let S be a procedure and $s \in ID(S)$ be a query. The set of dependent queries of s is defined by*

$$\begin{aligned} DEP(s) \quad = \quad \bigcup \quad & \{s' \mid s, s' \in \mathit{split} \wedge \mathit{split} \in Z(S) \wedge \\ & \qquad s <_E s'\} - IND(s) \end{aligned}$$

Since we define $IND(s)$ and $DEP(s)$ on data from the run time analysis, both are not necessarily precise. Hence, there can be more or less dependent queries. However, for defining a performance measure, the obtained dependencies are sufficient. Note that $IND(s)$ and $DEP(s)$ can be computed at compile time, hence they have only to be computed once.

7.3.3 Fragment Efficiency

As motivated by Example 7.1, we define now the efficiency of a fragment. Again we define it on top of execution sequences as defined in Section 7.2.1. That is, a procedure call SC produces a sequence

$$seq \quad = \quad (s_1, \#s_1, e_1, mode_1), \ldots, (s_n, \#s_n, e_n, mode_n)$$

at the cache. In Section 5.3.2 we have defined the function $eval_C(s_i, \#s_i, e_i)$ that is responsible for executing each s_i $(1 \leq i \leq n)$. For each s_i, the function computes the set of accessed fragments that is denoted by $access(e_i)$. In the following we extend the execution sequence by these sets and write

$$seq \quad = \quad (s_1, \#s_1, frag_1, mode_1), \ldots, (s_n, \#s_n, frag_n, mode_n)$$

instead and ignore the values of e_i, since they are not needed for computing the fragment-efficiency.

The fragment-efficiency of a proceduce call SC is computed by the following algorithm that collects the execution time of all valid queries on a given fragment, as well as the execution time of dependent queries.

Definition 7.4 (Fragment-Efficiency of a Procedure Call) *Let SC be a procedure call, seq the resulting execution sequence at the cache (as above), and F a fragment. The fragment-efficiency result eff(F, SC) is defined as*

```
1. $result = 0$
2. For all $(s_i, \#s_i, frag_i, mode_i) \in seq$
    (a) If $mode_i = valid$ and $F \in frag_i$, then
        i. $result = result + pt_C(s_i)$
```

```
      ii. For all $(s_j, \#s_j, frag_j, mode_j) \in seq$ with $j > i$
          A. If $mode_j = valid$ and $s_j \in DEP(s_i)$,
             then $result = result + pt_C(s_j)$
3. Return result
```

Given a cache in stage t and all procedure calls $SC_1, \ldots, SC_n$ that are performed in t, the fragment-efficency of a fragment F in t is defined by

$$mEff_t(F) \;=\; \sum_{i=1}^{n} eff(F^{(t)}, SC_i^{(t)}) \tag{7.10}$$

The measure respresents the total execution time at a cache for valid queries. The higher its value, the more query executions that affect F have been saved at the server. Note that the measure cannot be computed for fragments that are not replicated. For such fragments we set $mEff_t(F) = undef$.

Recall that a replicated fragment F at a cache is synchronized which causes an additional overhead at the server. Intuitively, the savings in $mEff_t(F)$ should exceed the costs for synchronizing F, since otherwise the server would spend more computation resources into the synchronization than it saves by valid query results on F. In Section 7.4 we provide a lower bound for $mEff_t(F)$ that defines which fragments are useful for replication.

7.4 The Optimization Problem

In this section we give an overview on the model of the client-server system and summarize all underlying assumptions. For this, we split the model in different categories and present each in an own subsection starting with a captial letter **M**. Finally, we present the dynamic optimization problem that defines the optimal performance of a stage.

M1 - The Server

The server is a multi-user system where transactions of different clients are executed concurrently on shared data. We assume that the server has enough storage capacity to host user and application data. Further, we assume that the server is always able to execute the requests imposed by clients, possibly with a low performance. The server might also run other applications, such that the remaining CPU power and main memory varies over time.

M2a - Caches at Clients

The server is connected to a set of $\mathcal{C}$ clients, each hosting a database cache. The number of clients can vary over time. A client is either a single-user system, e.g., a user PC, or a multi-user system, e.g., a web server.

M2b - Capture the Behavior and Performance of a Cache

In previous sections we have shown how to capture the behavior and the performance of a cache along stages $t_1, \ldots, t_i$ by a function f_C

$$y_t = f_C(x_t, t)$$

that returns a vector y_t of the performance measures

execution-performance: $mPT_t(S)$, $mIdleC_t^+(S)$, $mIdle_t(S)$
fragment-performance: $mEff_t(F)$, $mRead_t(F)$, $mWrite_t(F)$
storage space: $mDisk_t$, $mSize_t(F)$ (see below)

At each stage t a given configuration x_t is running and we can exactly measure one value of y_t.

We assume that the value of all basic measures, t.i. $pt_C(s)$, $pt_S(s)$, and $idle(s)$ for a query s, are observed by taking the start and the end time of an action, e.g., an execution of a query by the $eval(.)$ function or an idle time. Hence, the measures do not consider intermediate idle times that might occur if the underlying scheduler of the system assigns the CPU to another process, or if a process is waiting for a data item that is read from the disk.

M2c - Properties of a Cache

A database cache at a client provides a variable disk space $mDisk_t \in \mathbf{N}$ in KB. Note that $mDisk_t$ denotes the disk space of a specific cache and that the space can differ among caches. Note that a client can also host and run other applications. Hence, the available CPU power of a cache can also be variable.

M3 - The Network

Each client is connected by a high-speed network with the server. This is necessary as the intermediate communication of our simultaneous execution scheme requires a fast message exchange. The network can also be used by other applications. Hence, the bandwith and network speed can vary over time.

M4a - Fragments

The server operates on a set $\mathcal{F}_t = \{F_1, \ldots, F_i\}$ of fragments. Each fragment $F \in \mathcal{F}_t$ is of size $mSize_t(F)$ (in KB). Size and number of fragments cannot be predicted and can change over different stages, since they depend on user and application specific data (see Definition 4.4).

M4b - Replication of Fragments

A subset of the fragments in $\mathcal{F}_t$ is replicated to a cache. These fragments are periodically updated by the server. The server applies an optimistic update strategy without concurrency control, such that caches might operate on previous (wrong) versions of fragments. Such errors are detected by the verification step of the simultaneous execution scheme. The set of replicated fragments is set by the optimizer for each stage t.

M5a - Procedures

The server implements a set of $\mathcal{S}_t = \{S_1, \ldots, S_n\}$ stored procedures. A procedure call is defined by a type S and a number of input parameters. The procedure code contains IO statements s (SQL select, insert, delete or update). For a procedure S, each s is uniquely identified by a number. Let $ID(S) \subset \mathbf{N}$ be the set of identifiers for S. The total number of procedures is variable at run time (denoted by the subscript t) since database administrators might remove and add procedures. Procedures are free of side-effects (see Assumption 4.1). Procedures are exclusively invoked by clients.

M5b - Split of Procedures

A split of a procedure S is defined by *split* $\subseteq ID(S)$. All possible splits of S are computed in advance at compile time and denoted by $Z(S) \subseteq 2^{ID(S)}$. The *split* parameters are set at run time by the optimizer at each cache independently. Note that $ID(S)$ and $Z(S)$ are constant unless a stored procedure is not changed by the administrator. If the code of a procedure S is changed at run time, then we assume that S is removed and the changed procedure appears as a new one S' in $\mathcal{S}$. In addition, the sets $Z(S')$ and $DEP(.)$ have to be computed for a new procedure (see Sections 5.5 and 7.3.2).

M6 - Configuration Space of a Cache

Given the set of stored procedures $\mathcal{S}_t = \{S_1, \ldots, S_n\}$ of a stage t, the set of all possible configurations of their split parameters is defined by

$$\mathcal{Z}_t \quad = \quad Z(S_1) \times \cdots \times Z(S_n)$$

The set of replicated fragments at a cache is defined by the parameter $repl \subseteq \mathcal{F}_t$. Thus, the configuration space (search space) of a cache is defined by

$$\mathcal{Z}_t \times 2^{\mathcal{F}_t}$$

Thus, an element of the configuration space fully characterizes (1) the procedure execution at a cache and the server, and (2) the set of replicated fragments. Given the behavior $y_t = f_C(x_t, t)$ of a cache, the configuration vector x_t is a tuple $(z_t, repl_t)$ of the configuration space.

The size of the configuration space is approximated by

$$\begin{array}{ll} & 2^{|\mathcal{F}_t|} \cdot \mid Z(S_1) \mid \ \cdot \ldots \cdot \ \mid Z(S_n) \mid \\ \leq & 2^{|\mathcal{F}_t|} \ \cdot 2^{|ID(S_1)|} \ \cdot \ldots \cdot \ 2^{|ID(S_n)|} \\ \leq & 2^{|\mathcal{F}_t| \cdot |\mathcal{S}_t| \cdot m} \end{array}$$

with $m = max(\mid ID(S_i) \mid)$ over all $S_i \in \mathcal{S}_t$.

M7 - User-Provided Parameters

The goal is to determine a configuration $(z, repl)$ dynamically at run time by the optimizer of a cache. The administrator of the client-server system has only to define basic configuration parameters that serve as general conditions for the optimizer.

For each fragment F, the parameter *minEff* defines the minimal amount of execution time that results from computing valid query results on F at the cache. Hence, each replicated fragment has to fullfill the condition

$$mEff_t(F) \quad \geq \quad minEff \tag{7.11}$$

If the observed value $mEff_t(F)$ is below *minEff*, the fragment does not produce sufficient valid query results, and thus should not further be replicated.

The parameters *minRead* and $maxWrite$ define the read and write frequency of replicated fragments. That is, only fragments with a read frequency of

$$mRead_t(F) \quad \geq \quad minRead \tag{7.12}$$

and a write frequency of

$$mWrite_t(F) \quad \leq \quad maxWrite \tag{7.13}$$

are considered for replication.

Intuitively, good candidates for replication are fragments with a high read and a low write frequency. The setting depends heavily on the system architecture, e.g., network speed and computational power. In our experiments we found out that $maxWrite$ should be at most 1 updates per second and *minRead* at least greater than $maxWrite$. The value of *minEff* can be obtained by an experimental analysis of the synchronization effort of fragments. It should reflect the average total execution time of a stage t at the server to replicate a fragment.

M8 - The Objective Function

We have introduced 3 measures for the execution-performance of a cache. For a procedure S, these are the average processing time $mPT_t(S)$, the average corrected idle time at the cache $mIdleC_t^+(S)$, and the average code balancing $mIdle_t(S)$. In the following we use $mPerf_t(S)$ that represents one of these measures. As shown by the experiments in Chapter 6 and the comparisons in Section 7.2, the performance of a cache is optimal if the value of these measures is minimal.

Given a cache in stage t, its configuration $x_t = (z_t, repl_t)$ and its behavior $y_t = f_C(x_t, t)$, the objective function is defined by

$$g(x, y) \quad = \quad \sum_{S \in \mathcal{S}_t} y.mPerf_t(S) \tag{7.14}$$

which values the used configuration against the resulting behavior of the cache. By applying the following substitutions

$$\begin{aligned} mPerf_t(S, z_t, repl_t) \quad &= \quad y.mPerf_t(S) \\ g(z_t, repl_t) \quad &= \quad g(x_t, y_t) \end{aligned}$$

we get a more convenient and readable form of the objective function

$$g(z_t, repl_t) \quad = \quad \sum_{S \in \mathcal{S}_t} mPerf_t(S, z_t, repl_t) \tag{7.15}$$

Then, the minimum of $g_t(z, repl)$ represents the configuration $(z^*, repl^*)$ with the optimal performance in stage t.

M9 - Optimal Performance of a Stage

Intuitively, the optimization problem is formulated as follows: Find a configuration $(z_{t'}, repl_{t'})$ for a next stage $t' > t$ which minimizes $g(z_{t'}, repl_{t'})$ with $z_{t'} \in \mathcal{Z}_t$, $repl_{t'} \subseteq \mathcal{F}_t$ and all $F \in repl$ comply with access, efficiency and storage restrictions in t.

For a stage t' the optimal configuration $(z^*, repl^*)$ is defined as

$$(z^*, repl^*) \; = \; arg \quad \underset{\substack{z_{t'} \in \mathcal{Z}_t \\ repl_{t'} \subseteq \mathcal{F}_t}}{min} \quad \sum_{S \in \mathcal{S}_t} mPerf_{t'}(S, z_{t'}, repl_{t'})$$

subject to

$$\begin{aligned}
\forall F \in repl: \; mRead_t(F) \; &\geq \; minRead && (7.16)\\
\forall F \in repl: \; mWrite_t(F) \; &\leq \; maxWrite && (7.17)\\
\forall F \in repl: \; mEff_t(F) \; &\geq \; minEff \;\; \vee && (7.18)\\
mEff_t(F) \; &= \; undef &&\\
\sum_{F \in repl} mSize_t(F) + \epsilon \; &\leq \; mDisk_t && (7.19)
\end{aligned}$$

where ϵ is additional disk space that is reserved for replicated fragments that increase their size during a stage. Its value depends on the data scheme and the type of data, e.g., numerical or multi-media data.

M10 - Placement and Removal of Fragments

According to the above optimization problem a fragment is removed from a cache in the following cases:

- It does not fit onto the disk any longer or a new fragment has been replicated, such that due to the disk limit already replicated fragments have to be removed.
- The read and write access on a fragment does not correspond to the user-defined limits. This can be the case if users change their access behavior.
- All queries on a fragment do not produce the user-defined minimal execution time of computing valid query results. This will be the case if the error rate increases, e.g., due to a higher system load or if the split parameters in z have been chosen not to execute queries on that fragment any more.

A fragment is placed at a cache if its access frequency is above/below the user-defined access limits and if it fits onto the disk of the cache.

However, after a fragment has been placed for the first time in stage t, it can already be removed in the next stage if its fragment-efficiency is below the limit *minEff*. To avoid frequent placements and removals of fragments, we can modify the constraints 7.16, 7.17 and 7.18 to consider more than one previous stage. We provide an example for constraint 7.18. The others are analogously.

Let the above stage t' be the i-th stage, say t_i. Then a fragment is only removed if its fragment efficiency is below the given limit for the m last stages.

$$\begin{aligned} \forall F \in repl: \sum_{j=1}^{m} \frac{1}{m} \cdot mEff_{t_{i-j}}(F) &\geq minEff \quad \vee \\ mEff_{t_{i-1}}(F) &= undef \end{aligned} \tag{7.20}$$

As a result, fragments are placed and removed less frequently, but also poorly-performing fragments are kept longer at a cache. The parameter m has to be set by administrators upon a careful observation of re-placements of fragments. Note that the behavior of the systems changes over time. Hence, a fragment that has been removed in stage t can be replaced in a later stage t' and even can produce the required amount that is defined by *minEff*.

7.5 Related Work

Continuous optimization problems have attracted many authors and a variety of solutions have been proposed. Many authors argue that most of the real-world problems are non-stationary and that the goal is no longer to find the optima, but to track their movement through the search space as closely as possible.

Initial approaches have been proposed for on-line or real-time optimization of industrial processes, e.g., [28, 30]. During the last decade, some of the static optimization techniques have been extended to cope with dynamic environments, e.g., Dynamic Response Surface Method (DRSM) [34], Adaptive Simulated Annealing [76], Dynamic Simplex Method [107]. Current reseach proposes evolutionary approaches to solve non-stationary problems, e.g., [46, 98]. The author of [13] surveys techniques that make evolutionary algorithms suitable for changing optimization problems. To cope with cyclic and repetitive pattern in the search space, these algorithms have been enhanced by dynamic and case-based memory models, e.g., [36, 11], that allow to keep track of bad choices during the search and to remember good decisions.

Existing techniques can be grouped into the categories of model-based, direct-search (or heuristic-search) and hybrid approaches. Model-based approaches aim at deriving a process model by analysing the input/output behavior of a system. However, whenever an accurate process model is difficult to obtain or a measurement is expensive or time-consuming, as in our case, direct-search, heuristic or hybrid techniques are more applicable.

We do not suggest a specific technique to solve our optimization problem. The proper choice of an appropriate technique requires the implementation and a comparison of different approaches which is beyond the scope of this work.

Instead, we propose a model in Chapter 9 to partially compute good-performing configuration parameters. That is, given a stage t and a obtained measurement $y_t = f_C(x_t, t)$ we develop a stochastic model of the simultaneous execution scheme that allows us to compute $y'_t = f_C(x'_t, t)$ offline for all $x'_t < x_t$ on a partial ordering $(\mathcal{Z}_t \times 2^{\mathcal{F}_t}, <)$ Due to the partialness of the model, the optimizer still has to apply another search strategy on the remaining part of the search space, but can use the model to quickly compute alternative good-performing configurations.

7.6 Summary and Discussion

The dynamics of client-server database systems causes a non-stationary optimization problem and therefore requires a real-time (on-line) optimization technique for setting the configuration parameters. The dynamics depends on various time-varying influencing factors on the request and resource level that make it impossible to predict user behavior and the load-conditions of the system.

To measure the performance we have introduced a couple of measures for the execution of stored procedures and the placement of fragments. We define the execution-performance in terms of the minimal processing time of a procedure, the maximal utilization of a cache or the maximal code balancing of an execution between cache and server. The fragment-performance is defined in terms of access frequencies for the read and write access, and the fragment-effectiveness that reflects the frequency of valid queries computations at the cache.

We model the behavior of the system in stages of equal length. The dynamic optimization problem defines the optimal performance of a stage by taking the behavior of previous stages into account. The task of the optimizer is to detect changes in the environment, such as user behavior or load-conditions, and to track the optima accordingly.

The presented performance measures and the resulting optimization problem can be extended into the following directions.

Proper Measurement of Synchronization Costs

The synchronization of a replicated fragment causes additional effort at the server. Given a fragment F, intuitively, the savings that are gained by executing queries on F at the cache should exceed its synchronization costs. Only then is the placement of the replicated fragment cost-effective.

To obtain such a situation, we have introduced the notion of fragment-effectiveness *mEff*(F) and the user-defined parameter *minEff*. Then, a fragment is only further replicated if *mEff*$(F) \geq$ *minEff* holds. However, to set the parameter *minEff* manually by the administrator, a precise observation of the synchronization effort is required.

To overcome this limitation, a cost model has to be developed that on the one side captures the precise synchronization effort of a fragment and on the other side allows its comparison to the savings of a fragment, e.g., $\mathit{mEff}(F)$. By such a model, the user-defined parameter *minEff* is not necessary and the optimization problem can be extended by a condition of the form

$$\begin{aligned} \forall F \in \mathit{repl}:\ \mathit{mEff}_t(F) &\geq mSyncCosts_t(F) \quad \vee \\ \mathit{mEff}_t(F) &= undef \end{aligned}$$

where $mSyncCosts_t(F)$ captures in some format the total processing to synchronize the fragment F. Then, the problem is define $mSyncCosts_t(F)$ in such a way that it is comparable to $\mathit{mEff}_t(F)$.

Local vs. Global Optimization

From the perspective of a cache, the server is just a resource that, due to different load-conditions, computes requests with a different efficiency. However, the load of the server also depends on the optimization decisions of all connected caches. If caches perform most of the procedure code by themselves, there is only a little load at the server and vice versa. Hence, a better efficiency can be achieved by a global optimizer that for the entire client-server system establishes the setting of the replication and split parameter.

A global optimization seems very promising, but it also requires the optimization of multiple caches at the same time. Furthermore, the optimizer should not run on the server which is often the central bottle neck of such systems. Instead, the optimization tasks should be performed by the database caches that communicate with each other. In all a difficult, but challenging task.

Chapter 8

Modeling the Process Time of the Simultaneous Execution Scheme

The simultaneous execution scheme performs the procedure code at cache and server in a parallel manner. As shown in Chapter 6, the resulting processing time depends on the split of the code, the frequency of valid query results, and the execution time of IO statements. Based on these parameters, we define a model for the simultaneous execution scheme that allows to predict the resulting processing time. We estimate these parameters from a completed stage, since according to Chapter 7 our model is of Markovian type . The feasibility of the model is evaluated on experimental data of the ONE-System. In Chapter 9 we will introduce an optimization technique in order to find a configuration at run time with best performance.

8.1 Problem Statement

Given a database cache and a completed data set at stage t, the first problem is to find a suitable data structure which captures all relevant IO statements that are executed by a procedure S in t. IO statements are relevant inasmuch as they affect the total processing time of S.

Intuitively, these are all IO statements at the server and all query results that are delivered by the cache and reused by the server. In Section 8.2 we look at these statements and define the fundamental notion of a *valid subsequence*. It captures a subsequence of a execution sequence at the server, where for all queries a valid result has been delivered by the cache. We show that these subsequences reveal almost complete information about the code execution at the cache that affects the processing time of S. Based on valid subsequences, we define in Section 8.3 a tree-based execution history for capturing all executions of S in t.

In Chapter 6 we have roughly calculated the processing time of a procedure as depicted in Figure 8.1. The idea is to compute the interweave of the code execution and therewith the processing time of a procedure by taking basic parameters as the execution time of IO statements, network communication time, verification time, etc. into account.

The second problem is to find a model that allows us to precisely calculate the average

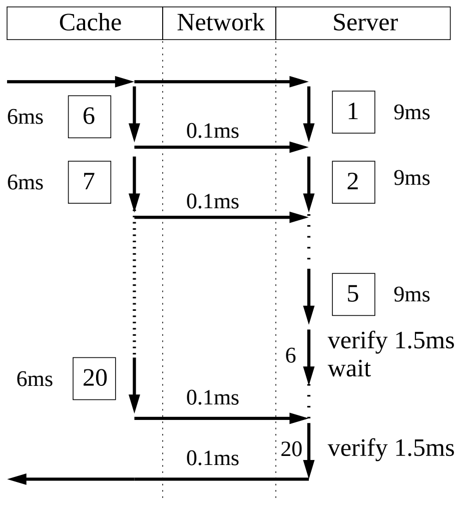


Figure 8.1: Calculating the processing time of the simultaneous execution scheme.

processing time of S in t. Such a model has to consider the waiting times for query results at the server which depend on the interweave of the execution.

In Section 8.4 we define all basic parameters that influence the processing time of the simultaneous execution scheme and provide an algorithm that computes the average processing time by simulating the behavior of the $eval_S(.)$ function at the server. Based on the experimental data of the ONE-System, we evaluate the preciseness of the model in Section 8.5. We show that there is a low deviation between the original observed and the calculated processing time.

8.2 Valid Subsequences

Given a simultaneous execution, the resulting execution sequences at cache and server precisely capture the set of executed IO statements. Execution sequences are initially defined by Definition 4.2 and further extended in Section 7.2.1. In the following we use execution sequences that consists of elements $(s, \#s, mode)$ where s is the statement identifier, $\#s$ the statement counter, and $mode \in \{exec, undef, invalid, valid\}$ the resulting execution mode.

The server constantly tries to use query results that are delivered by the cache. When an invalid result has been delivered, the server switches from shared to normal mode. Then, the remaining part is entirely executed at the server without considering any more results from the database cache. As a result, the execution sequence at the server can be viewed in two sequential parts — one for the shared mode and one for the normal mode.

Definition 8.1 (Valid Subsequence) *Let*

$$seq \;=\; (s_1, \#s_1, mode_1), \ldots, (s_n, \#s_n, mode_n)$$

be an execution sequence of a simultaneous execution at the server. A valid subsequence *is a subsequence of seq for $m \leq n$, if*

1. *$mode_i \neq invalid$ for $1 \leq i < m$, s_m is a query with $mode_m = invalid$ and the server switches to normal mode in s_m, or*
2. *$mode_i \neq invalid$ for $1 \leq i < m$, s_m is a query with $mode_m = valid$ and $mode_i \neq valid$ for $m+1 \leq i \leq n$, or*
3. *$m = 0$ with $mode_i \notin \{valid, invalid\}$ for $1 \leq i \leq n$*

Valid subsequences with $mode_m = invalid$ (case 1) represent the switch to normal mode, while executing the query s_m and those with $mode_m = valid$ (case 2) are used to represent executions where all query results are valid. We also allow a null subsequence (case 3), since a split can be chosen such that no query is executed at the cache. Hence, the modes *valid* and *invalid* can not occur in such sequences.

8.2.1 Implied Operations at the Cache

A valid subsequence represents the execution of IO statements at the server. In the following we take such a subsequence and reveal all information about the execution at the cache that follow from the simultaneous execution scheme.

The following claim defines this information. We refer to the $eval(.)$ which describes how IO statements are executed on data. Recall that the functions $eval_C(.)$ and $eval_S(.)$ have been introduced in Chapter 5 to implement cache- and server-specific behavior of the simultaneous execution scheme. Both functions call the original function $eval(.)$ that executes an IO statement on data. The claim is applicable to all procedures types and is independent of the chosen split parameter.

Claim 8.1 *Let S be a procedure with $split \in Z(S) \subseteq 2^{ID(S)}$ and*

$$seq = (s_1, \#s_1, mode_1), \ldots, (s_n, \#s_n, mode_n)$$

a valid subsequence. If $s_i \in split$ the following properties hold for $1 \leq i \leq n$:

1. *For a query s_i the cache executes $eval(.)$ with $mode_i = valid$ to execute s_i on cache data.*
2. *For $delete(.)$ operations s_i the cache executes $eval(.)$ to execute s_i on cache data.*
3. *For $insert(.)$ operations s_i it is unknown if the cache has executed $eval(.)$.*

Proof:
Property 1:

According to the function $eval_S(.)$ (see Section 5.4.2) a query is only valid if the cache has executed and delivered a correct query result. To execute the query, the cache performs Step 3e of the function $eval_C(.)$ which calls the original function $eval(.)$.

Property 2:

A *delete*(.) operation only results from DELETE and UPDATE statements in the original procedure code that have been replaced by a query q and a loop of low-level updates (see Section 4.2.3).

The statement UPDATE R SET A_j=<arith-expr> WHERE <cond> is translated into:

```
FOR $row$ IN SELECT ID,* FROM $R$ WHERE <cond> LOOP
    $row.A_j$ =<arith-expr'>;
    $delete(R, row.ID)$;
    $insert(R, row.ID, (row.A_1, \ldots, row.A_n))$;
END LOOP
```

Where <arith-expr'> results from replacing each A_j in <arith-expr> by $row.A_j$.

The statement DELETE FROM R WHERE <cond> is translated into:

```
FOR $row$ IN SELECT ID,* FROM $R$ WHERE <cond> LOOP
    $delete(R, row.ID)$;
END LOOP
```

The query q is necessary to retrieve the tuple identifiers *tid* that are required by the low-level updates.

If q is not executed at the cache, $delete(R, row.ID)$ also cannot be executed. Recall from the run time analysis in Section 5.5 that we do not consider split parameters that contain $delete(R, row.ID)$ but not q. Hence, there is a query $s_i \in split$ $(i < n)$ in the valid subsequence that corresponds to q with $mode_i = valid$. Its mode must be valid, since as per the definition of a valid subsequence only the last element s_n can be an invalid query.

For the valid query q the cache has executed the corresponding loop. Since the result of q is equal at cache and server (valid query), the cache also has performed the same number of loops. The loop assigns each row of the result of q to a local variable *row*. Hence, the cache calls $eval_C(.)$ for each $delete(R, row.ID)$ with the same parameters as the server.

The function $eval_C(.)$ (see Section 5.3.2) performs the Steps 2a, 2b and 2c. In Step 2a the statement $delete(R, row.ID)$ is checked for *undef* values. Since *row* is assigned by the loop and not further modified by the loop body, there can be no *undef* values. In Step 2b the availability of data is checked. The operation $delete(R, row.ID)$ is always executable on cache data, since it accesses a fragment that has already been used for q. Finally, Step 2c executes $delete(R, row.ID)$ on cache data by calling the function *eval*(.).

Hence, for *delete*(.) operations s_i with $s_i \in split$, the cache executes *eval*(.) to execute s_i on cache data.

Property 3:

> The *insert*$(R, row.ID, t)$ operation results only from `UPDATE` statements and the *insert*(R, t) only from `INSERT` statements. For both, an execution at the server does not necessarily imply an execution at the cache by the following observation:
>
> The value t in *insert*$(R, row.ID, t)$ is affected by computing
>
> $row.A_j$ `=<arith-expr'>;`
>
> in the loop of the replaced `UPDATE` statement (see above). The assigned expression can contain local variables that have not been assigned by the cache due to the partial execution. Hence, the expressions can be of value *undef*. Recall that the partial execution depends on the chosen split parameter. Hence, an operation *insert*$(R, row.ID, t)$ is not executed in case of *undef* values and executed in case all variables in `<arith-expr'>` have been properly assigned.
>
> The same argumentation holds for operations *insert*(R, t) that result from replacing `INSERT INTO` R `VALUES (<expr>,..,<expr>)` by
>
> ```
> row:=(<expr>,..,<expr>);
> insert(R,row);
> ```
>
> Each expression potentially contains *undef* values.

Hence, the execution of *eval*(.) for insert operations is unknown. □

8.2.2 Additional Operations at a Cache

Given a valid subsequence, Claim 8.1 shows the resulting executions of IO statements at a cache. However, to estimate the execution-performance we also have to show that the cache does not necessarily execute more than these statements. Consider, for example, the following `IF` statement that is executed between two valid queries of a valid subsequence.

```
IF <cond> THEN
      FOR row IN SELECT * FROM R LOOP
            ..
      END LOOP
END IF
```

If the server derives the condition *false* and the cache *true*, the cache can execute much more IO statements. Claim 8.1 only states which executions at the cache follow from executions at the server, but it does not show the completeness of these executions.

In the following we show that the cache does not execute additional queries, *delete*(R, tid), or *insert*(R, tid, t) operations. However, it can execute additional *insert*(R, t) operations. Recall from Section 4.2.3 that *delete*(R, tid) and *insert*(R, tid, t) result from `DELETE` and `UPDATE` statements and *insert*(R, t) from the `INSERT` statement.

First, we define the scope in which we view an execution at the cache as "additional". Then we make an appropriate claim. Finally, we discuss possible modifications of the function $eval_S(.)$ at the server that detect additionally executed $insert(R, t)$ operations at the cache.

Last Considered Notification

We only consider executions at the cache as long they affect the processing time of a procedure. According to Definition 8.1 (valid subsequence), the server does not consider further notifications after all IO statements of a valid subsequence *seq* have been executed at the server. Hence, only the execution of IO statements of a valid subsequence affect the processing time at the server.

Consider the last element of the valid subsequence *seq*. If it is a invalid query q, then exactly one notification is considered by the server. This notification indicates that the result of q will be invalid. If the last element is a valid query, two possible notifications are considered. The first indicates that the cache is going to execute the query and the second delivers the result of q. However, the server can also skip the first notification (see Section 5.4.2).

In the following we call the first notification of the last element in *seq* the *last considered notification*. There are three possibilities at the cache to send this notification:

1. Step 3 of the $eval_C(.)$ function puts a notification into the query result cache (QRC) if (1) a query expression contains an $undef$ value (e.g. caused by a variable that has not been assigned, but used in the query expression), (2) data is not available or (3) the query is going to be executed, but its result has not been delivered yet.

2. At the end of each execution at the cache, a notification *end* is put into the QRC. It indicates that for the current execution no more query results will be delivered.

3. Whenever the execution engine at the cache jumps over a query that is intended to be executed ($q \in split$), a $undef$ notification is put into the QRC (see Section 5.3.1). Then the server is informed that for this query no result will be delivered. The execution at a cache jumps over a query if the query is part of a branch of an `IF` statement whose condition is $undef$, or if the query is part of a body of a loop whose query is $undef$ (see also Section 5.3.1).

No Additional Operations at the Cache Except Insert Operations

Claim 8.1 lists all executions at the cache that can be derived from a valid subsequence. The following claim states that at a cache there can be additional $insert(R, t)$ operations, but not additional queries, $delete(R, tid)$ and $insert(R, tid, t)$ operations.

Claim 8.2 *Let S be a procedure that is simultaneously executed for a $split \in Z(S)$. Let seq be the resulting valid subsequence. Let $eval_C(s, \#s, e)$ be a call at the cache with $s \in split$ that does not send the last considered notification and that is executed before the last* *considered notification has been sent. Then,* $(s, \#s, mode) \in seq$ *holds for all such* s *for* $mode \in \{exec, valid\}$*, except when* e *is an* $insert(R, t)$ *operation.*

We do not consider the function call of $eval_C(.)$ that sends the last notification, since the notification is transmitted before the query in $eval_C(.)$ is executed. Hence, the query execution time does not affect the processing time at the server.
Proof: We separately prove the claim for queries and low-level updates.
Case I: Queries

Assume there is a call of $eval_C(s, \#s, e)$ at the cache with $(s, \#s, mode) \notin seq$ for $mode \in \{invalid, valid\}$. Then, independently from the execution of Step 3 in $eval_C(.)$, at least one notification will be sent to the server that documents the execution of the query s at the cache.

As soon as there is a notification that is not expected by the server or that does not correspond to the query that is currently processed, the server switches to normal mode and ignores all of the following notifications including query results. Recall from function $eval_S(.)$ (see Section 5.4.2) that the server always selects the oldest entry from the QRC.

Case IA: s is executed before a valid query in *seq*

Consider one of the queries q in *seq* with mode *valid*. If the cache executes another query $s \in split$ before q is executed, and also the server does not execute q, a notification of s appears before q in the QRC. Since the server always selects the oldest entry, s is retrieved instead of q. Hence, while executing q, the server detects an invalid query, since s does not correspond to the query q.
Since by the definition of a valid subsequence the result of q is valid, the server must have retrieved the correct entry for q from the QRC. Hence, the cache has not performed $eval_C(s, \#s, e)$ before a valid query in *seq*.

Case Bib: s is executed after the last valid query in *seq*

Consider the last valid query q in *seq* and the query s (as above) that is executed after q at the cache. Again the execution of s causes an entry in the QRC with the statement identifier of s.
Let q' be the invalid query in the valid subsequence. In order to verify q', the server retrieves the oldest entry of the QRC and compares the statement identifiers. Since s has been put first, it retrieves s instead of q'. As both are different, the server detects an invalid query and switches to normal mode. Hence, the query s itself causes the last considered notification. Since the Claim excludes calls of $eval_C(.)$ that send the last considered notification, such a s does not exist.

Case II: Low-level updates

We separately consider the three types of low-level updates. Let $eval_C(s, \#s, mode)$ be a call of a low-level update s with $mode = exec$.

Case Ilia: s is an $delete(R, tid)$ operation

As argued in the Proof of Claim 8.1, an $delete(R, tid)$ operation results from a `DELETE` or `UPDATE` statement that requires a query q to be executed at the cache. Hence, the cache must have called $eval_C(.)$ for q. According to the Claim the call does not send the last considered notification. By the results of **Case I** the query q and the is either not executed, or the query q has also been executed at the server. In the latter case, both results at cache and server must be equal. Thus, cache and server perform the same loop body and the server also executes the same $delete(R, tid)$ operation. Hence, $(s, \#s, mode) \in seq$.

Case Ii: s in an $insert(R, tid, t)$ operation

As argued in the Proof of Claim 8.1, an $insert(R, tid, t)$ operation results only from an `UPDATE` statement that requires a query to appear in a valid subsequence. Analogously to **Case Ilia**, $(s, \#s, mode) \in seq$.

Case Ii: s is an $insert(R, t)$ operation

For an $insert(R, t)$ no extra query is required at the cache. We show the existence of additional $insert(R, t)$ operations by the following example. Consider the procedure code

```
1 SELECT <attar> INTO var1 FROM R1 WHERE <cond>;
2 var2:=true;
3 IF var1 THEN
4       var2:=false;
5 END IF;
6 IF var2 THEN
7       INSERT INTO R2 VALUES ('error');
8 END IF;
```

The line of code represents the statement identifier. Assume the procedure is executed with $split = \{7\}$ and that the local variable `var1` is set to $true$.

The server executes the first query and both assignments on `var2`. Hence, `var2` is $false$ and the server does not execute the insert operation. The cache does not execute the first query, since $1 \notin split$. Hence, `var1` is set to $undef$ and the assignment `var2:=false` is not executed. As a result, `var2` is $true$ and the cache does execute the insert operation.

Note that the cache can only execute additional insert operations in branches that do not contain queries, $delete(R, tid)$, or $insert(R, tid, t)$ operations, since according to **Case I**, **Case Ilia** and **Case Ii** the cache does not execute additional queries, $delete(R, tid)$ and $insert(R, tid, t)$ operations.

□

In the following subsections we suggest two modifications of the function $eval_S(.)$ that reduce the number of additionally performed insert operations at the cache.

Detect Additional Operations For Each Delivered Query

For each query at the server for which a delivered result is considered, we perform an additional test to check if the cache has executed additional insert operations. For this, we add the following code between Step 1e and 1f of the $eval_S(.)$ function (see Section 5.4.2).

> *Let $(s, \#s, e, val)$ be an entry in the execution sequence seq' delivered by the cache with e as an $insert(R, t)$ operation. If there is no element $(s', \#s', e', val')$ in seq with $s = s'$, set normal mode and jump to 2.*

This modification checks whether the cache has executed insert operations that have not been executed by the server. For this, we simply check the execution sequence seq' that is delivered by the cache.

As a result, a valid query can also be rejected at the server if the cache has performed additional insert operations. Thus, by considering a valid subsequence, the only possibility for additional insert operations at the cache is in between a valid and invalid query. Between two valid queries, say q_1, q_2, there can be no such insert operations, since the call of $eval_S(.)$ for q_2 at the server would detect the additional insert operations and q_2 would be invalid.

Additional Notifications For Insert Operations

Another possibility is that a cache sends a notification for each insert operations by using the query result cache (QRC). Whenever the server is informed about an insert operation that it has not executed, it switches to normal mode. However, this would require a major change of our execution protocol. The advantage of this modification is that an additional insert is immediately known by the server.

In the following, we consider the first modification and accept possible additional inserts at a cache.

8.3 Capture the Run Time Behavior of Stored Procedures

At run time the execution of a procedure produces different execution sequences and therefore different valid subsequences. First, we discuss an upper bound for the number of different valid subsequences, then we define a tree-based run time history of stored procedures.

8.3.1 An Upper Bound for the Number of Valid Subsequences

At run time a stored procedure S produces an execution sequence

$$seq \quad = \quad (s_1, \#s_1, mode_1), \ldots, (s_k, \#s_k, mode_k)$$

of executed IO statements (queries and low-level updates) at the server. Based on this we define an execution path through the procedure code of S by the sequence

$$path \;=\; (s_1, \#s_1), \ldots, (s_k, \#s_k)$$

which only contains the nodes $n_i = (s_i, \#s_i)$ with the statement identifier $s_i \in ID(S)$ and the number $\#s_i$ of repetitions of s_i for all $1 \le i \le k$.

Given a $split \in Z(S) \subseteq s^{ID(S)}$, the server starts the execution of S in shared mode. In this phase of the execution, the $mode_i$ of all queries $s_i \in split$ is *valid* and that of all other IO statements s_i is *exec*. Once an invalid query has been delivered by the cache, the server switches to normal mode. Then, the $mode_i$ of all queries $s_i \in split$ is *invalid* and that of all other IO statements s_i is *exec*.

Given an execution path of length k, the different combinations of the values of $mode_i$ can be characterized by

$$(\ valid \mid exec\ )^{(0,a)}, (\ invalid \mid exec\ )^{(0,k-a)}.$$

It represents all possible sequences of the execution mode for the sequence *seq*. Cardinalities are denoted by $(0, a)$ or $(0, k - a)$ with 0 as minimal number and a as maximal number of repetitions. The left expression represents the shared mode and right one the normal mode.

Let $m \le k$ be the number of queries of *path*. There are exactly m possible sequences with *invalid* on the i-th query for $1 \le i \le m$, and there is one sequence where all queries are valid. Hence, there are $m + 1 \le k + 1$ possible valid subsequences for an execution path of length k.

At run time a procedure produces different execution paths. We summarize these paths by a tree-based *execution history*. The tree is a special directed acyclic graph. There is exactly one node (root node) that has no incoming edges and several nodes (leaf nodes) that have no outgoing edges. A path in the tree is a sequence of nodes $n_1, \ldots, n_i$ $(i > 0)$ of the graph where each node n_l is a direct predecessor of node n_{l+1} $(1 \le l \le i-1)$. There is exactly one path from the root node to each other node of the graph. A path from the root node to a leaf node is called root-path. Given a node n and a path p starting at n, we denote both as $n; p$.

Definition 8.2 (Execution History) *Let t be a stage, S a procedure and $path_1, \ldots, path_k$ all resulting distinct execution paths in t. The execution history with a root node r and leaf nodes $n_1, \ldots, n_k$ is denoted $hist_t(S)$ and there is exactly one and only root-path in $hist_t(S)$ for each triple $((r; path_i); n_i)$ $(1 \le i \le k)$.*

The root node of $hist_t(S)$ is not part of an execution path. We had to add this root node as two different execution paths do not necessarily start with the same IO statement. Consider, for example, the following piece of procedure code:

```
BEGIN
        IF <cond> THEN
```

```
            INSERT INTO R1 VALUES (a);
        ELSE
            INSERT INTO R2 VALUES (a);
        END IF
        ..
    END
```

Both insert operation have a different identifier. Hence, execution paths can have different first elements. The corresponding graph of the execution history is shown at the left side of Figure 8.2.

Further, we had to add extra leaf nodes that also are not part of an execution path, since an execution path might fully include another execution path, such that both paths cannot be distinguished. Consider, for example, the following piece of procedure code:

```
    BEGIN
        SELECT ID INTO var1 FROM R1 WHERE <cond>;
        IF var1>0 THEN
            RETURN;
        END IF
        SELECT ID INTO var2 FROM R2 WHERE <cond>;
        RETURN
    END
```

At run time there are two possible execution paths. The corresponding graph of the execution history is shown at the right side of Figure 8.2. The first includes only the first query and the second both. Hence, the second path fully includes the first. The extra leaf nodes represent both `RETURN` statements and allow us to distinguish between both paths in the execution history.

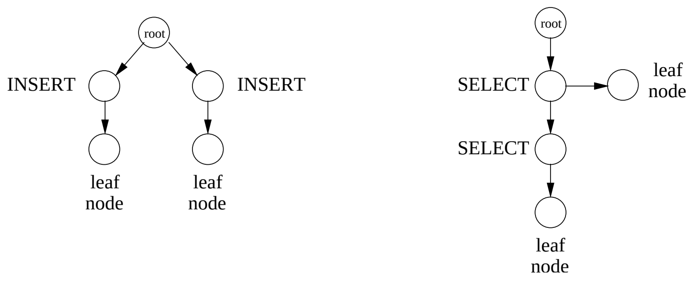


Figure 8.2: Example of execution histories.

The definition of the execution history only considers distinct execution paths $path_1, \ldots, path_k$. As a result, the last node of an execution path is always connected to a single leaf node n_k. If such a last node would be connected to more than one leaf node, then by the definition of the history there must be two paths $path_i = path_j$ with $i \neq j$. Then, $path_i$ and $path_j$ are not distinct. Hence, a leaf node represents exactly one execution path.

Let $split \in Z(S)$, k be the number of nodes in $hist_t(S)$ and $m \leq k$ be the number of nodes in $hist_t(S)$ that represent a query in $split$. Then, each path from the root node of $hist_t(S)$ to one of the m nodes represents valid subsequences, where the last node of the valid subsequence represents the invalid query. Hence, in $hist_t(S)$ there are at most $m \leq k$ valid subsequences that end with an invalid query.

Leaf nodes represent valid subsequences, where all queries are valid. In a tree with k nodes there are at most k leaf nodes which do not correspond to nodes that represent queries. Hence, there are at most k valid subsequences in a history and each valid subsequence is represented by a single node in the tree.

8.3.2 Execution Histories of Stored Procedures

The execution history $hist(S)$ of a procedure S is the fundamental data structure by which we define a model for estimating the performance related to different split parameters. In the following paragraphs we extend the history step by step and add more information about the run time behavior of a procedure.

We assume that the resulting execution sequence of the server is sent to the cache after the execution of a procedure has been completed. The cache adds the resulting execution path to the history, if it does not exist yet, e.g. added by a previous execution of a procedure that has followed the same path. Additionally, the cache maintains several counters as explained below. Thus, the history is automatically updated by the cache during run time.

I. Capture Frequency of Valid Subsequences

As noted above, an execution history with k nodes represents at most k valid subsequences. For collecting the frequency of these different subsequences at run time, we add the following counters to those nodes.

Let n be a node that represents a query. The counter $c_{invalid}(n)$ captures the number of valid subsequences during stage t of which the query is invalid. Recall that when this query is executed the server switches to normal mode, where no more delivered results are considered.

The counter $c_{valid}(n)$ is added to leaf nodes. It captures the number of valid subsequences during a stage t where all queries are valid. Recall that after s the server expects no more results from a cache. Hence, the cache has delivered only valid results during the entire execution of a procedure.

II. Execution Frequency of IO Statements

Besides valid subsequences, we capture the execution frequency of IO statements on different execution paths at the server. Each non-leaf node n of the tree is extended by a counter $c_{exec}(n)$. For each execution path at run time at the server, the counter $c_{exec}(n)$ is incremented by one for each node on the path. Hence, it is increased for each call of the function $eval_S(.)$. For the root node r of $hist_t(S)$, the value $c_{exec}(r)$ represents the total number of executions of S in stage t.

Let n be a node that represents a query. Additionally, we add the counter $c_{reuse}(n) \leq c_{exec}(n)$ which is increased whenever the function $eval_S(.)$ has reused the delivered result instead of executing the query on server data.

III. Frequency of Insert Operations

As shown by Claim 8.1, a valid subsequence does not reveal the number of executions of insert operations at the cache. Only the execution of queries and delete operations is known. For nodes that represent an insert operation, we add a counter $c_{insert}(n)$ that captures the number of executions at the cache. It is increased whenever the insert operation on the execution path is also executed by the cache.

8.4 Modeling the Simultaneous Execution Scheme

Given a procedure S, statements $s \in ID(S)$ and the execution history $hist_t(S)$, we can model the resulting processing time PT which includes idle times at the server.

8.4.1 Basic Execution Parameters

Our model used the following basic parameters.

1. $pt_C(s)$ - Processing time of statement s at the cache.
2. $pt_S(s)$ - Processing time of statement s at the server.
3. t_{cache} - Access time of the query result cache (QRC).
4. t_{net} - Network communication time between cache and server.
5. t_{verify} - Verification time at a cache during a call of $eval_C(.)$. It includes the function calls of $access(.)$ that determine the fragment access (see Definition 4.9).
6. t_{verify} - Verification time at the server during a call of $eval_S(.)$. It includes the function calls of $access(.)$ and the comparison of the execution sequences (see Step 1f and 1g of Definition 5.4).
7. t_{final} - Additional overhead of an execution. This includes the commit phase and the execution of procedure code that does not contain IO statements.

The first two parameters are related to each procedure, while the others apply to all executions at a cache.

The parameters represent averages in stage t. Since the verification includes a comparison of fragment versions and update sequences, we assume that it is independent from the query and its result. Further, we do not distinguish between read/write access of the QRC and the size of the read or written result.

The final commit is captured within t_{final}. We summarize each commit by this time and do not consider the issue that a different number of updates can cause different commit times.

For a node $n = (s, \#s)$ in the execution tree we also use $pt_C(n)$ instead of $pt_C(s)$ and $pt_S(n)$ instead of $pt_S(s)$.

8.4.2 A Model of the Simultaneous Execution Scheme

First, we present the basic model for capturing the simultaneous execution scheme. We consider a procedure S with its execution history $hist_t(S)$ of length k and a valid subsequence

$$(s_1, \#s_1, mode_1), \ldots, (s_i, \#s_i, mode_i)$$

with $1 \leq i \leq k$, $mode_i \in \{invalid, valid\}$ and $mode_j \in \{exec, valid\}$ for $1 \leq j \leq i-1$. Note that the execution path does not include the root node and leaf nodes.

We calculate the processing time of $n_1, \ldots, n_k$ in two steps:

- First, we compute the execution time of the valid subsequence $n_1, \ldots, n_i$ at the server which represents the shared mode and includes the idle times.
- Second, we compute the execution time of $n_{i+1}, \ldots, n_k$ where all remaining IO statements are processed at the server without considering delivered query results from the cache.

1. Processing Time of a Valid Subsequence

The execution time of a valid subsequence $n_1, \ldots, n_i$ is calculated by the following algorithm. It simulates the behavior of the $eval_S(.)$ function at the server and includes waiting times. We have to carefully handle waiting times as the following observation demonstrates.

Let t_1 be the time when a cache has delivered a result and t_2 the time when the server expects the result. The waiting time at the server is computed by $t_1 - t_2$. A positive waiting time means that the server has to wait. A negative waiting time means that a cache has delivered the result before it is required by the server. However, in the latter case the effective waiting time at the server is 0, since the result has already been delivered.

Algorithm 8.1 (Processing Time of a Valid Subsequence)

Syntax:	$PT_{sub}(split, (n_1, \ldots, n_i), mode)$, $n_j = (s_j, \#s_j)$, $1 \leq j \leq i$
Input:	*split parameter, valid subsequence, execution mode of last query of valid subsequence*
Output:	*processing time in shared mode*, PT_S
Local Variables:	PT_C *processing time at the cache*, PT_S *processing time at the server*

```
1. // Initialize, the processing time at cache and server.
   $PT_C = PT_S = 0$

2. // Compute the processing time stepwise for each element of the valid subsequence.
   // The last query $n_i$ is handled separately.
   For $j = 1$ to $i - 1$ do
```

```
(a) // The cache does not execute IO statements that do not appear in split.
    // Hence, $PT_C$ does not change.
    // At the server we add the execution time of the IO statement $s_j$.
    If $s_j \notin split$ then $PT_S = PT_S + pt_S(s_j)$ endif
(b) // A delete operation in split at the server is according to Claim 8.1 also
    // executed at the cache. Additionally the cache has to determine
    // the fragment access $t_{verifyC}$ for $s_j$.
    If $s_j \in split$ and $s_j$ is a delete operation then
        $PT_S = PT_S + pt_S(s_j)$
        $PT_C = PT_C + t_{verifyC} + pt_C(s_j)$
    endif
(c) // Analogous to delete operations we handle insert operations.
    // The factor represents the average ratio of inserts and executions
    // at the cache. It is used, since according to Claim 8.1 the
    // executions of inserts is unknown and according to Claim 8.2
    // there can be further inserts.
    If $s_j \in split$ and $s_j$ is an insert operation, then
        $PT_S = PT_S + pt_S(s_j)$
        $PT_C = PT_C + t_{verifyC} + \frac{c_{insert}(n_j)}{c_{exec}(n_j)} \cdot pt_C(s_j)$
    endif
(d) // For queries in split waiting times have to be computed.
    If $s_j \in split$ and $s_j$ is query then
        // The cache checks if $s_j$ can be executed and
        // sends a message to the server.
        $PT_C = PT_C + t_{verifyC}$
        // The server checks the QRC.
        $PT_S = PT_S + t_{cache}$
        // If no entry exists, it has to wait for the message.
        // Note that the messages is delayed by $t_{net}$.
        $PT_S = PT_S + wait(max\{PT_C + t_{net} - PT_S), 0\}$
        // The server checks the fragment access and verifies the query.
        // Note that up to $n_{i-1}$ all queries are valid.
        $PT_S = PT_S + t_{verifyS}$
        // In the meantime the cache executes the query and
        // delivers the result.
        $PT_C = PT_C + pt_C(s_j)$
        // The server has possibly to wait for the result.
        $PT_S = PT_S + wait(max\{PT_C + t_{net} - PT_S, 0\})$
    endif
endfor
```

```
3. // If the last query $s_i$ is invalid, wait at most for the first message.
   // Check fragment access, verify the query and execute $s_i$ at the server.
   If mode = invalid then
           $PT_C = PT_C + t_{verifyC}$
           $PT_S = PT_S + t_{cache}$
           $PT_S = PT_S + wait(max\{PT_C + t_{net} - PT_S, 0\}) + t_{verifyS} + pt_S(s_i)$
   endif

4. // If the last query $s_i$ is valid, proceed analogously to Step 2d.
   If mode = valid then
           $PT_C = PT_C + t_{verifyC}$
           $PT_S = PT_S + t_{cache}$
           $PT_S = PT_S + wait(max\{PT_C + t_{net} - PT_S, 0\}) + t_{verifyS}$
           $PT_C = PT_C + pt_C(s_i)$
           $PT_S = PT_S + wait(max\{PT_C + t_{net} - PT_S, 0\})$
   endif

5. Return $PT_S$
```

Our model does not consider insert operations that have been executed at the cache, but not at the server. Further, we do not distinguish whether the last invalid query is detected in the Steps 1b-1d or 1f-1g of the $eval_S(.)$ function. Only the latter two steps cause a verification time $t_{verifyS}$. However, to show the feasibility of our approach the above model is sufficient.

2. Processing Time of an Execution Path

Given the above execution path $n_1, \ldots, n_k$ with $n_1, \ldots, n_i$ as the valid subsequence, the total processing time is calculated by the following formula.

$$PT_{sub}(split, (n_1, \ldots, n_i), mode) = PT_{sub}(split, (n_1, \ldots, n_i), mode_i) + \sum_{j=i+1}^{k} pt_S(n_j) + t_{final}$$

For the remaining part of the execution we simply add the execution time of all IO statements and the additional overhead as the commit time. Recall that after executing a valid subsequence, the server runs either in normal mode where no computations of the cache are taken into account, or the server does not expect more queries. In both cases there is no waiting time and no additional overhead that is caused by accessing the query result cache or verifying query access.

8.4.3 The Average Processing Time of a Procedure

We have shown the calculation for a single execution path. However, at run time a procedure S produces different execution paths with different frequencies. In the following we determine the average processing time over all execution paths that have been recorded in $hist_t(S)$.

Consider a valid subsequence for S that ends with a node n. The node n can be connected to multiple child nodes in $hist_t(S)$ that represent different execution paths that are executed after n. Clearly, the processing time of the valid subsequence depends on the path from n to a leaf node that is chosen at run time. Let $N_{sub}(n)$ be the set of nodes that can be reached by following the outgoing edges of n. The node n is executed $c_{exec}(n)$ times at run time. A node n' in $N_{sub}(n)$ is executed at most as much as n, hence $c_{exec}(n') \leq c_{exec}(n)$ holds.

Let $n_1, \ldots, n_k$ with $k > 0$ be a valid subsequence in $hist_t(S)$ without the root node at the beginning, n_k a node that represents a query and an execution mode $mode \in \{invalid, valid\}$ of node n_k. Then, its average processing time over all possible paths in $N_{sub}(n_k)$ can be approximated by

$$PT_{sub}(split, (n_1, \ldots, n_k), mode) \;+ \sum_{n' \in N_{sub}(n_k)} \frac{c_{exec}(n')}{c_{exec}(n_k)} \cdot pt_S(n') + t_{final} \qquad (8.1)$$

We add the execution time of all corresponding IO statements in $N_{sub}(n_k)$ and weight it by the execution frequency

$$\frac{c_{exec}(n')}{c_{exec}(n_k)}$$

of a node n'. Note that the execution time of n_k is considered within $PT_{sub}(.)$.

Then, the average processing time of a procedure results from summing the processing time over all valid subsequences w.r.t. to their execution frequencies. Given the total number $c_{exec}(r)$ of executions of a procedure with r as the root node of the history, the execution frequency of a valid subsequence, represented by a node n, is defined by

$$\frac{c_{invalid}(n)}{c_{exec}(r)} \qquad \frac{c_{valid}(n)}{c_{exec}(r)}$$

where n represents either a query or a leaf node with $c_{exec}(r) > 0$. Since an execution must result into a valid subsequence, the sum over the counters $c_{invalid}(n)$ and $c_{invalid}(n)$ is $c_{exec}(r)$.

The following Algorithm computes this average processing time according to Equation 8.1.

Algorithm 8.2 (Average Processing Time of a Procedure)

Syntax:	$PT(split, hist_t(S))$
Input:	*split parameter, execution history of a procedure* S, t_{final}
Output:	*average processing time in stage* t
Side Effects:	*returns* $undef$ *for* $c_{exec} = 0$

```
1. // initialize processing time
   $PT = 0$

2. // for all valid subsequences with an invalid query at the end
   For all distinct root-paths $r, n_1, \ldots, n_k$
```

```
    with $r$ at the root node and $c_{invalid}(n_k) > 0$:

    $$PT = PT + \frac{c_{invalid}(n_k)}{c_{exec}(r)} \cdot \left( PT_{sub}(split, (n_1, \ldots, n_k), invalid) + \sum_{n' \in N_{sub}(n_k)} \frac{c_{exec}(n')}{c_{exec}(n_k)} \cdot pt_S(n') \right)$$

3. // for all valid subsequences without invalid queries
   For all distinct root-paths $r, n_1, \ldots, n_k, n_{leaf}$
   with $r$ at the root node, $n_{leaf}$ a leaf node and $c_{valid}(n_{leaf}) > 0$:

   // determine the last valid query in the path
   Let $n_q = (s, \#s)$ be the last node in $n_1, \ldots, n_k$ with $s$ as query and $s \in split$

    $$PT = PT + \frac{c_{valid}(n_{leaf})}{c_{exec}(r)} \cdot \left( PT_{sub}(split, (n_1, \ldots, n_q), valid) + \sum_{n' \in N_{sub}(n_q)} \frac{c_{exec}(n')}{c_{exec}(n_q)} \cdot pt_S(n') \right)$$

4. Return $PT + t_{final}$
```

8.5 Preciseness of the Model

We have evaluated the preciseness of our estimation procedure and the correctness of our model within various experiments of the simultaneous execution scheme. Therefore we compared the observed and calculated processing time of stored procedures.

8.5.1 Evaluation Set

In Chapter 6 we performed experiments with one- and four-client systems. Each experiment has been run for a single stored procedure that is executed for 273 different configurations: 21 different values of the split parameter and 13 different values for the delay of the synchronization of cache data. Recall, that the latter parameter is used to generate different error rates. For each of the 273 configurations the procedure has been executed 150-420 times within a stage. The number of executions depends on the number of clients and the chosen split parameter which balances the code between cache and server.

Each execution of the procedure for a specific configuration produces an individual execution history. We present two such histories in Figure 8.3 that correspond to the experiment at Figure 6.18. Recall that the procedure is a sequence of 20 queries. Hence, its execution history contains only a single execution path. Both sequences show the configuration where the server executes the queries 1 to 5 and the cache 6 to 20. The first sequence refers to the lowest possible update delay and the second for the highest. While the first shows that almost all 15 query executions are valid (green box), the second produces a high number of

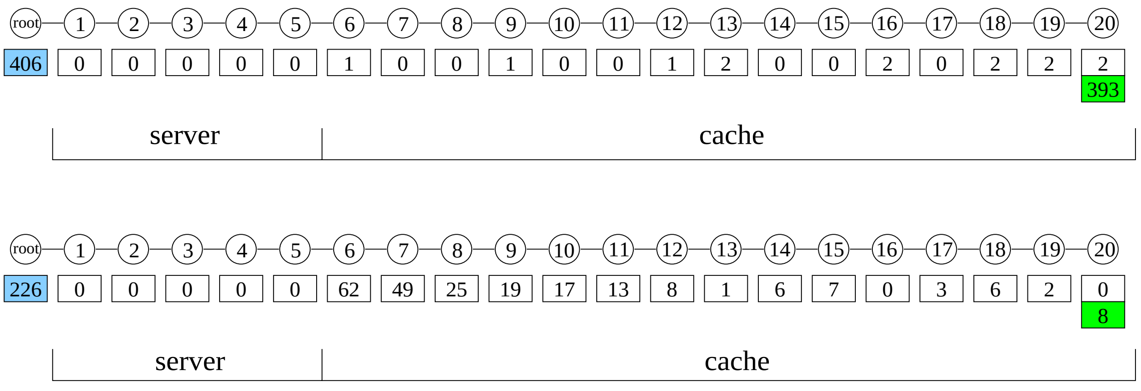


Figure 8.3: Example of an execution history. The blue box represents the counter c_{exec}, the white box $c_{invalid}$ and the green box a leaf node and c_{valid}. First (second) sequence refers to lowest (highest) possible update delay.

invalid queries (white boxes). We have skipped other counters at the nodes, since their value follows from the presented ones due to the single path in the history. The counter c_{exec} at the root node represents the total number of executions which is equal to the values of all other counters c_{exec}.

As a result of the experiment, we obtain for each of the 273 configurations a processing time, an execution history, and the values of the basic parameters. Each value of the processing time represents the average over all executions within the 80 seconds. The basic parameters are (see also Chapter 6):

1. $pt_C = 5.8$ms is constant for all configurations,
2. pt_C ranges from 5ms to 16ms according to the configuration,
3. $_{net} = 0.1$ms,
4. $t_{verifyS}$ ranges from 0ms to 2ms according to the configuration,
5. $t_{verifyC}$, t_{cache} have been set to 0, since the ONE-System does not implement fragment detection and a query result cache and
6. $t_{final} = 3$ms represents the average observed overhead of 2-4ms.

Given this information, we are able to compute the average processing time according to Algorithm 8.2 for each of the 273 configurations.

8.5.2 Compare Observed and Calculated Processing Time

Figures 8.4 and 8.5 show the observed and calculated processing time of the experiments in the Figures 6.12 and 6.18. As the figures show, both values are very close even in case of a high error rate. Since the Figures do not reveal the precise deviation of the observed

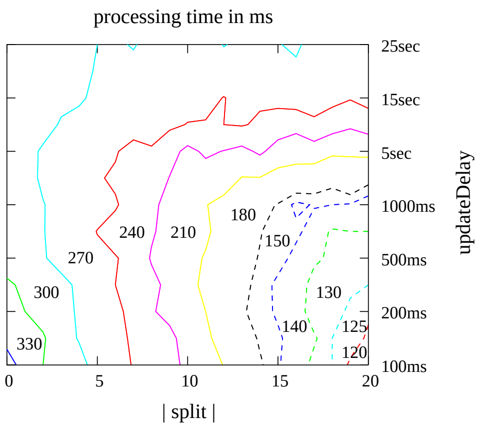


Figure 8.4: Observed (left) and calculated (right) processing time for four clients with dependent queries.

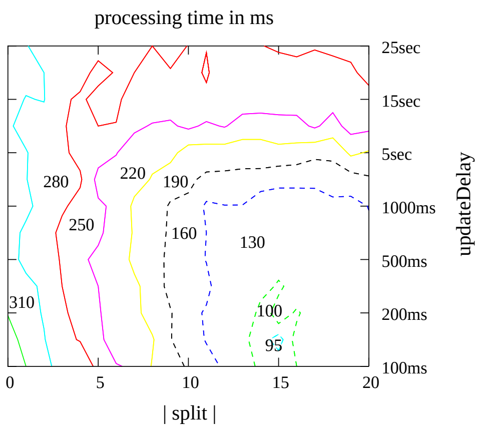


Figure 8.5: Observed (left) and calculated (right) processing time for four clients with independent queries.

and calculated values, we take a closer look at the frequency density and distribution of the deviation in the next Section.

8.5.3 Analyse Frequency Density and Distribution

As result of the above calculation we get a set of tuples (obs_i, cal_i) with $1 \leq i \leq 273$, obs_i as the observed and cal_i as the calculated processing time. To analyse the deviation of observed and calculated values, we define two characteristics — the frequency distribution $F(x)$ and the frequency density $f(x)$ for the error rate

$$1 - \frac{obs_i}{cal_i}.$$

The first represents the frequency of error rates below a given x. For $-\infty < x < \infty$ and $1 \leq i \leq 273$ it is defined as

$$F(x) \quad = \quad \frac{1}{273} \cdot \left| \left\{ (obs_i, cal_i) \;\middle|\; 1 - \frac{obs_i}{cal_i} \leq x \right\} \right| \tag{8.2}$$

Note that $F(-\infty) = 0$ and $F(\infty) = 1$.

The frequency density $f(x)$ tells us how the different values of x (deviations) are distributed over the 273 pairs. It is defined as a proxy of $F(x)$.

Figure 8.6 shows the resulting frequency density and distribution for the experiments of Figures 8.4 and 8.5. The dotted box marks the deviation of the interval $[x_1, x_2]$ with

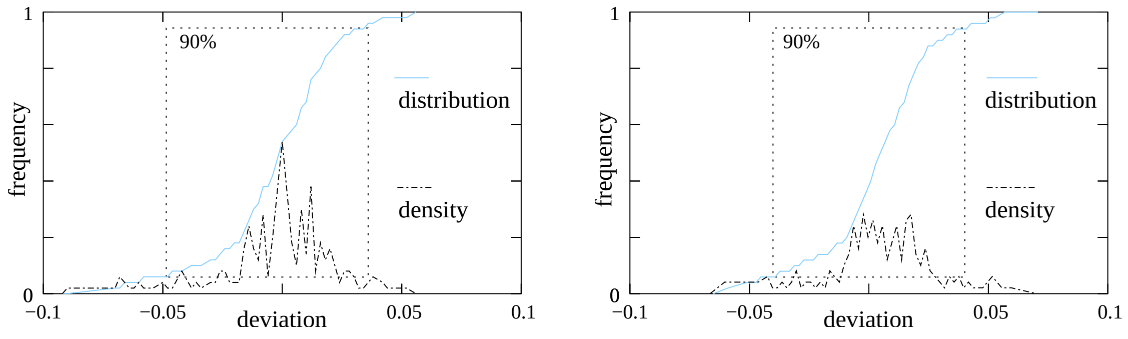


Figure 8.6: Frequency density (black) and distribution (blue) for simultaneous execution scheme without (left) and with (right) independent queries (four clients).

$f(x_1) = 0.05$ to $f(x_2) = 0.95$. It shows that the deviation is in a range of -5% and 5% for over 90% of the observations.

8.6 Summary and Discussion

Given a cache, a server and a load at stage t, we have developed a model that calculates the average processing time of a procedure executed in t. The model requires the parameter *split* of the procedure code at t, the execution frequencies of valid subsequences and a couple of basic parameters. The preciseness of the model has been evaluated with respect to error rates of observed and computed performance averages.

Algorithm 8.2 computes the average processing time for the given *split*. A challenging question arises when running the algorithm with $split' \neq split$. In this situation, Algorithm Algorithm 8.2 computes the average processing time for the behavior of a procedure, but for a another split of the procedure code. Intuitively this allows to compute the processing time (performance) of different split parameters. However, there are some hidden problems that we discuss in detail in the next Chapter. Furthermore, we show how the model can be used to provide a partial soultion of the optimization problem as defined in Chapter 7.

Extensions

A weakness of our model is the separate handling of $insert(R, t)$ operations, whose execution at a cache is not fully captured by an execution history. Therefore, we use a special counter c_{insert} that represents those cases. At the end of Section 8.2.2 we have discussed possible extentions of the simultaneous execution scheme to overcome this limitation.

As mentioned in Section 8.4.2, the model of calculating the processing time of a valid subsequence can be improved by mapping the behavior of the function $eval_S(.)$ more accurately. This concerns the handling of the first invalid query of a valid subsequence which results from two cases. The first detects an invalid query on the read entry of the query result cache (Step 1b-1d of $eval_S(.)$) and the second requires the determination of the fragment access of queries and low-level updates at the server which cause additional computational overhead (Step 1e of $eval_S(.)$). Currently, our model only covers the second case. If both cases are handled properly, we expect an improved precisness of the model.

Besides the processing time of a procedure, we have defined in Section 7.2 alternative performance measures for a maximal cache utilization and a maximal code balancing between cache and server. To calculate the measures, both require the idle time of a cache during an execution of a procedure. This time can only be determined if the total processing time of a procedure at the cache is known. Hence, the calculated processing time is a base for calculating both measures.

The computation of the processing time $mPT(.)$ of a procedure is the base for computing the other performance measures that target on a maximal cache utilization and maximal code balancing between cache and server. As defined in Section 7.2, both require the idle time of a cache during an execution of a procedure. This time can only be determined if the total processing time of a procedure at the cache is known.

Limitations

The use of execution frequencies for queries and low-level updates requires a reasonable high amount of executions at run time. Obviously, the model will be imprecise if only a few executions took place. In this case, the counters measure badly the average behavior of a procedure.

Chapter 9

Run Time Optimization

In Chapter 8 we have defined and evaluated a model for a simultaneous execution scheme that allows us to calculate the resulting processing time of a procedure. The model uses run time data of a completed stage. In this chapter we use the model to predict the execution-performance of a cache, hence allowing us to predict the processing time of procedures given alternative configurations. As we will show, the model provides a partial solution to the optimization problem, too. Furthermore, we evaluate the precision of the prediction technique and discuss a greedy algorithm as an example for integrating the prediction technique into an optimization algorithm.

9.1 Overview

In Chapter 7 we have formulated the optimization problem that has to be solved by the optimizer of a database cache: Given a database cache that is connected to a server, a set $\mathcal{S}_t$ of stored procedures, run time data of a completed stage t and unknown future load conditions in $t' > t$, find a configuration $(z_{t'}, repl_{t'})$ for the next stage t' which minimizes the execution-performance $mPerf_{t'}(.)$ for all stored procedures $\mathcal{S}_t$

$$(z^*, repl^*) = arg \underset{\substack{z_{t'} \in \mathcal{Z}_t \\ repl_{t'} \subseteq \mathcal{F}_t}}{min} \sum_{S \in \mathcal{S}_t} mPerf_{t'}(S, z_{t'}, repl_{t'})$$

subject to access, efficiency and storage restrictions on the fragments in $repl_{t'}$. The restrictions are defined in Section 7.4. Recall that $\mathcal{Z}_t \times \mathcal{F}_t$ is the configuration space where $\mathcal{F}_t$ represents the set of all fragments and $\mathcal{Z}_t$ all combinations of split parameters for the procedures in $\mathcal{S}_t$. As discussed in Section 7.2, the execution-performance $mPerf_{t'}(.)$ can be defined in terms of the (1) processing time $mPT_{t'}(.)$, (2) cache utilization $mIdleC^{+}_{t'}(.)$ or (3) code balancing $mIdle_{t'}(.)$. In this Chapter we only deal with (1) and set $mPerf_{t'}(.) = mPT_{t'}(.)$.

Given t, the behavior of a cache is captured by a function $y_t = f_C(x_t, t)$ where y_t is a vector of run time measures (see also Chapter 7) and x_t the configuration $x_t = (z, repl)$. As stated in Section 8.3 the run time behavior of the stored procedures $\mathcal{S}_t = \{S_1, \dots, S_k\}$ is captured by execution histories. Therefore we have available

$$\big(hist_t^{(i)}(S_i)\big)_{i=1,\dots,k} \tag{9.1}$$

where we also use the notation $y.hist_t^i(S_i)$, since these histories are run time measures too. According to Chapter 8, the model of the simultaneous execution scheme allows us to calculate the average observed processing time $mPT_t(S_i)$ in y_t with $1 \leq i \leq k$, hence to predict the execution-performance from data of the execution histories.

$$\sum_{i=1}^{k} y.mPT_t(S_i) \;\approx\; \sum_{i=1}^{k} PT(z.S_i, y.hist_t^i(S_i)) \tag{9.2}$$

$PT(.)$ represents the calculated processing time according to Algorithm 8.2. The notion $z.S_i$ represents the split parameter in z for the procedure S_i.

9.1.1 Problem Statement

We define the problem as follows:

> Given the observed run time behavior $y_t = f_C(x_t, t)$ of a completed stage t with $x_t \in \mathcal{Z}_t \times 2^{\mathcal{F}_t}$ (see also Section 7.4), find
>
> 1. a partial order $(\mathcal{Z}_t \times 2^{\mathcal{F}_t}, \leq)$ and an algorithm that predicts the run time performance of $y'_t = f_C(x'_t, t)$ for the same (already completed) stage t and for all configurations $x'_t \leq x_t$, and
> 2. a run time optimization algorithm that utilizes this prediction algorithm in order to select a configuration with best performance for each stage.

We predict the performance of stage t, since the run time behavior of future stages is unknown. When such an algorithm can predict the performance of all $x'_t \leq x_t$, the optimizer is able to select the configuration with the best performance for the next stage $t' > t$. Note that for the case $x'_t = x_t$ we already have such a prediction algorithm as stated by Equation 9.2.

9.1.2 Problem Solution

In this section we introduce a solution of the problem and point to the appropriate subsequent sections where the involved subproblems are discussed in detail.

Within our solution we use the following partial order:

Definition 9.1 *Let $x, x' \in \mathcal{Z}_t \times 2^{\mathcal{F}_t}$ be configurations with $x = \big((split_1, \dots, split_k), repl\big)$ and $x' = \big((split'_1, \dots, split'_k), repl'\big)$. The partial order is defined by $(\mathcal{Z}_t \times 2^{\mathcal{F}_t}, \leq)$ where $x' < x$ holds if $split'_i \subseteq split_i$ for $1 \leq i \leq k$ and $repl = repl'$.*

The configuration x' performs *less* or *equal* computations at the cache than x, since the split parameters of x' contain less or equal IO statements. Hence, the configuration x' considers as least as much IO statements as x, thus requiring at most the same set $repl = repl'$ of replicated fragments. In the following we use the notation $x = (z, repl)$ with $z \in \mathcal{Z}_t$, $repl \subset \mathcal{F}_t$ and $z = (split_1, \ldots, split_k)$. ($x' = (z', repl)$ follows analogously) and $z' < z$.

Given the above notions, our prediction technique operates in the following phases:

I. Predict Execution Behavior

Let t be a completed stage t, $(z, repl)$ a configuration and y_t be the resulting behavior including the execution histories $hist_t^{(1)}(S_1), \ldots, hist_t^{(k)}(S_k)$ for all k stored procedures. Given $z' < z$, we modify in Section 9.2 the execution frequencies in those histories, such that they reflect the run time behavior of z' in t.

For example, if for z a sequence of queries $q_1; q_2$ is always executed at the cache the corresponding nodes in the execution history are labelled by the counter, say $c_{invalid} > 0$, which reflects the number of invalid executions of the queries at the cache. If in z' the query q_1 is *not* executed at the cache ($z' < z$ performs less IO statements at the cache than z), $c_{invalid}(q_1) = 0$ follows and the corresponding counters for q_2 in the execution history will change, since the server does not switch to normal mode for an invalid result of q_1 and therefore considers more delivered results of q_2.

II. Determine Load-Dependent Basic Parameters

In Section 8.4.1 we have defined the basic parameters for the model of the simultaneous execution scheme, e.g., execution time of IO statements, verification time, etc. These parameters depend on the load conditions at a cache and the central server. The load of the system depends on the users' activities and the split parameters which balance the code execution between caches and the server.

Since a new configuration z' can significantly change the code balancing between cache and server, we first determine the load that is caused by z' and second determine the appropriate value of the load-dependent basic parameters. In Section 9.3 we present an appropriate approach.

III. Calculate the Execution-Performance

For the modified basic parameters and the modified execution histories, we calculate the average processing time for each procedure by Algorithm 8.2, $PT(.)$. Given $x_t = (z, repl)$ we show in Section 9.4 that for $x'_t = (z', repl)$ with $x'_t < x_t$ the resulting execution-performance of the cache in t run with x'_t follows from

$$\sum_{i=1}^{k} mPT_t(S_i) \quad \approx \quad \sum_{i=1}^{k} PT(z'.S_i, hist_t^i(S_i)). \tag{9.3}$$

with $hist_t^i(.)$ as the modified histories that have been observed for $x_t = (z, repl)$.

As a result, the optimizer at the cache can calculate the execution-performance for a subset of the configuration space $\mathcal{Z}_t \times 2^{\mathcal{F}_t}$, and thus can select a configuration z_{opt} with the best performance for a completed stage t. Then, z_{opt} is a possible candidate for the configuration of the next stage.

In Section 9.5 we apply a greedy search technique to the configuration space. A summary of our model and a discussion is given in Section 9.6. Additionally, we point to some open problems and possible extensions that may further improve our technique.

9.2 Simulate the Execution Behavior of Split Parameter

The execution history $hist_t(S)$ of a procedure S is recorded for stage t and $split \in Z(S)$. We modify the history in such a way that it simulates the behavior of a $split' \subset split$. Hence, the modified history represents the run time behavior of S in t as it would be observed if $split'$ has been used instead of $split$.

First, we present some motivating examples that demonstrate the main idea of the modification. Then, we define an appropriate algorithm for adapting an execution history.

9.2.1 Motivating Examples

Given $split$ and $split'$ we present the basic idea of modifying the history in Figure 9.1. The

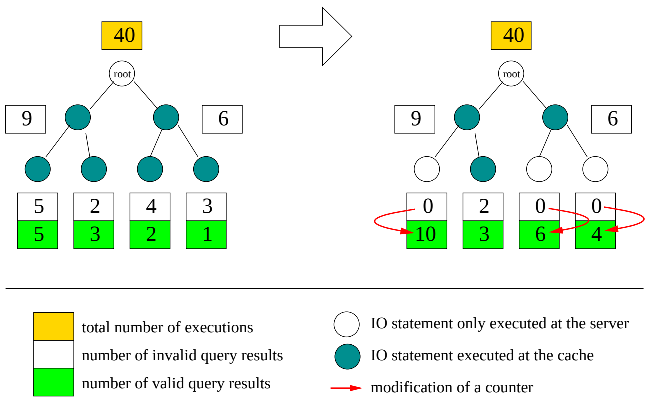

40
root
9
6
5
2
4
3
5
3
2
1
40
root
9
6
0
2
0
0
10
3
6
4
total number of executions
number of invalid query results
number of valid query results
IO statement only executed at the server
IO statement executed at the cache
modification of a counter

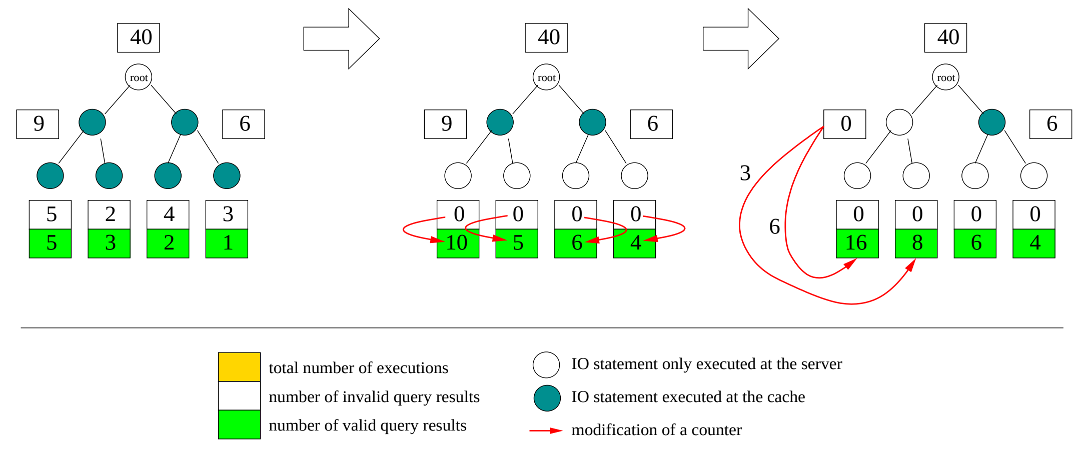
Figure 9.1: Example for modifying the execution frequencies of a history. Left: All statements are executed at the cache. Right: Three statements are removed from the parameter $split$.

left side represents the execution history $hist_t(S)$ where grey nodes represent IO statements in $split$. The box at the root r node depicts the total number $c_{exec}(r)$ of executions of S in t, white boxes at non-root nodes n the number $c_{invalid}(n)$ of valid subsequences with an invalid query n. Leaf nodes n are not attached to the tree, but are represented by the green boxes with the counter $c_{valid}(n)$. The counter $c_{valid}(n)$ represents the number of valid subsequences, where all query results are valid. Recall that each execution of S must result into one valid subsequence. Therefore the sum of all $c_{invalid}$ and c_{valid} must be 40.

The right side represents $split'$ and the modified history. There are three white nodes that represent queries that are not executed at the cache any more. Hence $split'$ contains 3 queries less than $split$. Let n be such a node. If n is not executed at the cache, then there can be no valid subsequence with n as an invalid query at the end. Hence, $c_{invalid}(n)$ must be 0 (white boxes with 0).

Furthermore, the counter $c_{valid}(n)$ at leaf nodes n represents all executions of the path to n where all query results are valid. If n is not in split, then the number $c_{invalid}(n)$ represent the number of executions of the path to n where all queries are valid. Hence, we add $c_{invalid}(n)$ to the counter $c_{valid}(n')$ of the corresponding leaf node n' that is attached to n. Recall that n' is always attached to exactly one leaf node (see Section 8.3). As a result, the sum of all $c_{invalid}$ and c_{valid} counter is 40 and the counters simulate a history for $split'$.

The next example in Figure 9.2 discusses a node that is removed from $split$ and that is connected to multiple leaf nodes. The left side again represents $split$ and the picture in the middle the modified history for removing the lower nodes from $split$ according to Figure 9.1.



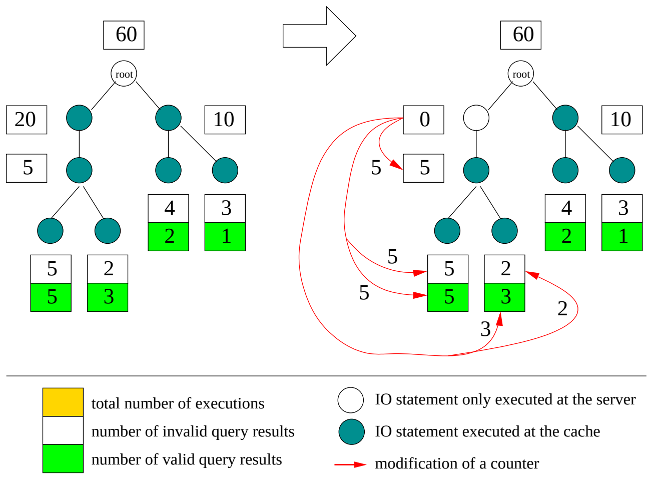
40
root
9
6
5
2
4
3
5
3
2
1
0
10
5
6
4
3
0
16
8
total number of executions
number of invalid query results
number of valid query results
IO statement only executed at the server
IO statement executed at the cache
modification of a counter

Figure 9.2: Example for modifying the execution frequencies of a history. Left: All statements are executed at the cache. Middle: Four statements are removed from the parameter $split$. Right: One statement with subsequent statements is removed from the parameter $split$.

The history at the right shows how we handle the removal of a node that is connected to multiple leaf nodes. Let n be this node. According to the Figure the valid subsequence with n as an invalid query is executed $c_{invalid} = 9$ times. Hence, for $split'$ these cases cannot occur any more. Again, we add $c_{invalid}(n)$ to $c_{valid}(n')$ of the leaf nodes n' that are connected to n.

However, the problem still is how to add the value of $c_{invalid}(n)$ to $c_{valid}(n')$ of two leaf nodes. Since the precise distribution is unknown, we distribute $c_{invalid}(n)$ proportional to the execution frequency $c_{valid}(n')$ of both leaf nodes. Again, the sum of all $c_{invalid}$ and c_{valid} counters is 40 and the counters simulate a history for $split'$.

The above two examples deal with queries that are removed from the end of execution paths. At Figure 9.3 we take a look at a query which is removed from the middle of an execution path, hence reflects an independent query q, such that all query executions after q can be executed independently of q. The left side shows the execution history for *split*. The procedure has



60
root
20
10
5
4
2
3
1
5
5
2
3
60
root
0
10
5
5
4
2
3
1
5
5
5
5
2
3
2
3
total number of executions
number of invalid query results
number of valid query results
IO statement only executed at the server
IO statement executed at the cache
modification of a counter

Figure 9.3: Example for modifying the execution frequencies of a history.

been executed 60 times.

At the right side we show the execution history for $split' \subset split$ that performs one query less at the cache. Let n be this query. Originally $c_{invalid}(n) = 20$ which means that the first query on the involved execution paths was invalid in 20 executions out of 40 (sum of the left subtree is 40, sum of the right subtree is 20).

If n is not executed at the cache, these 20 cases cannot occur. As a consequence the server does not switch to normal mode and expects more query results from the cache. Hence, these 20 executions distribute among the 5 valid subsequences that correspond to the 3 subnodes of n and the 2 leaf nodes (represented by the green boxes).

Since the precise distribution among these subsequences is unknown, we assume that these 20 executions are equally distributed according to the execution frequencies $c_{invalid}$ and c_{valid}. Figure 9.3 shows how the 20 executions are distributed. Again, the sum of all $c_{invalid}$ and c_{valid} counter is 60 and the counters simulate a history for $split'$.

A special cases arrises if the sum of the counters $c_{valid}(n')$ and $c_{valid}(n')$ is 0 for all subnodes of n. Then, at run time none of these subsequences has been executed. The problem is to distribute the value of $c_{invalid}(n)$ to these nodes. In such a case we distribute the value according to the counters $c_{exec}(n')$ that is greater 0, since the history contains only execution paths that have been executed in a stage.

9.2.2 An Algorithm for Modifying the Execution History

Given a history $hist_t(S)$ and two splits with $split' \subset split$, the following Algorithm returns the modified history $hist'_t(S)$. Its evaluation is supplied in Section 9.4.

Algorithm 9.1 (Modify Execution History)

```
Syntax:   $computeHist(split, split', hist_t(S))$
Input:    split parameter, execution history
Output:   modified execution history $hist'_t(S)$

1. Let $N = \{n_1, \dots, n_k\}$ be the node set of $hist_t(S)$ where $n_i = (s_i, \#s_i)$ with $s_i \in ID(S)$.
   Let $N_{modify} \subseteq N$ be the subset of all nodes with $s_i \in split$ and $s_i \notin split'$.
   // $N_{modify}$ does not contain leaf nodes nor the root node.

2. For each node $n \in N_{modify}$ do

   (a) // Handle low-level updates.
       // There is nothing to do for delete operations.
       // For insert operations the counter for executions at the cache is set to 0.
       If $n$ corresponds to an $insert(R, t)$ operation then set $c_{insert}(n) = 0$ endif
   (b) // Handle queries.
       If $n$ corresponds to a query then
       i.   Set $N_{sub}$ as all subnodes that follow on execution paths after $n$ and that
            correspond to a query or a leaf node. Note that $n \notin N_{sub}$.
       ii.  // Compute sum of valid subsequences over all subnodes.
            $\#exec := \sum_{n' \in N_{sub}} c_{invalid}(n') + c_{valid}(n')$
       iii. // Add $c_{invalid}(n)$ proportional to all subnodes and
            // modify counters within the history $hist_t(S)$.
            If $\#exec > 0$ then for each node $n' \in N_{sub}$:
            A. $c_{valid}(n') := c_{valid}(n') + \frac{c_{valid}(n')}{\#exec} \cdot c_{invalid}(n)$
            B. $c_{invalid}(n') := c_{invalid}(n') + \frac{c_{invalid}(n')}{\#exec} \cdot c_{invalid}(n)$
            endif (Step 2.b.iii)
       iv.  // If sum is 0 then none of the valid subsequences represented by the node
            // $N_{sub}$ has been executed in stage $t$. For this, we apply a distribution
            // according to the total execution frequency $c_{exec}$ of a query.
            If $\#exec = 0$ then
            A. $\#exec := \sum_{n' \in N_{sub}} c_{exec}(n')$
            B. // Modify counters within the history $hist_t(S)$.
               For each node $n' \in N_{sub}$:
                   1. $c_{valid}(n') = c_{valid}(n') + \frac{c_{exec}(n')}{\#exec} \cdot c_{invalid}(n)$
```

```
                    2. $c_{invalid}(n') = c_{invalid}(n') + \frac{c_{exec}(n')}{\#exec} \cdot c_{invalid}(n)$
                 endfor
              endif (Step 2.b.iv)
          v. // If $n \notin split'$ no valid subsequence can end in $n$.
              $c_{invalid}(n) = 0$
          vi. // If $n \notin split'$ there can be no reused results for $n$.
              $c_{reuse}(n) = 0$
          endif (Step 2.a)

    endfor (Step 2)

 3. Return $hist_t(S)$
```

As explained in previous sections, an execution of a procedure must result into one of the valid subsequences as captured by the counters $c_{invalid}$ and c_{valid} of the execution history. Recall that $c_{exec}(r)$ of the root node r of a tree represents the number of all executions in a stage. Hence, the sum over all $c_{invalid}$ and c_{valid} of the nodes of the tree has to be $c_{exec}(r)$. From the above algorithm we require that it preserves this property.

Given a node n of the tree and all of its subnodes N_{sub}, both represent a subtree of the execution history. We show that the sum over all $c_{invalid}$ and c_{valid} of this subtree is not modified by the Algorithm. If this holds for all subtrees, it also holds for the whole tree.

For the case $\#exec > 0$ the sum over all $c_{invalid}$ and c_{valid} of a subtree n is

$$c_{invalid}(n) + \sum_{n' \in N_{sub}} \Big(c_{invalid}(n') + c_{valid}(n') \Big)$$

The following transformations exactly represent Step *2(b)iii* of the Algorithm

$$\begin{aligned}
& c_{invalid}(n) + \sum_{n' \in N_{sub}} \Big(c_{invalid}(n') + c_{valid}(n') \Big) \\
= \; & \frac{\#exec}{\#exec} \cdot c_{invalid}(n) + \sum_{n' \in N_{sub}} \Big(c_{invalid}(n') + c_{valid}(n') \Big) \\
= \; & \frac{c_{invalid}(n)}{\#exec} \cdot \Big(\sum_{n' \in N_{sub}} c_{invalid}(n') + c_{valid}(n') \Big) + \sum_{n' \in N_{sub}} \Big(c_{invalid}(n') + c_{valid}(n') \Big) \\
= \; & \sum_{n' \in N_{sub}} \left(c_{invalid}(n') + \frac{c_{invalid}(n')}{\#exec} \cdot c_{invalid}(n) + c_{valid}(n') + \frac{c_{valid}(n')}{\#exec} \cdot c_{invalid}(n) \right)
\end{aligned}$$

and shows that the sum is preserved.

For the case $\#exec = 0$ the sum of a subtree n is also

$$c_{invalid}(n) + \sum_{n' \in N_{sub}} \Big(c_{invalid}(n') + c_{valid}(n') \Big)$$

where the sum over the counters $c_{invalid}$ and c_{valid} is 0. The following transformations represent Step *2(b)iv*

$$
\begin{aligned}
& c_{invalid}(n) + \sum_{n' \in N_{sub}} \Big(c_{invalid}(n') + c_{valid}(n') \Big) \\
= \; & \frac{\#exec}{\#exec} \cdot c_{invalid}(n) + \sum_{n' \in N_{sub}} \Big(c_{invalid}(n') + c_{valid}(n') \Big) \\
= \; & \frac{c_{invalid}(n)}{\#exec} \cdot \sum_{n' \in N_{sub}} c_{exec}(n') + \sum_{n' \in N_{sub}} \Big(c_{invalid}(n') + c_{valid}(n') \Big) \\
= \; & \sum_{n' \in N_{sub}} \frac{c_{exec}(n')}{\#exec} \cdot c_{invalid}(n) + \sum_{n' \in N_{sub}} \Big(c_{invalid}(n') + c_{valid}(n') \Big) \\
= \; & \sum_{n' \in N_{sub}} \left(c_{invalid}(n') + \frac{c_{exec}(n')}{\#exec} \cdot c_{invalid}(n) + c_{valid}(n') + \frac{c_{exec}(n')}{\#exec} \cdot c_{invalid}(n) \right)
\end{aligned}
$$

and again shows that the sum is preserved. Hence, the sum over all $c_{invalid}$ and c_{valid} of the nodes of the tree is not changed by the Algorithm.

9.3 Capture Load-Dependent Basic Parameters

The higher the load at a cache or the server, the higher the execution time of queries and low-level updates. Based on this observation, we present a model that roughly captures this correspondence. It is suitable for all systems that utilize database caches and the central server only for the running database application. Hence, the model does not support other applications that have an impact on the load of caches and the server. Extentions of the model are discussed at the end of this Chapter in Section 8.6.

Caches and the server perform stored procedure code that consists of IO statements (queries and low-level updates). Hence, the load at a cache and the server can be captured by the total amount of IO statements that are executed in a stage t. The amount of IO statements corresponds to the number of calls of the function $eval(.)$ that executes such a statement on data. We do not consider the functions $eval_C(.)$ and $eval_S(.)$, since they do not necessarily execute a statement, e.g., $undef$ values, reuse of a delivered result, etc.

We add two more measures at a cache: $mIOC_t$ represents the total number of all in a stage t performed IO statements at the cache and $mIOS_t$ those at the server. Note that $mIOS_t$ is determined by the configuration of the split parameters of *all* attached database caches, since these define the load balancing between caches and the server. The value of $mIOS_t$ is obtained by the server and send to all attached caches at the end of a stage. Hence, a cache is informed about the load at the server.

Consider again the configuration $z = (split_1, \dots, split_k)$ of the cache at stage t and the new configuration $z' = (split'_1, \dots, split'_k)$ for which we have modified the execution histories $hist^1_t(S_1), \dots, hist^k_t(S_k)$ to simulate its run time behavior. Since a configuration z' defines the code execution at cache and server, it can also affect the load conditions of both. Hence, z'

can also cause a change of the basic parameters, e.g., query execution time, that are required to predict the processing time of procedures. Based on the histories, we first determine the load of the cache and the server, and second determine the value of these load-dependent parameters.

9.3.1 Determine the Load of a Configuration

To define $mIOC_t$ and $mIOS_t$, we introduce some intermediate notations that start with a # to denote their counter-like nature. Given a procedure S_i, $split_i \in Z(S_i)$ and its execution history $hist_t^i(S)$, we compute the load of S_i at the server as follows: Let N_{update} be the set of all nodes in $hist_t^i(S_i)$ that represent low-level updates and N_{query} those that represent queries. Then,

$$\#IOS_t(hist_t^i(S_i)) \quad = \quad \sum_{n \in N_{update}} c_{exec}(n) + \sum_{n \in N_{query}} \Big(c_{exec}(n) - c_{reuse}(n) \Big)$$

captures the load of S_i by summarizing the number of executions of IO statements at the server. A low-level update is always executed at the server. For query executions we subtract the amount of valid queries whose result was delivered by the cache and not executed by the server. Note that Algorithm 9.1 sets $c_{reuse} = 0$ for all queries that are not executed by the splits in z' at the cache. Hence, the value $\#IOS_t$ can be computed for the original, as well as for the modified histories.

For all procedures $S_1, \ldots, S_k$ that are executed at the cache in t we calculate the portion of the load that is imposed on the server. The *cache-server* load is defined by

$$\#IOCS_t \quad = \quad \sum_{i=1}^{k} \#IOS_t(hist_t^i(S_i))$$

Note that $\#IOCS_t$ is only equal to the total load $mIOS_t$ of the server if only one cache is attached to the server. Then, other caches do not produce load at the server and only the procedures $S_1, \ldots, S_k$ are executed at the server. Otherwise $mIOS_t$ equal to the sum of the $\$IOCS_t$ of all attached database caches.

Let $mIOS_t$ be the observed total load at the server, $\#IOCS_t$ the calculated cache-server load from the original execution histories and $\#IOCS'_t$ the cache-server load of the modified execution histories. Then, the resulting total load $mIOS'_t$ at the server for z' is approximated by

$$mIOS'_t \quad = \quad mIOS_t - \#IOCS_t + \#IOCS'_t$$

which is based on the original load of t and the change of the cache-server load. Note that this prediction ignores the existence of other database caches. However, from the point of view of a single cache the behavior of other caches is unknown, such that no better prediction than $mIOS'_t$ can be made, except that caches communicate with each other (see also Section 7.6) and synchronize their optimization and prediction.

The load $mIOC_t$ at the cache can be captured in a similar way, however not by using our specific notion of an execution history which considers IO statements at the cache only up to the last notification (see Section 8.2.2). To capture $mIOC_t$, we have to define a separate history for the partial execution at the cache which can be constructed in a similar way as the current history.

Since we have not done this, we apply the proposed prediction technique only to cache that runs in single-user mode, such that there is no concurrent access at the cache. As a consequence the database system always processes one request at a time and we expect constant values of the basic parameters. We show this later on in this section.

9.3.2 Determine the Value of Load-Dependent Basic Parameters

According to Section 8.3.2 the basic parameters are (1) the execution time of an IO statement s of a procedure S at the cache $pt_C(s)$ and at the server $pt_S(s)$, (2) the access time of the query result cache (QRC) t_{cache}, (3) the network time t_{net}, (4) the verification time $t_{verifyC}$ at the cache and $t_{verifyS}$ at the server, and (5) the final commit and additional overhead of an execution.

Given the parameter pt_S and the total server load $mIOS_t$, we capture the load-dependent value of $pt_S(s)$ by a function

$$f_{ptS} : \mathcal{S} \times \mathbf{N} \times \mathbf{R} \mapsto \mathbf{R}$$

such that $pt_S = f_{ptS}(S, s, mIOS_t)$.

We assume that this function is maintained by a database cache. For each completed stage a cache has obtained the values $mIOS_t$ and $pt_S(s)$ from the server, hence over multiple stages a cache can be trained to capture the function $f_{ptS}(.)$ Analogously, the parameter $pt_C(s)$ can be handled if the load $mIOC_t$ at the cache has been captured appropriately.

For the remaining parameters, e.g., $t_{verifyS}$, a function

$$f_{verifyS} : \mathbf{R} \mapsto \mathbf{R}$$

is sufficient, since the parameter does not depend on a procedure nor its IO statements. Then, $t_{verifyS} = f_{verifyS}(mIOS_t)$. Analogously, the values for these parameters are known at a cache, such that an appropriate function can be trained at run time.

An Example for f_{ptS}

For the ONE-System (Experiment of Section 6.3.2) with one procedure S and one type of query, say s, we have observed the values x for $f_{ptS}(S, s, mIOS_t)$ for a stage of length 80s with $0 \leq x \leq 16000$ as depicted by the scatter diagram in Figure 9.4. The dots represent all pairs of $mIOS_t$ and $pt_S(s)$ that have been observed during the experiment. The line shows the resulting regression function $f_{ptS}(S, s, mIOS_t)$ that represents the average value of $pt_S(s)$ for a given $mIOS_t$.

It shows that the average query execution time increases from 5ms to 15ms at the server. The maximum of 16000 IO statements results from the following observation: a procedure

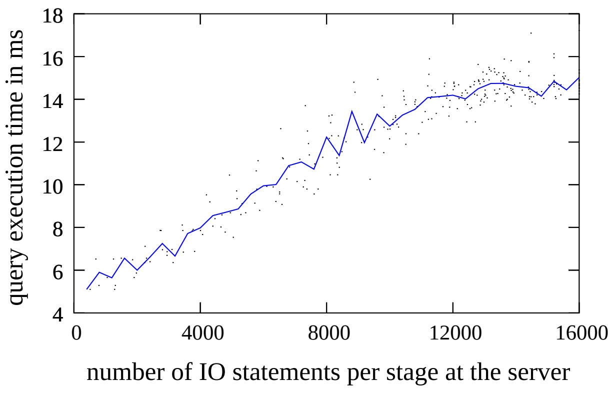


Figure 9.4: Query execution time pt_S and server load $mIOS_t$ for a four-client system.

call performs at most 20 queries at the server. For those configurations we obtain a maximal throughput of about 200 procedure calls in 80 seconds (see Chapter 6). In total these are about $200 \cdot 20 = 4000$ queries for one client and 16000 queries for four clients. Taking the updates at the server (1 per second) into account, we get about 16080 IO statements per second.

Analogously we observed $f_{ptC}(S, s, mIOC_t)$ for a cache. Figure 9.5 shows the query execution time at the cache w.r.t to the load. Again, the dots represent the measured values

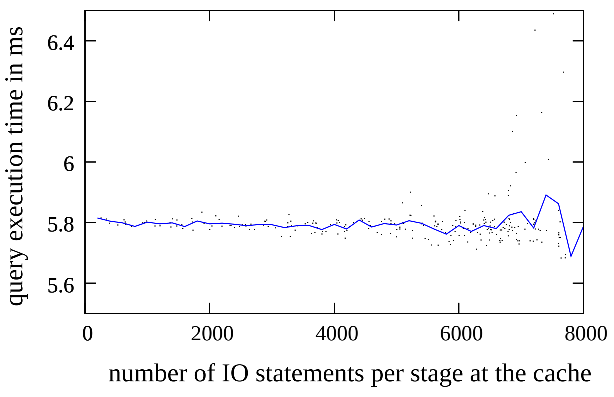


Figure 9.5: Query execution time pt_C and cache load $mIOC_t$ for the four-client ONE-System.

at run time and the line the trained function.

Since a cache performs procedures in a sequential manner and only one query can occupy the computational resources of the system, the query execution time is almost constant. The maximum of 8000 IO statements per stage results from the following observation: the maximum throughput is 400 procedure calls in a stage of 80 seconds. This maximum is achieved by executing all queries at the cache. Hence, in a stage the cache executes $400 \cdot 20 = 8000$ queries.

Since the cache runs in single-user mode, it performs 8000 queries in a stage much more efficiently than the server. The overhead at the server results from the concurrent access of four clients and the intermediate updates.

9.4 Evaluation of the Prediction Technique

For the experiments with four clients of the Sections 6.3.2 and 6.3.3 we have evaluated the predication technique. Both experiments measure the processing time of a procedure for 21 different split and 13 different synchronization configurations. The split balances the code execution between cache and server, and the different synchronization rates determine the freshness of cache data, thus determining the amount of valid and invalid queries. Recall that the procedure S implements 20 equal queries. In the following we use s for this query. We apply the technique as defined in Section 9.1.2 and show that our technique produces reasonable results in predicting the execution-performance of a cache.

Dependent Queries

After the experiment completes, we obtain 273 execution histories, one for each of the $21 \cdot 13 = 273$ configurations. Let us, for example, consider those configurations that perform 19 queries at the cache and one query at the server, hence $split = \{1, \ldots, 19\}$. Then, there are 13 execution histories $hist_1, \ldots, hist_{13}$, one for each configuration of the synchronization.

For each of the 13 histories we perform the prediction technique for the splits $\emptyset$, $\{1, \ldots, i\}$ with $1 \leq i \leq 18$ — in total 19 different split configurations, denoted $split_1, \ldots, split_{19}$. Let $hist_i$ be one of the 13 histories and $split_j$ one of the 19 splits. Then, we apply the prediction technique with the above defined algorithms as follows:

1. // Modify the history to simulate $split_j$.
   $hist'_i = computeHist(split_j, hist_i)$
2. Let $mIOS$ be the original observed server load of the configuration 19 and i for $1 \leq i \leq 13$
   // Compute the cache-server load $\#IOCS$ for the original and the modified histories.
   // Since only a single procedure is executed in a stage, only one history is considered
   // for the computation of $\#IOCS$ and $\#IOCS'$.
   $\#IOCS = \#IOS(hist_i)$
   $\#IOCS' = \#IOS(hist'_i)$
   // Approximate the server load of the new configuration.
   // The factor four results from the four attached database caches.
   // Note that all attached caches use the same configuration for a stage.
   // Hence, the change of the server load is equal for all 4 caches.
   $mIOS' = mIOS - 4 \cdot \#IOCS + 4 \cdot \#IOCS'$
3. // Based on the trained functions and the server load,
   // determine the query execution time at the server.
   $pt_S(s) = f_{ptS}(S, s, mIOS')$

4. // Determine the remainding basic parameters.
   // $pt_C(s)$ is static for a cache in single-user mode (see also Figure 9.5).
   $pt_C(s) = 5.8$
   // As shown by the experiments, the value of these parameters is very low
   // and therefore have only a minor impact on the prediction technique.
   $t_{cache} = t_{net} = t_{verifyC} = t_{VerifyS} = 0$ for all configurations
   // For all experiments the additional overhead is about 2-4ms.
   $t_{final} = 3$

5. // Predict processing time.
   $PT_{i,j} = PT(split_j, hist_i)$

For the $19 \cdot 13$ configurations Figure 9.6 depicts the original observed and the predicted

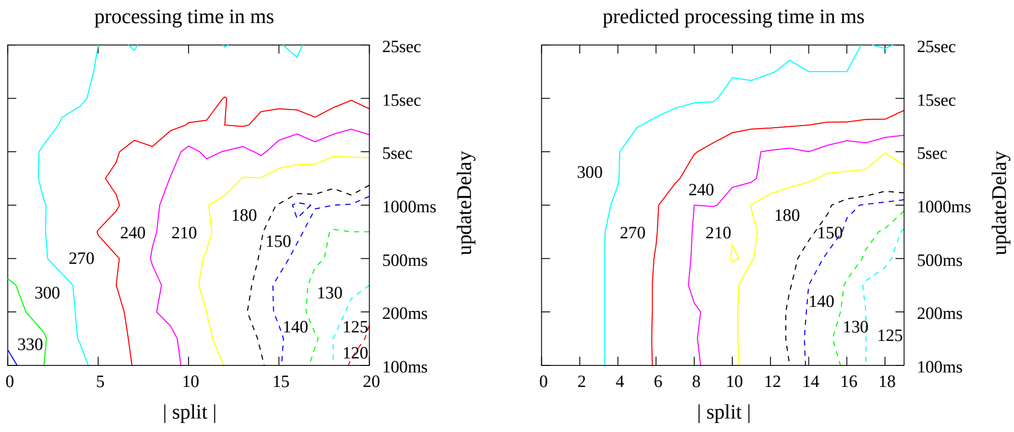


Figure 9.6: Observed (left) and predicted (right) processing time with four clients and dependent queries.

processing time $PT_{i,j}$. Again the X-axis denotes the split configuration, hence the amount of queries that are intended to be executed at the cache, and the Y-axis represents the 13 different synchronization delays. As we observe, there is only a little difference between the original and the predicted processing time.

In the following we take a closer look at the deviation between the original observed and the predicted values. For this, we have performed the prediction also for all other split configurations, not only this with 19 queries. Hence, we obtained

$$\sum_{k=1}^{20} k \cdot 13 \quad = \quad 2730$$

pairs (obs_i, est_i) where obs_i represents an observed value and est_i an predicted value. We skipped the empty split (case $k = 0$) since for no query computations at a cache there is nothing to predict. Again, we visualize the deviation in terms of the frequency distribution and density (see Section 8.5 for a definition). According to Figure 9.7 we obtain a deviation of -7% up to 7% for 90% of the pairs.

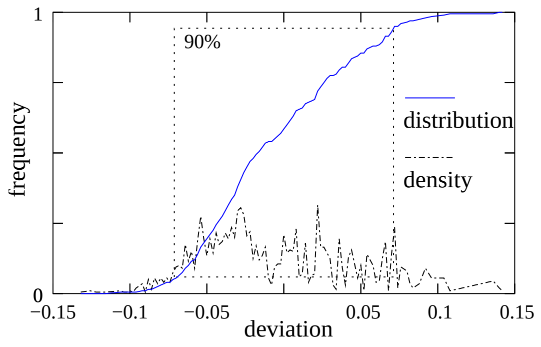


Figure 9.7: Preciseness of the prediction technique of relative deviations (frequency density and distribution) between the original observed and the predicted values for dependent queries.

Independent Queries

The same experiment is repeated for independent queries. The only difference are the split configurations. Instead of the above, we use $split_i = \{i, \ldots, 20\}$ for $2 \leq i \leq 20$ which are again 19 different split configurations. Hence, the cache jumps over the first $i-1$ queries in the procedure code.

Figure 9.8 depicts the original observed and the predicted processing time. Again there

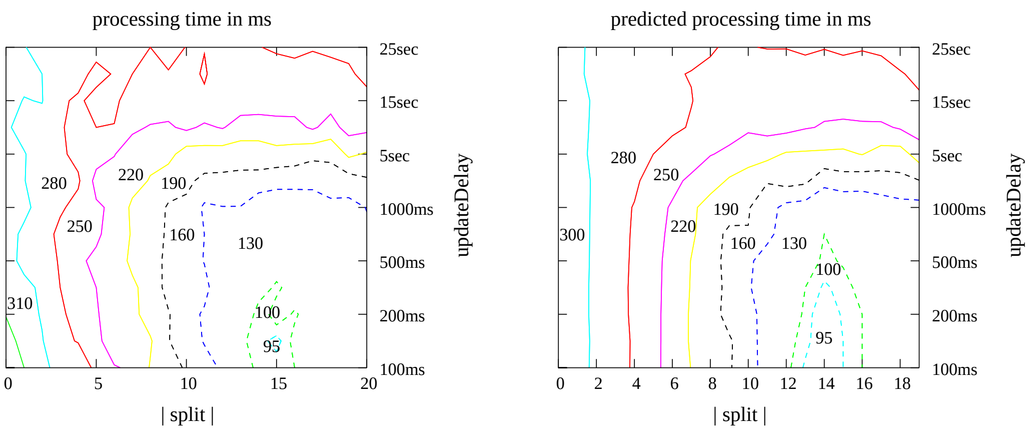


Figure 9.8: Observed (left) and predicted (right) processing time with four clients and independent queries.

is only a little difference. The optimum for 15 queries at the cache shows the usefulness of our prediction technique. If, for example, a cache executes the configuration with 20 queries, a better configuration can be detected by predicting the execution-performance of all configurations that perform less executions at the cache, but more at the server. Such an analysis would reveal the configuration with 15 queries at the cache as optimum.

According to Figure 9.9 we obtain a deviation of -10% up to 8% for 90% of the pairs.

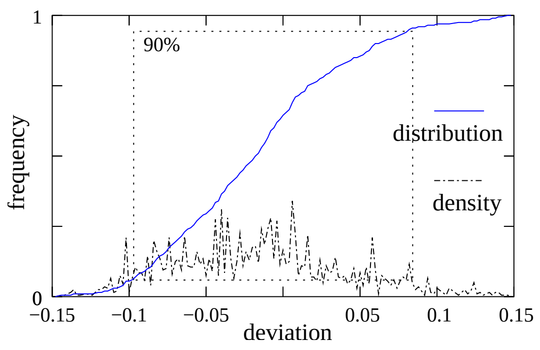


Figure 9.9: Preciseness of the prediction technique of relative deviations (frequency density and distribution) between the original observed and the predicted values for independent queries.

9.5 Towards Solving the Dynamic Optimization Problem

In this Section we present a greedy algorithm as a primitive solution for the dynamic optimization problem. Based on the experimental results of the ONE-System, we first discuss the values of the user-provided parameters *minRead*, *minWrite* and *minEff* that are used to parameterise the optimization problem (see Section 7.3). Second, we present and discuss the algorithm that incorporates the above technique for predicting the cache-performance of a cache.

9.5.1 The User-Provided Parameters of the Optimization Problem

According to Section 7.4, a fragment F is only replicated to a cache in stage t', if it suffices the following properties already in the previous stage t:

$$\begin{aligned} mRead_t(F) &\geq minRead \\ mWrite_t(F) &\leq maxWrite \\ mEff_t(F) &\geq minEff \end{aligned}$$

The first represents the read-frequency of F (number of read operations per time unit), the second the write-frequency (number of write operations per time unit), and the latter the efficiency of a fragment that is defined by the total execution time of valid queries at the cache in stage t. The values *minRead*, *maxWrite* and *minEff* are provided by the administrator of the database system. In Chapter 6 we have evaluated the simultaneous execution for the ONE-System. Based on this, we discuss possible values of the above parameters. However, in general the proper choice of these parameters depends on the application, e.g., number of fragments, network speed, computational power of the clients and the central server, etc. To obtain these values for an application, a similar analysis as in Chapter 6 should be performed.

Write Frequency

As defined in Section 4.5, the proposed database system uses an optimistic update propagation protocol to synchronize cache data. Hence, updates that have been applied at the server are propagated to the cache with a possible delay. The delay causes data inconsistencies at the cache, such that query executions at the cache can produce invalid results. The higher the frequency of updates at the server, the lower the probability of a query at the cache to hit a consistent data version. For this, only the replication of less-frequently fragments is useful. Further, high-frequently updated fragments cause a high synchronization effort at the server.

For the ONE-System in Chapter 6 we have chosen a value

$$maxWrite \quad = \quad 1$$

such that only those fragments are replicated that are updated less than once per second. This is feasible, since there is only one fragment. For applications with a high number of frequently updated fragments, the value $maxWrite = 1$ is probably to high, since for example 1000 of such fragments will cause 1000 updates per second at a cache. If these updates overload the cache, the simultaneous execution does not necessarily benefit from local computations. Hence, the value of $maxWrite$ should be chosen this way that a reasonable amount of updates per second are applied at a cache.

Read Frequency

Our scheme only participates from valid query executions at a cache. The fewer queries are executed on a fragment at the server, the fewer potential valid queries at a cache. For the experiments in Chapter 6 we obtained a read frequency of a fragment of 50 up to 100 per second. These values result from

$$50 \leq \frac{x \cdot 20\ queries}{80\ sec} \leq 100$$

where 80 is the length of a stage, 20 the number of query executions within a procedure call and $x \in [200, 400]$ the number of procedure calls in a stage (throughput). Recall that the throughput depends on the chosen split parameter.

However, in general the parameter $minRead$ is used to filter out fragments that are rarely accessed, since there is no need to replicated fragments that are not accessed by an application. For this, a setting of

$$minRead \quad > \quad 0$$

should be sufficient to block such fragments. Note that $minWrite$ and $minRead$ are only used as entry barrier for the replication of a fragment F, but there is no guaranty that the cache-performance will benefit from the replication of F. For this we also consider the fragment efficiency.

Fragment Efficiency

A low fragment efficiency indicates a low number of valid query executions at the cache. The fragment efficiency of the ONE-System is depicted at Figure 9.10. The X-axis represents the number of query executions at the cache and the Y-axis the update delay which influences the error rate. The higher the delay, the higher the error rate and the more queries have to be re-executed at the server. The re-execution is costly and increases the amount of query executions at the server. Recall that the ONE-System uses only one fragment. Hence, all 20

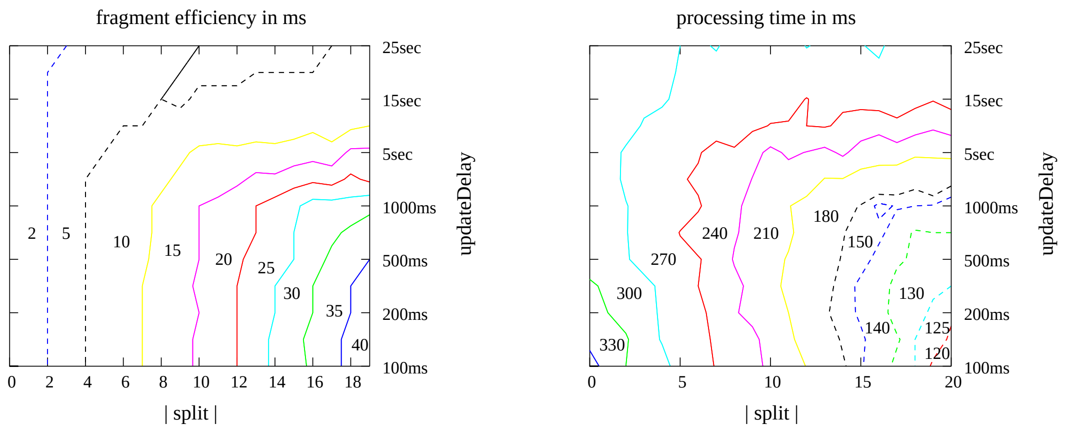


Figure 9.10: ONE-System. Left: fragment efficiency in seconds. Right: processing time of a procedure in ms.

queries of a procedure are executed on the same fragment.

As expected, the higher the number of valid queries, the higher the fragment efficiency which is the total sum in seconds of valid query executions at the cache. According to the Figure, a setting of

$$minEff = 2$$

for the ONE-System is feasible, since then no significant improvement is observable for the processing time.

However, the value of *minEff* is based on the query execution time and the error rate which again depends on the application and its stored procedures. For this, a proper selection of *minEff* requires an analysis of the application's run time behavior.

9.5.2 A 2x2-Phase Greedy Optimization Algorithm

In the following we demonstrate an example of an optimization algorithm that incorporates the above prediction technique to solve the dynamic optimization problem. However, the algorithm does not provide a solution for all concerns of a dynamic optimization problem and relies on the following assumptions and restrictions:

1. The algorithm requires a 24-hour cycle of an application where at night time no significant executions take place. Hence, at night time there are free resources that can be utilized by the optimizer.

2. Since the initial replication of fragments to a cache can result into heavy work load, the placement of new fragments and the removement of existing fragments of a cache is only done at night time.

3. A cache has been trained to provide the functions f_x with $x \in \{pt_C, pt_S, \mathit{verify}_S, \dots\}$ (see Section 9.3.2).

4. The algorithm neither detects repetitive pattern in the system's behavior, nor does it utilize a memory to keep good-performing configurations of previous stages.

5. In each stage the system creates a new execution history for a procedure. There is a garbage collector that removes histories that are not used any more.

6. The algorithm performs a random search over the configuration space $\mathscr{Z}$ which represents the split parameters of all procedures.

The algorithm runs in different phases. Over a 24-hour cycle there are the two main phases. The operational phase at day time tries to figure out a configuration that results in an optimal performance for the given work load. The data-reconfiguration phase takes place at night time and, as mentioned above, adds/removes fragments to/from a cache.

The operational phase at run time continously toggles two types of stages. The first pushes the execution of all IO statements to a cache and the second uses an optimal configuration that is computed by the above prediction technique. The goal of the first type of stages is to execute as much as possible (greedy) of the queries at the cache, such that the resulting execution histories contain the execution frequency for a high number of valid subsequences. On such histories we run the presented predication technique and compute a configuration z_{opt} with the best execution-performance, which then is used for the second type of stages. Hence, in the best case every second stage runs with an optimal configuration.

However, the algorithm has also to consider the computation effort to compute z_{opt} at run time. Instead of computing z_{opt} between two successive stages, we shift its computation into background. Thus, a computation can run multiple stages and as soon as z_{opt} has been computed it can be used by a stage of the second type.

The algorithm uses the notation $mRead_{\varnothing}(F)$ that represent the arithmetic average of $mRead_t(F)$ over all stages t of the operational phase. Let $t_1, \dots, t_k$ be all stages during the operation phase, then the average read frequency of a fragment F is defined by

$$mRead_{\varnothing}(F) \quad = \quad \frac{1}{k} \cdot \sum_{i=1}^{k} mRead_{t_i}(F)$$

Analogously, we use $mWrite_{\varnothing}(F)$ and $mEff_{\varnothing}(F)$.

Algorithm 9.2 (Greedy Optimization Algorithm at a Cache)

```
Input:            m - number of configurations for predicting the cache-performance,
                  minEff, minRead, minWrite
Output:           set internal configuration parameters for each stage
Local Variables:  repl - set of replicated fragments, $z_{max}$, $z_{opt}$ and $z_i$ - split configurations,
                  F - fragment, $i, k$ - running index, t - stage, j - stage counter

1. // No fragments are replicated. Set first stage.
   $repl = \emptyset$, $j = 0$
2. // Operational phase. We assume that at midnight there are no user
   // activities, such that the system can perform reconfiguration tasks.
   Repeat until midnight
   (a) // Set configuration, where all IO statements are executed at the cache.
       $z_{max} = (ID(S_1), \dots, ID(S_k))$ for all $S_i \in \mathcal{S}_{t_j}$ with $1 \leq i \leq k$
   (b) // Shift maximum of computations to the cache.
       $j = j + 1$
       Run stage $t_j$ with configuration $(z_{max}, repl)$
   (c) If there is no computation in background and $repl \neq \emptyset$, do in background:
       i. Pick randomly $z_1, \dots, z_k$ with $z_i < z_{max}$ or $z_i = z_{max}$ for all $1 \leq i \leq k \leq m$
       ii. Estimate cache-performance for all $z_i$ as defined in Section 9.1.2
       iii. Assign $z_i$ with best cache-performance to $z_{opt}$
   (d) // Run a stage with the best known performance.
       If $z_{opt}$ has been assigned,
       i. $j = j + 1$
       ii. run stage $t_j$ with configuration $(z_{opt}, repl)$
3. // data-reconfiguration phase
   Add to repl all fragments $F \in \mathcal{F}_{t_j}$ with
   $F \notin repl$, $mRead_{\varnothing}(F) \geq minRead$, $mWrite_{\varnothing}(F) \leq minWrite$
4. Remove from repl all fragments F with
   $mRead_{\varnothing}(F) < minRead$, $mWrite_{\varnothing}(F) > minWrite$ and $mEff_{\varnothing}(F) < minEff$
5. Jump to 2
```

The algorithm starts with no replicated fragments. In the operational phase, first (Step 2a) the configuration z_{max} is computed that for all procedures in $\mathcal{S}_t$ the execution of all IO statements is shifted to the cache. For this, each split parameter is set to $ID(S_i)$ which is the set of identifiers of all IO statements in a procedure S_i. Note that the set $\mathcal{S}_t$ depends on a stage t, since procedures can be added or removed at run time.

In Step 2b the systems performs a stage with the configuration $(z_{max}, repl)$ where a cache executes all IO statements for which data is locally available. Note that in the very first operational phase no IO statement is executed at a cache, since $repl = \emptyset$. However, during this phase the values of $mRead_t(F)$ and $mWrite_t(F)$ are collected for each fragment, such that in Step 3 the first fragments will be placed at a cache.

Based on the prediction technique, a configuration z_{opt} is computed in Step 2c. Since we only consider continuous stages, the computation is done in background, such that there is no additional time slot between two successive stages. Further, the prediction is only done if there are replicated fragments ($repl \neq \emptyset$). Otherwise, there are no executions at a cache and therefore no valid subsequences in the execution histories, hence nothing to predict.

To compute z_{opt}, the algorithm picks at most m distinct configurations z_i that perform less or equal computations than z_{max} at the cache. Note that k depends on z_{max}. The more IO statements are executed at a cache, the more different z_i exist that contain less than these IO statements. For each z_i the algorithm applies the above prediction technique and computes the resulting cache-performance. The configuration with the best cache-performance is assigned to z_{opt}. As we have shown in Section 9.4, the prediction procedures reasonable results.

When such a z_{opt} has been assigned, a new stage is executed in Step 2d with a possibly better performance. If a computation is still running in background, the last known z_{opt} is used. As a result, every second stage is performed with a possibly optimal performance. Clearly, the success depends on the parameter m that determines the duration of the background computation. If it is too small, possibly not enough z_i are considered and no better configuration can be found. If it is too high, the background computation can take multiple stages, such that the resulting z_{opt} possibly does not match the current load situation any more. Hence, an appropriate search strategy has to be applied. We follow up this issue at the end of this Section.

After the operational phase, the algorithm enters its data-reconfiguration phase (Step 3 and 4) where the set of replicated fragments is adapted. In Step 3 all fragments are placed at the cache which satisfy the constraints on the read and write frequency. For this we consider the average of the frequency over the entire operational phase. Recall from Chapter 4 that the number of fragments is dynamic at run time and that the server maintains a set $\mathcal{F}_t$ of fragments. The subscript t emphasizes its dependency of a stage.

In Step 4 we remove all fragments from the cache that do not satisfy the constraints on the read and write frequency, and that do not produce the required execution time of valid queries. Hence, the algorithm is rather greedy in the placement of fragments, since a fragment is kept replicated during the entire operational phase.

The algorithm demonstrates how the prediction technique can be integrated into a primitive search over the configuration space. The algorithm adapts the set of replicated fragments and the split of the code between cache and server according to the requested data and the load conditions. Replicated data is adapted in a 24-hour cycle with the assumption that at midnight there are no user activities, such that the system can perform reconfiguration tasks. If other periods with minor activities are known the cycle can be shortened. The split of the code is adapted every second stage (Step 2c and 2d).

As mentioned at the beginning of this Section, the algorithm relies on a number of assumptions and restrictions. In order to obtain a complete and feasible solution for the dynamic optimization problem, a standard technique (see Section 7.5) has to be customized appropriately. We suggest to use genetic algorithms (GAs) for the following reasons:

1. GAs are suggested by many authors to solve complex dynamic optimization problems. These heuristic search methods try to track moving optima, rather than to determine their precise location.

2. Much work has been done in extending GAs with a case-based memory which are used to detect repetitive pattern of the user behavior and to predict the position of the optima in the search space. The case-based memory is an ideal place for incorporating our prediction technique.

3. A genetic algorithm performs a heuristic search over the search space. For this it uses evolutionary concepts, such as mutation and combination of genetic information, to derive new positions in the search space. In order to apply a genetic algorithm to our problem an appropriate mapping between an individual and a configuration $(z, repl)$ has to be defined.

However, such a mapping requires a detailed study of existing approaches of genetic algorithms, their implementation, and comparison. In a student reseach project [90] we have investigated primitive mappings to genetic algorithms for the in Chapter 3 discussed web shopping application. The results have also been partially published in [60].

9.6 Summary and Discussion

Given a cache, a configuration $(z, repl)$, and data of a completed stage t, we have shown how to predict the performance of the stage t for the configurations $(z', repl)$ with $z' < z$. The configurations z' differ from z by performing less computations at the cache, thus adding more computations to the server. As the evaluation has shown, the deviation between the predicted and observed values is comparatively low. As a result, we are able to select a $z_{opt} < z$ with the best performance in t. We have presented an example of a greedy optimization algorithm that uses the prediction technique to compute the configuration of the next stage. Since the prediction technique is partial, the algorithm requires a stage where all computations are done at the cache. From the run time behavior of this stage, the performance of a maximal number m of configurations can be predicted. However, more work has to be done to turn the presented optimization algorithm into a complete and applicable solution for the dynamic optimization problem.

During the course of this Chapter we have evaluated the prediction technique on the ONE-System. However, there are still the following limitations that open enough space for future extentions of our approach:

1. The method is partial, since it only applies to configurations $z' < z$ and requires the same set *repl* of replicated fragments. The latter is necessary since z' requires the same or less fragments at the cache. A z' does not require new fragments to be replicated at a cache, since z' performs equal or less computations at a cache.

   Our model does not allow the prediction of the processing time of splits $z' > z$ which puts more computational efforts to the cache or which use a different set of replicated fragments. In terms of the presented model, this requires the *guessing* of the frequency of valid subsequences (captured by the counters $c_{invalid}$ and c_{valid}) that include new query executions at a cache. However, this is not possible with our model.

2. As we have shown, the values of the basic parameters depend on the load of a cache and the central server. Hence, we require a way to observe the load and a technique to determine the values of these parameters for the observed load. Our model for observing the load considers only the amount of executed IO statements, hence does not recognize the impact of other applications that run at cache and server. Furthermore, we restricted the prediction to single-user caches where the execution time of IO statements is rather constant. This results from the nature of the defined execution histories that do not fully capture the partial execution at a cache, hence do not allow to reason about all executed IO statements at a cache.

3. As stated at the end of Chapter 8, the prediction technique also requires a high difference between the minimal and maximal cache-performance. With a deviation of at most $\pm$ 10% our technique will not produce reasonable results for situations where the minimal and maximal processing times are very close and also differs by $\pm$ 10%.

4. Another important issue is the synchronization of the optimization and prediction tasks of all attached caches. The configurations of all attached caches significantly influence the load conditions at the central server. For a proper detection of the server load, it is essential that a cache knows about the decisions of other caches in changing the configuration. However, we consider only the optimization from the perspective of a single cache and the simultaneous optimization of multiple database caches remaines a challenging issue.

Chapter 10

Conclusion

Database caching techniques have been suggested in literature to improve the performance of client-server database systems. Recent techniques further improve the performance by integrating the caching scheme into concurrency control and query processing. In this dissertation we further integrate such a technique into the execution engine of a database system and develop a novel execution scheme that allows to split and simultaneously execute stored procedure code between cache and server. As the main result we show that this integration improves the efficiency of database caching.

10.1 Summary and Contributions

We briefly discuss contributions of this thesis and summarize the content of all Chapters.

Part I

In Chapter 2 we investigate current database caching techniques and classify them along multiple criteria. We show that one strategy for improving a caching scheme is its deeper integration into the database system. This has mainly been achieved by combining caching with concurrency control and query processing. We show the research gap in integrating database caching into the execution engine of database management systems for procedural application logic that is kept inside the database in terms of stored procedures, triggers, etc. These concepts are very common and provided by almost every commercial database product.

Chapter 3 presents an illustrative example of our approach. On an intuitive level we first discuss two traditional execution protocols for database caching and second compare it to our novel simultaneous execution scheme. As a result we could show that the novel scheme improves the performance for the chosen example. The improvement results from our key concept of twin transactions in combination with a partial execution and query result shipping. That is, after a stored procedure has been initiated, it is executed at the cache of the client and the server in parallel. Both execute only a portion of the procedure code, such that the execution is split among cache and server. The cache sends each computed query result to the server which re-uses these results instead of computing them by itself. Hence, load is taken from the central server and portions of the procedure code can be executed in a parallel manner. In the example we also discuss limitations of this scheme. In general, it does not apply to situations where traditional caching schemes are not successful, where huge query results have to be shipped to the server and where cache or server are connected by a slow and unreliable network.

Part II

In Chapter 4 we define an elementary client-server database system where each client is extended by a local database management system that is able to execute stored procedures. Server data is replicated to clients in terms of table fragments by an optimistic synchronization protocol. It is optimistic, since updates are propagated to clients apart from the boundaries of transactions which can cause data inconsistencies. To efficiently detect data inconsistencies at clients, we attach a version identifier to each fragment such that equal versions at client and server denote equal fragment content.

The main contribution of Chapter 4 is the concept of twin transactions that defines the parallel execution of procedure splits at a local database management system of a client and the central server. Both executions of the procedure produce the same effect on local data if they have operated on equal fragment versions. To achieve this property, we have to restrict the procedural language by removing non-deterministic operations and further side-effects that can result from trigger or the check of integrity constraints. Twin transactions are the starting point for developing our novel simultaneous execution scheme.

The novel execution scheme and split twin transactions are elaborated in Chapter 5. We show how the local database management system partially executes the procedure code and how computed query results are shipped to a query result cache (QRC) at the server. The QRC serves as an intermediate memory to store query results. Whenever the server is looking for a result, it consults the QRC. For the server we have shown how delivered query results are efficiently verified by utilizing fragment versions.

The main contribution of Chapter 5 is the novel execution scheme that can be configured at run time by two parameters: (1) the amount of code execution at a client that is captured by the split parameter and (2) the set of replicated fragments. We show that the scheme is correct for any of these settings. Furthermore, we show how to preselect meaningful settings of the split parameters that respect dependencies in the procedure code. The preselection is used in Part III by the optimization to reduce the size of the search space.

Chapter 6 presents a primitive client-server database system with a prototype implementation of our scheme. We show by various experiments that the new scheme outperforms the traditional one. The improvement results from: (1) less communication costs between cache and server, (2) a verification of queries that is done at the server in parallel to computations at the cache and (3) independent queries (if they exist). Further, we investigate different setting of the configuration parameters and their impact on the performance. Another contribution of Chapter 6 is the conclusion that the proper setting of the configuration parameters depends on run time factors such as the system's load and available hardware resources. Hence, a proper optimization of these parameters requires to consider run time information.

Part III

In Chapter 7 we take a closer look at the run time behavior of client-server database systems and point to the influencing factors at the request (user behavior) and resource level. As we argue, these factors are difficult to measure and to predict. By this observation and the results of the experiments, we suggest to define the proper setting of the configuration parameters by a dynamic optimization problem that takes changes of these factors during run time into account.

To define the optimization problem for a cache, we modell the system's behavior by successive time periods of constant length and presented measures to capture the performance of a cache that target the efficiency of procedure executions and fragment placements. For capturing the performance of procedures we suggest three alternative measures — the response time of procedures, the cache utilization, or the code balancing between cache and server. For capturing the performance of a fragment we suggest using its read- and write frequency, as well as the fragment efficiency which represents the amount of query executions at a cache whose results could be successfully reused by the server.

The presented optimization problem defines the maximal performance of a cache and has to be "independently" solved by a cache. The problem is to find an appropriate setting of the configuration parameters for each stage with a maximal resulting cache performance subject to storage and fragment access and efficiency restrictions. We define the maximal performance of a stage by taking the previous completed stage into account which provides the values of the observed run time measures, such as fragment access and efficiency. This model allows a cache to react to changing run time behavior stage-by-stage.

The first step towards solving the optimization problem is made in Chapter 8 where we define a model of the simultaneous execution scheme for computing the processing time of a procedure. The model is based on the run time behavior of a procedure that is collected within execution histories for each stage. Such histories mainly contain information about the execution time of elementary database operations and the frequency of valid and invalid queries on different paths through the procedure code. Based on this information the model allows us to recompute the observed processing time of a procedure. The model mimics the simultaneous execution scheme including the shipping of query results, verification, idle times, etc. The preciseness of the model is evaluated by using a prototype implementation. We are able to show, that the predicted and the observed values of the processing time are very close.

The contribution of Chapter 9 is the extention of the frequency-based model of Chapter 8 to predict the performance of a cache for different settings of the split parameters. As a result, the performance can be computed in advance for a subset of all possible split configurations (subset of the search space) which provides a local solution for the optimization problem. The key of the estimation is the prediction of the execution frequencies of valid and invalid queries for a given split configuration. The method is suboptimal, since these execution frequencies can only be derived from already performed executions at a cache which produce the frequencies for valid and invalid queries and executed paths through the procedure code. These frequencies cannot be predicted for split configurations that refer to procedure code that has not yet executed at a cache.

The proper estimation of the performance also requires to consider the load of the server and the cache. We introduce a primitive measure for capturing the number of database operations of a stage. Based on this measure a cache is able to learn the correlation between the system's load and the execution time of primitive database operations (queries and updates) which is important for a precise estimation. Again, the preciseness of the prediction technique is evaluated based on the experimental data gained from the prototype implementation.

Finally, we present a greedy algorithm in Chapter 9 that provides a solution for the optimization problem. Due to the partialness of the prediction technique it requires a special learning stage where all executions are shifted to a cache. This stage produces execution frequencies for all valid and invalid queries which are the base for predicting the frequencies of alternative split configurations. Based on this, the algorithm computes the performance of a set of split configurations and uses the best of them for the next stage. These two types of stages are constantly toggelled, such that the algorithm is able to react to changing load conditions within a few stages. Further, the algorithm adapts the content of a cache within a fixed cycle to remove less-frequently and to add high-frequently accessed data. However, the algorithm serves primarily as an example for demonstrating the prediction technique, rather than as a full solution for the optimization problem. Hence, more work has to be done in finding an appropriate solution for the optimization problem.

10.2 Open Directions for Further Work

During the course of this dissertation we have pointed out numerous open directions for further work that in the following are briefly summarized.

Full Support of Stored Procedures and Queries

To exemplify our approach, we put restrictions on the schema constraints (e.g. foreign keys) and advanced concepts for integrity enforcement (e.g. trigger). To use the novel scheme in a real-world setup, these concepts have to be taken into account by the simultaneous execution scheme. As we suggest in Chapter 4, this can be done at the compiler level.

Another subject is the integration of query execution plans into the low-level procedure code. Then, internal functions and sub-procedure calls in the query expressions could also be handled by our scheme. As a result the low-level procedure contains the computation of sub-queries, joins, etc. explicitly, which enables our scheme to handle partial query evaluation at caches.

An interesting question is whether our approach affects the programming or the coding rules of developers. To support our scheme, developers could mark independent queries and put them at the beginning of the procedure code. These queries are candidates for being executed at a cache and do not have to be automatically detected by a semantic analysis of the procedure code.

Fine Tuning the Simultaneous Execution Scheme

The presented simultaneous execution scheme provides numerous starting points for further optimization and fine tuning.

One issue is the reduction of useless idle times at the server that result from waiting times for notifications and query results that are delivered by a cache. While processing a statement, a cache could perform a look-ahead within the procedure code to determine the state of the next query. The earlier the state can be reported to the server, the less useless idle time occurs.

Another issue is the broader use of the query result cache (QRC). Currently, a transaction at the server does only consider query results in the QRC that result from the same twin transaction. In a broader use, a transaction at the server can check the entire QRC for usable query results. Then, cached query results can also be reused from other previous or currently running transactions. Of course, this would increase the search time for valid results in the QRC, but potentially the number of reuse cases can be increased.

As we discuss in Chapter 5 and 6, the server switches to normal mode whenever an invalid query has been detected. As shown be the experiments, this switch can increase as well as decrease the performance of a cache. To determine the best switching strategy, more experiments on real-world applications have to be done.

Our scheme undoes all updates that have been made during an execution at a cache. This is useful, since a cache might have performed erroneous updates which can cause inconsistencies at the cache. However, if a cache has operated on the same fragment version as the server and has performed exactly the same updates as the server in their execution, these updates can be committed and its rejection is needless. Another open direction is the extension of our scheme by committing updates at the cache. This possibly further improves the up-to-dateness of cache data, since then locally committed updates are not delayed by the optimistic synchronization scheme.

As defined in Chapter 8, the model of calculating the processing time of a procedure on the basis of execution histories mimics the behavior of the simultaneous execution including the shipping of query results, verification, idle times, etc. Currently, our model does not cover all behavior of the scheme, such that a proper mapping of this behavior to the model potentially further improves its preciseness.

Eliminate User-Provided Parameters for the Optimization Problem

The optimization problem in Chapter 7 relies on three user-provided parameters that control the placement and removement of fragments at a cache. The first two (*minRead* and $minWrite$) are used as entry barriers for fragments to be replicated. If the read-frequency is too low or the write-frequency too high, a fragment is not considered for replication. The third parameter (*minEff*) is used to remove already replicated fragments that do not produce the required amount of valid query executions at a cache. An open question is whether these parameters can be eliminated.

As we argue at the end of Chapter 7, the parameter *minEff* can be eliminated by developing a model for capturing the synchronization costs of a fragment and comparing it with the benefits from valid query executions at the cache on this fragment. If the benefits exceed the costs, the fragment is further replicated and removed from a cache otherwise.

Solutions to the Optimization Problem

The primary goals of the Chapter 8 and 9 are to develop a model for the novel scheme and to derive possible model-specific optimization strategies. The example of an optimization algorithm in Chapter 9 can be extended in many ways. As we suggest, further work should be applied in considering heuristic search approaches, e.g., evolutionary approach, such as genetic algorithms, to efficiently solve the optimization problem.

Such a mapping has to consider our partial solution of the optimization problem as well as the proper handling of the feed-back problem. The latter results from a cyclic dependency of influencing parameters. That is, the optimizer requires the precise values of the execution time of elementary database operation which depend on the load of a cache and the server. Since the optimizer sets the split parameters at a cache it influences the load of both. Hence, the decisions of the optimizer possibly change this execution time by tracking the optimal performance.

Local versus Global Optimization

Another important issue is the synchronization of the optimization tasks of all attached caches of the system. The configurations of all attached caches significantly influence the load conditions at the central server. For a proper detection of the server load it is essential that each cache knows about the decisions of other caches in changing the configuration. However, we consider only the optimization from the perspective of a single cache, where the server is just a resource that due to different load-conditions computes requests with a different efficiency.

Hence, a possibly better optimization result can be achieved by a global optimizer that for the entire client-server system coordinates the setting of the replication and split parameters. A global optimization seems very promising, but it also requires optimizing multiple caches at the same time. Further, the optimizer should not run at the bottle-necked server, such that the only possibility is autonomous optimizers at caches that communicate with each other. In all a difficult, but challenging task.

Bibliography

[1] Serge Abiteboul and Oliver M. Duschka. Complexity of Answering Queries Using Materialized Views. In *Proc. of the ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS), Seattle*, pages 254–263, 1998.

[2] S. Adali, S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Montreal, Canada*, pages 137–148, 1996.

[3] A. Aho, R. Sethi, and J. Ullmann. Compiler construction. volume 3, 1989.

[4] M. Altinel, C. Bornhövd, S. Krishnamurthy, C. Mohan, and H. Pirahesh. Cache tables: Paving the way for an adaptive database cache. In *In Proc. of the 2003 Conf. on Very Large Data Bases (VLDB), Berlin, Germany*, pages 718–729, 2003.

[5] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. Dbproxy: A dynamic data cache for web aplications. In *ICDE, Bangalore, India, March*, pages 493–504, 2003.

[6] J. Anton, L. Jacobs, X. Liu, J. Parker, Z. Zeng, and T. Zhong. Web caching for database applications with oracle web cache. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Madison, WI*, pages 594–599, 2002.

[7] D. F. Bacon and R. E. Strom. Optimistic parallelization of communicating sequential processes. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, volume 26, pages 155–166, Williamsburg, VA, April 1991.

[8] M. Balaban and P. Shoval. Enhancing the ER model with structure methods. In *CAiSE'98/IFIP Workshop on Evaluation of Modeling Methods in Systems Analysis and Design*, June 1998.

[9] M. Balaban and P. Shoval. Enhancing the ER model with structure methods. *Journal of Database Management*, 10(4):14–23, 1999.

[10] Daniel Barbara. Mobile Computing and Databases - A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):108–117, 1999.

[11] C. N. Bendtsen and T. Krink. Dynamic Memory Model for Non-Stationary Optimization. *In Proc. of the Fourth Congress on Evolutionary Computation (CEC-2002)*, 1:145–150, 2002.

[12] Elisa Bertino, Elena Ferrari, and Evaggelia Pitoura. An Access Control Mechanism for Large Scale Data Dissemination Systems. In *RIDE-DM, April 2001, Heidelberg, Germany*, pages 43–50, 2001.

[13] Jürgen Branke. Evolutionary Approaches to Dynamic Optimization Problems. In Jürgen Branke and Thomas Bäck, editors, *Evolutionary Algorithms for Dynamic Optimization Problems*, pages 27–30, San Francisco, California, USA, 7 2001.

[14] F. Bry, H. Decker, and R. Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. *Proceedings of Extending Database Technology*, pages 488–505, 1988.

[15] K. S. Candan, W. Li, Q. Luo, W. Hsiung, and D. Agrawal. Enabling dynamic content caching for database-driven web sites. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Santa Babara, CA*, pages 532–543, 2001.

[16] M. Carey, M. Franklin, M. Linvy, and E. Shekita. Data Caching Tradeoffs in Client-Server DBMS Architecture. In *In Proc. of the ACM SIGMOD Conf. on Management of Data, Denver, USA*, pages 357–366, 1991.

[17] M. Celma and H. Decker. Integrity checking in deductive databases. the ultimate method? *Proceedings of 5th Australiasian Database Conference*, pages 136–146, 1995.

[18] S. Ceri, P.Fraternali, S. Paraboschia, and L. Tanca. Automatic gerneration of production rules for integrity maintenance. In *ACM Transactions on Database Systems*, volume 19(3), pages 367–422, 1994.

[19] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. *Proceedings of the 16. International Conference on Very Large Data Bases*, pages 566–577, 1990.

[20] Stefano Ceri and Piero Fraternali. *Designing Database Applications with Objects and Rules*. Addision Wesley, 1997.

[21] U. Cetintemel and P. Keleher. Light-Weight Currency Management Mechanisms in Deno, In The 10 th IEEE Workshop on Research Issues in Data Engineering (RIDE2000), pages 17-24, February 2000.

[22] U. Cetintemel, P. Keleher, and M. Franklin. Support for speculative update propagation and mobility in deno, pages 509-516, ICDCS 1999.

[23] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseak Shim. Optimizing Queries with Materialized Views. In *11th Int. Conference on Data Engineering*, pages 190–200, Los Alamitos, CA, 1995. IEEE Computer Soc. Press.

[24] C. Chen and N. Roussopoulos. Implementation and performance evaluation of the adms query optimizer: Integrating query result caching and matching. In *Proc. of EDBT Conf.*, pages 323–336, 1994.

[25] Panos K. Chrysanthis. Transaction processing in mobile computing environment. In *IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 77–83, Princeton, New Jersey, 1993.

[26] Simon Cuce, Arkady B. Zaslavsky, Bing Hu, and Jignesh Rambhia. Maintaining consistency of twin transaction model using mobility-enabled distributed file system environment. In *DEXA Workshops*, pages 752–756, 2002.

[27] K. Curewitz, P. Krishnan, and J. Vitter. Practical prefetching via data compression. In *In Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 43–53, 1993.

[28] C.R. Cutler and R.T. Perry. Real Time Optimization with Multivariable Control is Required to Maximize Profits. In *Computers and Chemical Engineering*, volume 7, pages 663–667, 1983.

[29] Shaul Dar, Michael J. Franklin, B. T. Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *Proc. VLDB Conf., Bombay, India, September, 1996*, pages 330–341, 1996.

[30] M.L. Darby and D.C. White. On-line Optimization of Complex Processes. In *Chemical Engineering Progress*, pages 51–59, 1988.

[31] S.K. Das and M.H. Wiliams. A path finding method for constraint checking in deductive databases. *Data and Knowledge Engineering 3*, pages 223–244, 1989.

[32] H. Decker. Integrity enforcements on deductive databases. *Proceedings of the 1st Internetional Conference on Expert Database Systems*, pages 271–285, 1986.

[33] S. Dressloch, T. Härder, N. Mattos, B. Mitschang, and J. Thomas. KRISYS: Modeling concepts, implementations techniques and client/server issues. In *The VLDB Journal*, pages 7(2):79–95, 1998.

[34] I.M. Edwards and A. Jutan. Optimization and Control using Response Surface Methods. In *Computers and Chemical Engineering*, volume 21(4), pages 441–453, 1997.

[35] W. Keith Edwards, Elizabeth D. Mynatt, Karin Petersen, Mike Spreitzer, Douglas B. Terry, and Marvin Theimer. Designing and implementing asynchronous collaborative applications with bayou. In *ACM Symposium on User Interface Software and Technology*, pages 119–128, 1997.

[36] J. Eggermont and T. Lenaerts. Dynamic Optimization using Evolutionary Algorithms with a Cased-based Memory. *In Proc. of the 14. Belgium-Netherlands Conference on Artifical Intelligence*, pages 107–114, 2002.

[37] M. Franklin, M. Carey, and M. Livny. Transactional client-server cache consistency. In *In ACM Trans. on Database Systems*, volume 22(3), pages 315–363, 1997.

[38] M. Franklin and D. Kossman. Cache investment strategies. In *Technical Report CS-TR-3803, University of Maryland, College Park, MD 10742. Submitted to IEEE Knowledge and Data Engineering*, 1997.

[39] M. J. Franklin and M. Carey. Client-server caching revisited. In *Readings in Database Systems (3rd Edition), M. Stonebraker, J. M. Hellerstein (Eds.), Morgan Kaufmann Publishers Inc., In Proc. of the IWDOM*, pages 57–78, 1998.

[40] M. J. Franklin, M. J. Carey, and M. Livny. Transaction Client-Server Consistency: Aternatives and Performance. In *Technical Report CS-TR-3511 and UMIACS TR 95-84, University of Maryland, College Park, MD 10742*, 1995.

[41] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance tradeoffs for client-server query processing. In *Proceedings of the 1996 SIGMOD Conference, Montreal, Canada*, pages 149–160, 1996.

[42] M.J. Franklin. *Caching and Memory Management in Client-Server Database Systems.* Dissertation, Computer Sciences Department, University of Wisconsin-Madison, 1993.

[43] P. Fraternali, S. Paraboschi, and L. Tanca. Automatic rule generation for constraints enforcement in active databases. In U. Lipeck and B. Thalheim, editors, *Modeling Database Dynamics*, pages 153–173. springer WICS, 1993.

[44] C. A. Gerlhof and A. Kemper. A multi-threaded architecture for prefetching in object bases. In *In Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, volume 779 of Lecture Notes in Computer Science, pages 351–364, 1994.

[45] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The Dangers of Replication and a Solution. In *In Proc. of the 1996 SIGMOD Conference, Montreal, Canada*, pages 173–182, 1996.

[46] John J. Grefenstette. Genetic algorithms for changing environments. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature 2 (Proc. 2nd Int. Conf. on Parallel Problem Solving from Nature, Brussels 1992)*, pages 137–144, Amsterdam, 1992. Elsevier.

[47] R. Grimm. Systems Directions for Pervasive Computing. In the Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Elmau, Germany, May 2001, pages 128-132, 2001.

[48] Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques and Applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, 1995.

[49] Ashish Gupta and Jennifer Widom. Local verification of global integrity constraints in distributed databases. In *Proc. of the 2003 SIGMOD Conference, Washington D.C.*, pages 49–58, 1993.

[50] Richard G. Guy, Peter L. Reiher, David Ratner, Michial Gunter, Wilkie Ma, and Gerald J. Popek. Rumor: Mobile Data Access Through Optimistic Peer-to-Peer Replication. In *ER Workshops, Singapore*, pages 254–265, 1998.

[51] Laura M. Haas, Donald Kossmann, and Ioana Ursu. Loading a cache with query results. In *In Proc. of the Conf. on Very Large Data Bases (VLDB), Edinburgh, Scotland*, pages 351–362, 1999.

[52] Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors. *DBCache: Middle-tier Database Caching for Highly Scalable e-Business Architectures.* In the Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003.

[53] T. Härder, B. Mitschang, U. Nink, and N. Ritter. Workstation/Server Architekturen für datenbankbasierte Ingeniueranwendungen. In *Informatik - Forschung und Entwicklung*, pages 10(2):43–53, 1995.

[54] Theo Härder and Andreas Bühmann. Database caching - towards a cost model for populating cache groups. In *ADBIS, Budapest, Hungary*, pages 215–229, 2004.

[55] Christop Hartwig. *A Middleware Architecture for transactional, object-oriented applications.* Dissertation, Freie Universität zu Berlin, 2004.

[56] Holland and John. *Hidden Order - How adaptation builds complexity.* Addision Wesley, 1995.

[57] S. Jurk and M. Balaban. Improving Integrity Constraint Enforcement by Extended Rules and Dependency Graphs. In *Proc. 22th Conf. on DEXA*, pages 501–516, 2001.

[58] S. Jurk and M. Balaban. Update-Consistent Query Results by Means of Effect Preservation. In *Proc. Fifth International Conf. on Flexible Query Abswering Systems (FQAS'02)*, pages 28–43, 2002.

[59] S. Jurk and M. Balaban. Towards effect preservation of updates with loops. In *Fifth IFIP TC-11 WG 11.5 Working Conference on Integrity and Internal Control in Information Systems*, pages 59–75. Kluwer Academic Publishers, 2003.

[60] S. Jurk, U. Leser, and J.-L. Marzo. Cooperative Transaction Processing between Clients and Servers. In *ADBIS (Local Proceedings)*, pages 132–146, 2004.

[61] S. Jurk and M. Neiling. Client-Side Dynamic Preprocessing of Transactions. In *ADBIS, 7th East European Conf., Dresden, Germany*, volume 2798 of *LNCS*, pages 103–117. Springer, 2003.

[62] M. Kaiser, K. Tsui, and J. Liu. Adaptive distributed caching. in proceedings of the 2002 congress on evolutionary computation, pages 1810-1815, 2002.

[63] F. Kappe. A scalable architecture for maintaining referential integrity in distributed information systems. *J.UCS: Journal of Universal Computer Science*, 1(2):84–95, 1995.

[64] Arthur M. Keller and Julie Basu. A Predicate-based Caching Scheme for Client-Server Database Architectures. *VLDB Journal: Very Large Data Bases*, 5(1):35–47, 1996.

[65] A. Kemper and D. Kossmann. Dual-buffering strategies in object bases. In *In Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 427–438, 1994.

[66] Donald Kossmann, Michael J. Franklin, Gerhard Drasch, and Wig Ag. Cache investment: integrating query optimization and distributed data placement. *ACM Transactions on Database Systems*, 25(4):517–558, 2000.

[67] S.Y. Lee and T.W. Ling. Further improvement on integrity constraint checking for stratisfiable deductive databases. In *Proc. 22th Conf. on VLDB*, pages 495–505, 1996.

[68] S. Link and K.-D. Schewe. Computability and Decidability Issues in the Theory of Consistency Enforcement. In *Electronic Notes in Theoretical Computer Science*, volume 42, 2001.

[69] Peng Liu, Paul Ammann, and Sushil Jajodia. Incorporating Transaction Semantics to Reduce Reprocessing Overhead in Replicated Mobile Data Applications, In Proc. of the ICDCS, Austin, Texas, USA, pages 414-423, 1999.

[70] Q. Luo and J. F. Naugthon. Form-based proxy caching for database-backed web sites. In *In Proc. of the Conf. on Very Large Data Bases (VLDB), Rome, Italy*, pages 191–200, 2001.

[71] Sanjay Kumar Madria and Bharat K. Bhargava. A Transaction Model for Mobile Computing. In *International Database Engineering and Application Symposium (IDEAS), Cardiff, Wales*, pages 92–102, 1998.

[72] E. Mayol and E. Teniente. Structuring the process of integrity maintenance. In *Proc. 8th Conf. on Database and Expert Systems Applications (DEXA), Toulouse, France*, pages 262–275, 1997.

[73] E. Mayol and E. Teniente. Addressing efficiency issues during the process of integrity maintenance. In *Proc. 10th Conf. on Database and Expert Systems Applications (DEXA), Florence, Italy*, pages 270–281, 1999.

[74] E. Mayol and Ernest Teniete. A survey of current methods for integrity constraint maintenance and view updating. In Chen, Embley, Kouloumdjian, Liddle, Roddick, editor, *Intl. Conf. on Entity-Relationship Approach*, volume 1727 of *Lecture Notes in Computer Science*, pages 62–73, 1999.

[75] Subhasish Mazumdar and Panos K. Chrysanthis. Achieving Consistency in Mobile Databases through Localization in PRO-MOTION. In *DEXA Workshop, Florence, Italy*, pages 82–89, 1999.

[76] M. Miki, T. Hiroyasu, M. Kasai, K. Ono, and T. Jitta. Temperature Parallel Simulated Annealing with Adaptive Neighborhood for Continuous Optimization Problem. In *In Proc. of the 2. International Workshop on Intelligent Design and Application*, pages 149–154, 2002.

[77] C. Mohan. Caching technologies for web applications. In *In Proc. of the Conf. on Very Large Data Bases (VLDB), Rome, Italy*, page Tutorial, 2001.

[78] C. Mohan. Application servers and associated technologies. In *In Proc. of the Conf. on Very Large Data Bases (VLDB), Hong Kong, China*, page Tutorial, 2002.

[79] Brian D. Noble. *Mobile Data Access.* PhD thesis, Carnegie Mellon University Pittsburgh, USA, 1998.

[80] T. Oszu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[81] M. Palmer and S. Zdonik. FIDO: A cache that lears to fetch. *In Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 255–264, 1991.

[82] E. Pitoura and G. Samaras. *Data Management for Mobile Computing*, volume 10. Kluwer Academic Publishers, 1998.

[83] Evaggelia Pitoura. A Replication Schema to Support Weak Connectivity in Mobile Information Systems. In *7th International Conference on Database and Expert Systems Applications (DEXA)*, pages 510–520, 1996.

[84] Evaggelia Pitoura and Bharat Bhargava. Maintaining Consistency of Data in Mobile Distributed Environments. In *15th International Conference on Distributed Computing Systems (ICDCS)*, pages 404–413, 1995.

[85] PostgreSQL. *http://www.postgreSQL.org.*

[86] A. Rasheed. *Twin-transaction model to support mobile data access.* Ph.d. thesis, School of Computer Science and Software Engineering, Monash University, 1999.

[87] A. Rasheed and A. Zaslavsky. Ensuring Database Availability in Dynamically Changing Mobile Computing Environments. In *Proc. of the 7th Australasian Database Conf.*, pages 100–108, 1996.

[88] M. Rodgriguez-Martinez and N. Roussopoulos. Mocha: a self-extensible database middleware system for distributed data sources, In Proc. of the 2000 ACM SIGMOD Conf. Dallas, TX, USA. pages 213–224, 2000.

[89] Y. Saito. Optimistic Replication Algorithms. Technical report, General Examination Report, University of Washington, 2000.

[90] S. Sämann. Experimentelle Analysen zu clientseitigen Vorberechnungen von Datenbankprozeduren, Studienarbeit, Brandenburgische Technische Universität Cottbus, Lehrstuhl Datenbank- und Informationssyteme, November 2004.

[91] K.D. Schewe and B. Thalheim. Consistency enforcement in active databases. In S. Chakravarty and J. Widom, editors, *Research Issues in Data Engineering – Active Databases*, pages 71–76. IEEE Computer Society Press, 1994.

[92] K.D. Schewe and B. Thalheim. Limitations of rule triggering systems for integrity maintenance in the context of transition specifications. *Acta Cybernetica*, 13:277–304, 1998.

[93] K.D. Schewe and B. Thalheim. Towards a theory of consistency enforcement. *Acta Informatics*, 36:97–141, 1999.

[94] BEA WebLogic Application Server. *http://www.bea.com/products/weblogic/server/*.

[95] IBM Websphere Application Server. *http://www.ibm.com/software/webservers/appserv/*.

[96] The Times Ten Team. Mid-tier caching: The fronttier approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Madison, WI*, pages 588–593, 2002.

[97] Douglas B. Terry, Karin Petersen, Mike Spreitzer, and Marvin Theimer. The case for non-transparent replication: Examples from bayou. *Data Engineering Bulletin*, 21(4):12–20, 1998.

[98] K. Trojanowski and Z. Michalewicz. Evolutionary Algorithms for Non-Stationary Environments. In *Proc. of 8th Workshop: Intelligent Information systems*, pages 229–240. ICS PAS Press, 1999.

[99] van der Voort and A. Siebes. Termination and confluence of rule execution. In *In Proceedings of the Second International Conference on Information and Knowledge Management*, pages 245–255, November 1993.

[100] Gary D. Walborn and Panos K. Chrysanthis. Supporting Semantics-Based Transaction Processing in Mobile Database Applications. In *Symposium on Reliable Distributed Systems*, pages 31–40, 1995.

[101] Gary D. Walborn and Panos K. Chrysanthis. PRO-MOTION : Management of Mobile Transactions. In *Selected Areas in Cryptography*, pages 101–108, 1997.

[102] Y. Wang and L. A. Rowe. Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture. In *Proceedings of the 1991 ACM SIGMOD Int. Conf. on Management of Data*, pages 367–376. ACM Press, 1991.

[103] C. Wen and K. Yelick. Compiling sequential programs for speculative parallelism, In Proceedings of the International Conference on Parallel and Distributed Systems, Taiwan, Dec. 1993.

[104] J. Widom and S. Ceri. Deriving production rules for constraint maintenance. In *Proc. 16th Conf. on VLDB*, pages 566–577, 1990.

[105] J. Widom and S. Ceri. *Active Database Systems.* ISBN 1-55860-304-2, Morgan-Kaufmann, 1996.

[106] W. K. Wilkinson and M. Neimat. Maintaining Consistency of Client-Cached Data. In *16th International Conf. on VLDB, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 122–133. Morgan Kaufmann, 1990.

[107] Q. Xiong and A. Jutan. Continuous Optimization using a Dynamic Simplex Method. volume 58, pages 3817–3828. Chemical Engineering Science, 2003.

[108] Arkady B. Zaslavsky and Zahir Tari. Mobile Computing: Overview and Current Status. *Australian Computer Journal*, 30(2):42–52, 1998.

[109] C. Zhou and M. Hsu. A theory for rule triggering systems. In *Advances in Database Technology-EDBT' 90*, volume 416 of *Lecture Notes in Computer Science*, pages 407–421, 1999.

Notations

Notation	**Explanation**
$access(s)$	function that returns a set of fragments and versions (F, ver) that have been accessed by the execution of the IO statement s, see Section 4.6
$c_{insert}(n)$, $c_{exec}(n)$	counters that are attached to nodes n of an execution history, nodes represent statements (low-level updates or queries), $c_{exec}(n)$ represents the number of executed statements at the server, $c_{insert}(n)$ represents the number of executed insert operations at the cache, see Section 8.3.2
$complete(R)$	function that is called at caches and the server, returns true if all fragments of a table R are locally available, false otherwise, see Section 4.6
$\mathcal{C}$, C	set $\mathcal{C}$ of clients with $C \in \mathcal{C}$, see Section 4.1
$data(F)$	function that returns the content (set of tuples) of a fragment F, see Section 4.4.3
$eval(s, \#s, e)$	function provided by the database management system to execute an IO statement, $s \in$ *ID(S)* is the internal identifier of the IO statement of the corresponding procedure S, e the expression (update or query) and $\#s \in \mathbf{N}$ counts the repetitions of s during the execution of S, see Section 4.3
$eval_C(s, \#s, e)$	implements the function $eval(s, \#s, e)$ and additional behavior to implement the simultaneous execution scheme at a cache, see Section 5.3.1
$eval_S(s, \#s, e)$	implements the function $eval(s, \#s, e)$ and additional behavior to implement the simultaneous execution scheme at the server, see Section 5.4

$\#exec$	is used as execution counter in algorithms, e.g. Section 9.2
$\mathcal{F}$, F	relation of fragments, fragment F in $\mathcal{F}$, $\mathcal{F}$ is also treated as a set, a fragment (F, R, A, c, ver) is represented by an identifier $F \in \mathbf{N}$, a table R, an attribute A, a value c and a version $ver \in \mathbf{N}$, the content of a fragment is defined by $data(F) = \sigma_{A=c}(R)$, see Section 4.4.3
FAH	fragment access history, contains tuples (F, ver, U) with fragment F, version ver and sequence of updates U, see Section 4.4.4
$fragDef(s)$	returns set of fragments (R, A, c) that are required to execute the IO statement s, a value $c = *$ summarizes all fragments of the table R, see Section 4.4.2
$fragID(R, A, c)$	function that returns the identifier of a fragment in $\mathcal{F}$ that is specified by a table R, an attribute A and a value c, see Section 4.4.3
$ID(S)$	set of unique identifiers for all IO statements in a procedure S, see Section 4.3
$hist_t(S)$	execution history of a procedure S in stage t, see Section 8.3
$mLabel_t$	represents a run time value that is observed in a stage t
$mIdle_t(S)$	total sum of the idle time in ms at the cache and the server during all procedure calls of type S in t, see Section 7.2.4
$mDisk_t$	disk space in KB at a cache in t, see Section 7.4
$mEff_t(F)$	fragment efficiency, total sum of the execution time in ms for valid queries at a cache in t, see Section 7.3.3
$mIdle_t^+(S)$	total sum of the idle time in ms at the cache during all procedure calls of type S in t, see Section 7.2.3
$mPT_t(S)$	average processing time in ms of all procedure calls of type S in t, see Section 7.2.2
$mRead_t(F)$	read frequency of a fragment F in t, see Section 7.3.1
$mSize_t(F)$	size in KB of a fragment F in t, see Section 7.4
$mWrite_t(F)$	write frequency of a fragment F in t, see Section 7.3.1
$\pi_A(r)$	projection of table r on column A, see Section 4.1
$pt_C(s)$, $pt_S(s)$	processing time of an executed IO statement at the cache and the server respectively, see Section 7.2.1
$PT(SC)$	processing time of a procedure call SC, see Section 7.2.2

$repl$, $repl_C$	set of replicated fragments for a cache at client C, $repl \subseteq (F)$, see Section 4.5
$\mathcal{R}$, R	set $\mathcal{R}$ of relations/tables with $R \in \mathcal{R}$, see Section 4.1
$\sigma_\varphi(r)$	selection of tuples in r that satisfy the condition φ, see Section 4.1
s	identifier of an IO statement, $s \in \mathbf{N}$, see Section 4.3
$\mathcal{S}$, S	set $\mathcal{S}$ of stored procedures with $S \in \mathcal{S}$, see Section 4.1
SC	a call of a procedure S, see Section 7.2.2
$split$	set of IO statements, $split \subseteq ID(S)$, see Section 5.1
t	in Part II used as tuple, in Part III as a period of time (stage)
$ver(F)$	version of a fragment F, $ver(F) \in \mathbf{N}$, see Section 4.4.3
$Z(S)$	set of split parameters, $Z(S) \subseteq 2^{ID(S)}$, see Section 5.5
$\mathcal{Z}$	configuration space, $\mathcal{Z} = Z(S_1) \times \cdots \times Z(S_k)$ for all $S_i \in \mathcal{S}$ with $1 \leq i \leq k$, see Section 7.4
z	configuration $z \in \mathcal{Z}$, see Section 7.4
$\mathcal{Z}_t$	configuration space of a stage t, emphasis the problem that the number of stored procedure is variable at run time, see Section 7.4