Performance Modelling of Database Designs using a Queueing Networks Approach

Rasha Izzeldin Mohammed Osman

PhD

2010
Performance Modelling of Database Designs using a Queueing Networks Approach

An investigation in the performance modelling and evaluation of detailed database designs using queueing network models

Rasha Izzeldin Mohammed Osman

A thesis submitted for the degree of
Doctor of Philosophy

Department of Computing
School of Computing, Informatics and Media
University of Bradford

2010
Abstract

Databases form the common component of many software systems, including mission critical transaction processing systems and multi-tier Internet applications. There is a large body of research in the performance of database management system components, while studies of overall database system performance have been limited. Moreover, performance models specifically targeted at the database design have not been extensively studied.

This thesis attempts to address this concern by proposing a performance evaluation method for database designs based on queueing network models. The method is targeted at designs of large databases in which I/O is the dominant cost factor. The database design queueing network performance model is suitable in providing what if comparisons of database designs before database system implementation.

A formal specification that captures the essential database design features while keeping the performance model sufficiently simple is presented. Furthermore, the simplicity of the modelling algorithms permits the direct mapping between database design entities and queueing network models. This affords for a more applicable performance model that provides relevant feedback to database designers and can be straightforwardly integrated into early database design development phases. The accuracy of the modelling technique is validated by modelling an open source implementation of the TPC-C benchmark.
The contribution of this thesis is considered to be significant in that the majority of performance evaluation models for database systems target capacity planning or overall system properties, with limited work in detailed database transaction processing and behaviour. In addition, this work is deemed to be an improvement over previous methodologies in that the transaction is modelled at a finer granularity, and that the database design queueing network model provides for the explicit representation of active database rules and referential integrity constraints.
Acknowledgements

Alhumdilil Allah min gablu wa min ba’ad

I would like to express my deepest thanks and gratitude to Prof. Irfan Awan, my supervisor for his support and encouragement and Prof. Michael Woodward, my co-supervisor, for his generous professional guidance, attention and support. Without their help, I would not have been able to complete this project.

I am very grateful to the Iqra Foundation, especially Mr Mohammed Al-Ikhadir, for an initial grant; and the Overseas Research Students Awards Scheme for partial funding of my research. In addition, I would like to extend my deep gratitude to my Uncle, Dr Elamin, for his generosity and hospitality.

Many thanks go to Catherine Gregory from Career Services and to Rona Wilson, Bev Yates and Mark Tympalski from the Department of Computing. Thanks to all the research students I have met through these years: for ideas and inspiration; with special thanks to Monis Akhlaq and Faiz Elsirhani of the Networks and Security Group. A special thanks to my dear friends Ameena Al-Sawaai, Ibitisam Izzeldin, Manal Abdulazzez and Salma M Osman; you have contributed in your own special way.

Finally, I wish to extend my heartfelt appreciation to my parents and sisters for their unconditional love, support and sacrifice which has enabled me to pursue my dreams, whatever and wherever they may be.
Publications


# Table of Contents

ABSTRACT ........................................................................................................................................... I

ACKNOWLEDGEMENTS .................................................................................................................. III

PUBLICATIONS ................................................................................................................................. IV

TABLE OF CONTENTS ...................................................................................................................... VI

LIST OF FIGURES ............................................................................................................................. XI

LIST OF TABLES ............................................................................................................................... XV

CHAPTER 1 INTRODUCTION ............................................................................................................. 1

1.1 MOTIVATION .............................................................................................................................. 2

1.2 OBJECTIVES ............................................................................................................................. 5

1.3 CONTRIBUTIONS ....................................................................................................................... 5

1.4 THESIS OUTLINE ...................................................................................................................... 6

CHAPTER 2 BACKGROUND AND RELATED WORK ........................................................................... 9

2.1 INTRODUCTION .......................................................................................................................... 9

2.2 AN OVERVIEW OF DATABASE DESIGN CONCEPTS ............................................................... 9

2.3 DATABASE SYSTEM PERFORMANCE TUNING ......................................................................... 16

2.4 QUEUEING NETWORKS .............................................................................................................. 18

2.4.1 Queueing Networks for Software Performance Evaluation ............................................... 21

2.5 PERFORMANCE EVALUATION OF DBMS COMPONENTS ..................................................... 22

2.6 A CATEGORIZATION OF TRANSACTIONS IN DATABASE PERFORMANCE MODELS ............ 23

2.6.1 The Black Box Model ........................................................................................................... 24

2.6.2 The Transaction Processing Model ....................................................................................... 25

2.6.3 The Transaction Size Model ................................................................................................ 27

2.6.4 The Transaction Phase Model .............................................................................................. 28
2.6.5 Discussion .......................................................................................................................... 30

2.7 The Exponential Service Time Assumption ........................................................................... 31

2.8 Database System Performance Evaluation Methodologies ................................................. 33

2.8.1 A General Framework for Database System Performance Prediction ......................... 33

2.8.2 Methodologies Based on Sevcik’s Layered Approach .................................................. 36

2.8.2.1 The Hierarchical DBMS Evaluation Model ................................................................. 36

2.8.2.2 The Prophet Model ....................................................................................................... 37

2.8.2.3 The MOSES Model and JOSHUA Prototype ............................................................... 37

2.8.2.4 A Relational Database Performance Analysis Tool ................................................... 38

2.8.2.5 CLISSPE: Client/Server Software Performance Evaluation Tool ......................... 39

2.8.2.6 Discussion .................................................................................................................... 41

2.8.3 An Approach for Parallel Relational Database System Performance Evaluation ........ 44

2.8.3.1 Discussion .................................................................................................................... 47

2.9 Summary and Contribution ................................................................................................. 48

CHAPTER 3 A QUEUEING NETWORKS APPROACH FOR THE PERFORMANCE MODELLING OF DATABASE DESIGNS............................................................................................................. 50

3.1 Introduction ........................................................................................................................ 50

3.2 The Database Design Queueing Network Model ............................................................... 52

3.2.1 Specifying Service Demands .......................................................................................... 54

3.2.1.1 The Service Demand Cost Model .............................................................................. 55

3.2.2 Building the Queueing Network Model .......................................................................... 59

3.2.3 An Example .................................................................................................................... 61

3.3 The Formal Specification ..................................................................................................... 63

3.3.1 Database Design Formal Specification ......................................................................... 64

3.3.2 Queueing Network Model Formal Specification ............................................................. 68

3.3.3 Building the Queueing Network Model from the Database Design ............................ 69

3.4 Summary ............................................................................................................................ 72

CHAPTER 4 MODELLING THE TPC-C BENCHMARK .................................................................. 73
4.1 INTRODUCTION .................................................................................................................. 73
4.2 THE TPC-C BENCHMARK .................................................................................................. 73
4.3 THE TPCC-UVA IMPLEMENTATION ............................................................................... 77
4.4 BUILDING THE PERFORMANCE EVALUATION MODEL ................................................. 80
   4.4.1 Measuring DB Page Access Time ................................................................. 81
   4.4.2 Calculating Transaction Service Demands ....................................................... 81
   4.4.3 Building the Queueing Network Model ............................................................ 83
4.5 EXPERIMENTAL RESULTS .......................................................................................... 84
   4.5.1 Transaction Mean Response Time and Mean Throughput ..................................... 85
   4.5.2 Scalability .............................................................................................................. 87
4.6 A PERFORMANCE COMPARISON OF DIFFERENT DATABASE DESIGNS .................... 92
   4.6.1 The Database Design Descriptions ................................................................. 92
   4.6.2 Experimental Results ......................................................................................... 94
   4.6.3 Analysis ............................................................................................................... 98
4.7 SUMMARY ...................................................................................................................... 103

CHAPTER 5 MODELLING ACTIVE DATABASE RULES .......................................................... 105
5.1 INTRODUCTION .............................................................................................................. 105
5.2 MODELLING ACTIVE DATABASE RULES .................................................................... 106
5.3 EXTENSION OF THE FORMAL SPECIFICATION FOR TRIGGERS ............................... 108
   5.3.1 Trigger Formal Specification ............................................................................. 109
   5.3.2 Calculating Service Demands for Transactions that Invoke Triggers .................... 110
   5.3.3 Calculating the Routing Path ............................................................................. 114
5.4 TPCC-UVA TRIGGER PERFORMANCE MODELLING .............................................. 118
   5.4.1 Experimental Results ......................................................................................... 121
5.5 SUMMARY ...................................................................................................................... 127

CHAPTER 6 MODELLING REFERENTIAL INTEGRITY ...................................................... 128
6.1 INTRODUCTION .............................................................................................................. 128
6.2 MODELLING REFERENTIAL INTEGRITY CHECKING ............................................... 129
6.3 EXTENSION OF THE FORMAL SPECIFICATION FOR FOREIGN KEYS ................................................. 132

6.3.1 Referential Integrity Formal Specification ................................................................................... 132

6.3.2 Calculating Service Demands for Transactions that Invoke Referential Integrity Checks 135

6.3.3 Calculating the Routing Path ...................................................................................................... 138

6.4 TPCC-UVA FOREIGN KEY PERFORMANCE MODELLING .......................................................... 142

6.4.1 Experimental Results ............................................................................................................... 144

6.5 SUMMARY ...................................................................................................................................... 148

CHAPTER 7 CONCLUSIONS AND FUTURE WORK ................................................................................. 150

7.1 MAIN CONTRIBUTIONS ................................................................................................................ 150

7.2 FUTURE WORK .............................................................................................................................. 152

APPENDIX A: THE TPC-C TRANSACTION SPECIFICATION ............................................................. 154

A.1 THE NEW-ORDER TRANSACTION ............................................................................................... 154

A.2 THE PAYMENT TRANSACTION ...................................................................................................... 155

A.3 THE ORDER-STATUS TRANSACTION ............................................................................................ 157

A.4 THE DELIVERY TRANSACTION ...................................................................................................... 158

A.5 THE STOCK-LEVEL TRANSACTION .............................................................................................. 158

APPENDIX B: THE TPCC-UVA TABLE SPECIFICATIONS ..................................................................... 159

APPENDIX C: THE TPCC-UVA TRANSACTION SQL SOURCE CODE AND SERVICE DEMAND CALCULATION ......................................................................................................................... 167

C.1 CALCULATION OF TPCC-UVA INDEX I/O COST ........................................................................... 167

C.2 THE NEW-ORDER TRANSACTION ................................................................................................ 171

C.3 THE PAYMENT TRANSACTION ...................................................................................................... 173

C.4 THE ORDER-STATUS TRANSACTION ............................................................................................ 176

C.5 THE DELIVERY TRANSACTION ...................................................................................................... 177

C.6 THE STOCK-LEVEL TRANSACTION .............................................................................................. 178
List of Figures

Figure 2.1 An entity-relationship model................................................................. 11
Figure 2.2 A relational logical database design. .................................................... 12
Figure 2.3 A physical database design................................................................. 13
Figure 2.4 A (a) relational algebra expression and (b) equivalent query tree. .......... 15
Figure 2.5 An example of a queueing network. ..................................................... 19
Figure 2.6 A representation of (a) the black box model and (b) the transaction processing model. 25
Figure 2.7 A representation of the transaction phase model, where $p_i$ is the probability of a transaction moving from phase $i$ to phase $i+1$. ................................................................. 28
Figure 2.8 Sevcik’s database system performance evaluation framework.................. 35
Figure 2.9 CLISSPE specification of a simple client/server application. .................. 40
Figure 2.10 A queueing network model for a database system using the methodologies in the literature.................................................................................................................. 43
Figure 2.11 An example of (a) a resource usage profile and (b) the corresponding queueing network for two transaction classes. ......................................................................................... 45
Figure 3.1 Details of New_Emp and List_Emp transactions.................................... 62
Figure 3.2 A queueing network model for the database design example. ................ 62
Figure 4.1 The TPCC-UVA architecture. .............................................................. 78
Figure 4.2 TPCC-UVA queueing network model.................................................. 84
Figure 4.3 Comparison of the New-Order transaction mean response time per minute for a
measurement interval of (a) 2 hours (b) 4 hours and mean throughput per minute for a
measurement interval of (c) 2 hours (d) 4 hours for 100x2 clients. ......................... 86
Figure 4.4 Comparison of the New-Order transaction mean response time per minute for different number of clients. .......................................................................................... 89
Figure 4.5 Comparison of the Payment transaction mean response time per minute for different number of clients. .......................................................................................... 89
Figure 4.6 Comparison of the Order-Status transaction mean response time per minute for different number of clients. ................................................................. 90

Figure 4.7 Comparison of the Delivery transaction mean response time per minute for different number of clients. ........................................................................ 90

Figure 4.8 Comparison of the Stock-Level transaction mean response time per minute for different number of clients. ................................................................. 91

Figure 4.9 New-Order transaction mean response time per minute for a ramp-up period of 20 minutes and measurement interval of 4 hours for 100x2 clients for the I_1 database design. The TPCC-UVA system starts to stabilize 80 minutes into the measurement interval, i.e. 100 minutes from the beginning of the system run. ......................................................... 95

Figure 4.10 New-Order transaction mean response time per minute for a ramp-up period of 30 minutes and measurement interval of 7 hours for 100x2 clients for the I_3 database design. The TPCC-UVA system starts to stabilize 140 minutes into the measurement interval, i.e. 170 minutes from the beginning of the system run. ......................................................... 95

Figure 4.11 Comparison of the New-Order transaction mean response time per minute for 100x2 clients for the design (a) I_1 (b) I_2 (c) I_3................................................................. 96

Figure 4.12 Comparison of the Payment transaction mean response time per minute for 100x2 clients for the design (a) I_1 (b) I_2 (c) I_3................................................................. 97

Figure 4.13 Comparison of the Order-Status transaction mean response time per minute for 100x2 clients for the design (a) I_1 (b) I_2 (c) I_3................................................................. 97

Figure 4.14 Comparison of the Delivery transaction mean response time per minute for 100x2 clients for the design (a) I_1 (b) I_2 (c) I_3................................................................. 97

Figure 4.15 Comparison of the Stock-Level transaction mean response time per minute for 100x2 clients for the design (a) I_1 (b) I_2 (c) I_3................................................................. 97

Figure 4.16 DB page access trace for 100x2 clients for the design (a) I_1 (b) I_2 (c) I_3................................................................. 99

Figure 4.17 CUSTOMER table DB page access trace for 100x2 clients for the design (a) I_1 (b) I_2 (c) I_3................................................................. 100

Figure 4.18 The effect of large values on the mean of a population................................................. 102

Figure 5.1 A queueing network model with trigger invocations. .................................................. 108
Figure 5.2 A BEFORE trigger invocation. ................................................................. 111

Figure 5.3 New-Order transaction mean response time per minute for a ramp-up period of 20
minutes and measurement interval of 480 minutes for 100x1 clients. ........................ .... 118

Figure 5.4 Details of trigger1: AFTER UPDATE trigger on HISTORY. .......................... 119

Figure 5.5 Details of trigger2: AFTER INSERT trigger on HISTORY .............................. 120

Figure 5.6 TPCC-UVA queueing network model with ORDERCopy table. .................................. 121

Figure 5.7 New-Order transaction mean response time per minute for a ramp-up period of 20
minutes and measurement interval of 480 minutes for 100x2 clients. The TPCC-UVA system
with trigger1 starts to stabilize 140 minutes into the measurement interval, i.e. 160 minutes
from the beginning of the system run. ................................................................................. 123

Figure 5.8 New-Order transaction mean response time per minute for a ramp-up period of 20
minutes and measurement interval of 300 minutes for 100x2 clients. The TPCC-UVA system
with trigger2 starts to stabilize 130 minutes into the measurement interval, i.e. 150 minutes
from the beginning of the system run. ................................................................................. 123

Figure 5.9 Comparison of the New-Order transaction mean response time per minute for the
TPCC-UVA with (a) trigger1 and (b) trigger2 designs for 100x2 clients and measurement
interval of 120 minutes. ........................................................................................................ 125

Figure 5.10 Comparison of the Payment transaction mean response time per minute for the TPCC-
UVA with (a) trigger1 and (b) trigger2 designs for 100x2 clients and measurement interval of
120 minutes. ............................................................................................................... 125

Figure 5.11 Comparison of the Order-Status transaction mean response time per minute for the
TPCC-UVA with (a) trigger1 and (b) trigger2 designs for 100x2 clients and measurement
interval of 120 minutes. ........................................................................................................ 125

Figure 5.12 Comparison of the Delivery transaction mean response time per minute for the TPCC-
UVA with (a) trigger1 and (b) trigger2 designs for 100x2 clients and measurement interval of
120 minutes. ............................................................................................................... 126

Figure 5.13 Comparison of the Stock transaction mean response time per minute for the TPCC-
UVA with (a) trigger1 and (b) trigger2 designs for 100x2 clients and measurement interval of
120 minutes. ............................................................................................................... 126
Figure 6.1 A queueing network model with IMMEDIATE referential integrity checking. .......... 131
Figure 6.2 A queueing network model with DEFERRED referential integrity checking. .......... 132
Figure 6.3 Details of the foreign key constraint on the ORDER-LINE table. ................................. 142
Figure 6.4 TPCC-UVA queueing network model with ITEMCopy table. ....................................... 144
Figure 6.5 New-Order transaction mean response time per minute for a ramp-up period of 20
minutes and measurement interval of 480 minutes for 100x2 clients. The TPCC-UVA system
starts to stabilize 120 minutes into the measurement interval, i.e. 140 minutes from the
beginning of the system run. ........................................................................................................ 145
Figure 6.6 Comparison of the (a) New-Order (b) Payment (c) Order-Status (d) Delivery (e) Stock-
Level transactions mean response time per minute for a measurement interval of 120 minutes
for 100x2 clients. .......................................................................................................................... 147
Figure D.1 QNAP2 description of the TPCC-UVA clients. ............................................................ 180
Figure D.2 QNAP2 description of the TPCC-UVA queueing network servers. .............................. 182
List of Tables

Table 3.1 I/O DB page cost model for SQL operations................................................................. 57
Table 3.2 Mapping between database designs and queueing network models. ..................... 61
Table 3.3 Service demands for the New_Emp and List_Emp transactions. ........................... 63
Table 3.4 Formal specification notation .................................................................................. 64
Table 4.1 Summary of the TPC-C benchmark transactions....................................................... 75
Table 4.2 Scaling requirements for the TPC-C database......................................................... 76
Table 4.3 Initial loading size for the TPCC-UVA queueing network model. ......................... 82
Table 4.4 Number of I/O DB pages for the TPCC-UVA transactions.................................. 82
Table 4.5 Comparison of transaction mean response times for different number of clients..... 91
Table 4.6 Number of I/O DB pages for the TPCC-UVA transactions.................................... 94
Table 4.7 Comparison of transaction mean response times for different designs............... 98
Table 5.1 Number of I/O DB pages for the TPCC-UVA transactions.................................... 121
Table 5.2 Comparison of transaction mean response times for TPCC-UVA with trigger1 and
 trigger2 designs...................................................................................................................... 126
Table 6.1 Number of I/O DB pages for the TPCC-UVA transactions.................................... 143
Table 6.2 Comparison of transaction mean response times for the TPCC-UVA design with foreign
 key referencing....................................................................................................................... 148
Table 6.3 Transaction mean response times for the TPCC-UVA design with foreign key referencing
 the ITEM table and TPCC-UVA original design.................................................................. 148
Table C.1 Calculation of the TPCC-UVA index fan-out......................................................... 169
Table C.2 Partial calculation of the TPCC-UVA index I/O cost............................................. 170
Chapter 1 Introduction

The recognition of the need for detailed software design representation and performance evaluation has been the catalyst for the development of various software performance engineering methodologies [9, 64, 97, 98]. Furthermore, it has been established that for software system performance modelling to give accurate predictions, more comprehensive performance models need to be developed that provide for detailed software design modelling and evaluation [91, 97, 98]. The importance of performance evaluation of software systems at early design phases was initially proposed by Lazowska et al. [55] using software level queueing network performance models. Lazowska’s ideas were further developed by Smith [97] into the software performance engineering methodology.

Since Smith introduced the software performance engineering methodology the evaluation of software performance during the software lifecycle has not been uniformly adopted by the software engineering community [64], mainly due to the academic nature of performance models which do not appeal to software engineers in industry [81]. Hence, for a performance model to be usable, it must be able to reflect the application domain in such a way that it can be seamlessly integrated into the software development lifecycle.

This thesis provides a performance evaluation methodology for database designs that is more closely aligned with the database design domain. This is to provide for a more applicable performance model that provides more relevant feedback to database
designers and can be straightforwardly integrated into the database development lifecycle.

1.1 Motivation

A necessary part of any computer, communication network or system is information storage and retrieval. Jagadish et al. [48] have recognized that even though the majority of the available data resides in unstructured and semi-structured formats outside of traditional databases, databases are still the superior technology for data and information storage and retrieval [48]. Databases form the backend of online transaction processing systems, service-oriented architectures and multi-tier applications [11], all with critical performance requirements.

In more recent trends in database systems, the Internet has allowed for the outsourcing of database applications as a service and for multi-tenant architectures in which multiple businesses are consolidated onto the same physical database [6]. These new architectures lend to more complex database schemas and designs, and thus more complex performance problems. Therefore, databases that perform efficiently and which are matched to the user demands are crucial to the performance of these systems as a whole. It is apparent that the design of efficient databases, optimised to meet the projected traffic demands becomes a crucial part within the overall design process of any computer system.

Jagadish et al. [48] have considered the recent decline in the utilization of database technology, in comparison to the success of search engines, a symptom of database
usability and not database obsolescence. Given that the performance of the database affects the performance of the applications that depend on it [104], the performance of the database system accordingly affects its usability. Thus, a method in aiding in the design of more responsive databases is needed.

Moreover, the performance tuning of database systems in industry is a practical and complex problem, involving the combined knowledge of the database system design and operation, the underlying database management system (DBMS) and its functionality, the operating system, and the underlying hardware platform [28, 95]. Performance tuning is a major contributor to the total cost of ownership of database systems [48, 118]. The complexity and significance of database system performance tuning has led commercial database vendors to develop a number of database system performance tuning prototypes for the major DBMS [2, 15, 28, 123]. Hence, the problem is a current and significant one.

The software performance evaluation literature has expanded greatly in the last decade, with the majority of the software performance models targeted at the software architecture level of systems [9]. However, work in performance evaluation of database systems has been limited [104]. In most analytical models for database systems, it is assumed that the number of times a transaction visits the system resources and the distribution of service times at each station can be measured directly. Unfortunately, this is easier said than done, leading to complications in implementing the performance models in early system design phases. Moreover, the main emphasis of these models is capacity planning [1, 20, 43, 67, 89, 92], and in so, the feedback is not relevant to the
In this thesis, we suggest that the more natural the specification of the service demands and their ability to map to the original design, the easier it is to specify these measurements. Additionally, this thesis has recognized the need for a special-purpose performance evaluation method for database designs. The method should take into account the changing state of the database system, the relationships between the different structures of the database design, and the granularity of the expected feedback. These performance models can be used by the database designer for evaluating different design options before system implementation, in determining the configuration of the system to meet user needs after deployment, and for post-deployment performance tuning.

Furthermore, we note that the performance evaluation of many software and hardware architectures is based on the use of queueing network models [9] and this suggests that the database design performance evaluation method should also be similarly based. This will allow, in the future, for the two performance models to be integrated into a single queueing network model which combines both the hardware architecture and its associated database systems.

This thesis contributes a database design performance evaluation model within a queueing network environment; however, at a finer granularity which allows the database design constructs to be modelled and evaluated. This is a radical departure from previous database design performance methods, which consider the database design only in terms of processing demands on the hardware architecture [1, 20, 43, 67,
89, 92]. By using queueing networks to model the dynamic behaviour of the database design, the database designer can evaluate the expected performance of the design, before the physical deployment of the database system.

1.2 Objectives

The objective of the research presented in this thesis is to develop a novel performance evaluation methodology for database system designs based on queueing networks. This methodology should encompass the following aspects:

- provide performance feedback at a granularity that is relevant to database designers;
- be applicable to the performance evaluation of database designs at design time;
- provide a simple formulation to map database system design specifications to queueing network models;
- have the ability to model modern DBMS functionality, i.e. active database rules and referential integrity.

1.3 Contributions

The contributions of this thesis are:

The methodology allows for the modelling of detailed database designs improving over previous methods in the literature through: (1) the modelling of the transaction processing on database tables, (2) the incorporation of active database rules and referential integrity, and (3) the fact that detailed knowledge and performance modelling of the hardware architecture is not required. This makes the methodology more applicable for use by database designers in comparison to previous approaches.

- A formal specification of the methodology providing: (1) a description of database designs and transactions, (2) an algorithm to map database designs to queueing network models, and (3) an algorithm to extract transaction routing probabilities for the queueing network model.

- A categorization of transaction modelling in queueing network performance models for database systems and DBMS components. This categorization classifies the models in the literature based on the level of detail of the representation of the internal details of the database transaction.

- A justification for the exponential service time assumption for transactions in queueing network models in which transaction details are represented.

### 1.4 Thesis Outline

The rest of the thesis is structured as follows:
Chapter 2 presents background material in database design terminology and in queueing networks. A categorization of transaction modelling in database system queueing network models is presented with examples from the literature. In addition, previous methodologies of database system performance models and their shortcomings are discussed.

Chapter 3 introduces our approach for the modelling of database designs using queueing networks. The details regarding the steps to apply the method are given along with a formal specification of the transformation of database designs to queueing networks.

Chapter 4 details the modelling of the TPC-C benchmark using our modelling technique. Results are presented for the comparison of the model with the TPCC-UVA open source implementation of the TPC-C benchmark. In addition, a comparison is conducted between different database designs.

Chapter 5 presents the modelling of active database rules. The formal specification of Chapter 4 is extended to include active database rules. The extended model is validated by comparing its results with a modified version of the TPCC-UVA implementation that incorporates active database rules.

Chapter 6 details the modelling of referential integrity constraints using queueing networks. The formal specification of Chapter 4 is extended to incorporate database tables with referential integrity constraints. A comparison is conducted with a modified version of the TPCC-UVA implementation with referential integrity constraints.
Chapter 7 concludes the thesis by summarizing and discussing the contributions, with a discussion of future work.

Appendices A, B and C provide details of the TPC-C benchmark and the TPCC-UVA implementation.

Appendix D gives examples of the QNAP2 simulation model descriptions.
Chapter 2 Background and Related Work

2.1 Introduction

In this Chapter, the context for the thesis is set by reviewing the database and DBMS queueing network performance evaluation models in the literature. The Chapter begins with an overview of database design terminology and a brief discussion on database system performance tuning. Then, queueing networks and their applicability to software system performance evaluation are discussed. A categorization of database transaction modelling representations in the literature is presented, followed by a justification of the exponential service time assumption for transactions in database and DBMS queueing network models. The performance evaluation methodologies targeted at database system performance evaluation are detailed and their shortcomings are discussed. Finally, the Chapter concludes with the justification of our modelling approach for database designs and database systems and our contributions.

2.2 An Overview of Database Design Concepts

Databases (DB) are used to store collections of related data. Database management systems (DBMS) are the underlying runtime environment for a database. A DBMS provides a high-level language to define the structure of the data; known as the data
definition language (DDL). In addition, DBMS have high-level languages to access and modify data in the database; this is the data manipulation language (DML). The standard DML is the Structured Query Language (SQL) [47], which is based on relational calculus [32]. Database access entails either: a request for data, i.e. a SQL SELECT statement or a modification of the data, i.e. SQL INSERT, UPDATE or DELETE statements. Programs that access the database are called transactions and are written in a data manipulation language such as SQL or in a procedural language with SQL extensions. Transactions are executed by the DBMS as one atomic unit.

Database designers are assigned the task of transforming an enterprise’s data from its external representation in the real world to a representation that can be stored in the database. The first step in this transformation is the conceptual data model of the database. The conceptual data model represents real-world data using the entity-relationship (E-R) model or UML class diagrams [51, 69, 84]. At this stage, the data is represented as entities with attributes. In addition, relations between the different entities are represented. For example, if the entities are employees and departments, then the attributes of an employee would include his/her name and employee number. A relationship would be Works-In, which is an employee working in a certain department. This is illustrated in Figure 2.1. Details of entity-relationship diagrams and their constraints are in [51, 69, 84].
The next step is the logical database design, which describes the logical schema of the database. The logical schema is the transformation of the conceptual data model to a logical data model, which constitutes the data model of the database. In this work we refer only to the relational data model and relational databases. A relational database consists of relations or tables, which are constructed from records/rows and fields/columns: these terms will be used interchangeably. Each column has a specific domain which specifies the data type that can be stored in the column. Columns can have constraints, e.g. a column that uniquely defines a row is a primary key, and a column that has values related to a primary key of another table is a foreign key. A foreign key is known as a referential integrity constraint. Figure 2.2 gives an example of a relational model for the employee and department data. Properties of keys under the relational model are rigorously defined in [29].

Figure 2.1 An entity-relationship model.
The next stage is the physical database design, where the logical database model is extended to describe the physical storage of the database [51, 84]. The physical database design includes modelling the following [51, 69, 84]:

- the data table’s *indexes*, which are auxiliary data structures that support rapid access to the rows of a table;

- database and data table *partitions*; i.e. dividing a table or database into multiple physical data files or locations;

- DBMS *schemas, tablespaces*, which are the logical grouping of database tables;

- data files, which are the physical storage files of the database schemas and tablespaces;
• any additional properties of the DBMS chosen for deployment, i.e. *views* and *triggers*. Views are stored SQL SELECT statements that are automatically computed when a view is referenced. Triggers are event-condition-action rules that monitor the occurrence of a database event, e.g. an update of a certain column, and execute the given action if their condition evaluates to *true*. Triggers are also known as *active database rules*.

Figure 2.2 illustrates an index and a trigger on the Employee table, and a view on the Employee and Department tables. Figure 2.3 shows a physical design of a database with a single schema, and two tablespaces, where each tablespace is stored on one or two physical data files. The database is partitioned over two physical disks.

![Database Diagram](image)

**Figure 2.3** A physical database design.

When an SQL query is submitted to the DBMS, it is executed in the following steps [32]:

```sql
[SQL query]
```
Translating SQL queries into relational algebra expressions: The SQL query is parsed and then validated against the definition of the database schema by checking that all relation and attribute names are correct and semantically meaningful. The SQL query is then translated into its equivalent relational algebra expression. This is further transformed into a query tree data structure. A query tree represents the input relations of the SQL query as leaf nodes and the relational algebra operations as internal nodes. An execution of the query tree consists of executing an internal node when its operands are available and then replacing the internal node with the result relation. The query tree execution terminates when the root node is executed and produces the result of the query. Figure 2.4 details a relational algebra expression and query tree for the following SQL query based on the database design of Figure 2.2:

```
SELECT emp_no, emp_name FROM EMPLOYEE, DEPARTMENT
WHERE dept_name = 'HR' and
    DEPARTMENT.dept_no = EMPLOYEE.dept_no
```

From Figure 2.4, the $\pi$ symbol represents the relational algebra *project* operation, which picks out a subset of columns from the table. The $\sigma$ symbol represents the *select* operation, which selects a subset of the rows of a table based on a certain condition. The JOIN operation concatenates rows of two or more tables usually based on equal values in the JOINed columns. The result of a relational algebra operation is the *result relation* or table. The query tree represents an initial representation of the relational algebra expression.

Query optimization: Next, the DBMS *query optimizer* transforms the initial query tree to an equivalent more efficient query tree, known as the optimized query tree. Heuristic
optimization rules are applied in this transformation. Examples of optimization rules include: perform a `select` before a `JOIN` or use indexes before table scans. The rules take into account the expected relational algebra operation result size and aim to produce the smallest result set possible for each relational algebra operation. Figure 2.4(b) is the optimized query tree for the previous SQL query.

\[
\Pi_{\text{emp_no, emp_name}} (\sigma_{\text{dept_name} = \text{HR}}(\text{DEPARTMENT}) \ JOIN \ \text{department}.\text{dept_no} = \text{Employee}.\text{dept_no} \ \text{EMPLOYEE})
\]

(a)

<table>
<thead>
<tr>
<th>\Pi_{\text{emp_no, emp_name}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\JOIN \text{department}.\text{dept_no} = \text{Employee}.\text{dept_no}</td>
</tr>
<tr>
<td>\sigma_{\text{dept_name} = \text{HR}}(\text{DEPARTMENT})</td>
</tr>
<tr>
<td>\sigma_{\text{dept_name} = \text{HR}}(\text{EMPLOYEE})</td>
</tr>
<tr>
<td>\text{DEPARTMENT}</td>
</tr>
</tbody>
</table>

(b)

**Figure 2.4** A (a) relational algebra expression and (b) equivalent query tree.

**Access plan:** The optimized query tree is used to prepare an execution or access plan. The access plan for a relational algebra expression is the optimized query tree with information about the access methods (e.g., indexes) available for each relation and the algorithms used to compute the relational operators. For the query tree in Figure 2.4, each relational algebra operation is annotated with the access methods for each relation, e.g., a table scan for the DEPARTMENT table since there is no index to access the dept_name field.
**Execution:** The access plan is converted to executable code and run against the database to produce the results of the SQL query.

More details on database design and DBMS functionality can be found in [51, 84]. Details of relational algebra, query trees and optimizing heuristics are in [32].

### 2.3 Database System Performance Tuning

Database system performance tuning is a post deployment activity performed by the database administrator (DBA). Performance problems of database systems manifest themselves in query and transaction response time. Hence, the performance tuning effort is concentrated on the database and the transactions accessing the database.

Database performance tuning is a difficult task in that the DBA must be familiar with the database system design and operation, the underlying DBMS and its functionality, the operating system, and the hardware platform that runs the database system [28, 94]. It is the major contributor to the total cost of ownership of database systems [48, 118].

Performance tuning of database systems involves one or all of the following activities [51, 84, 94]:

- restructuring of high-use SQL statements and transactions: avoiding costly access plans due to faulty statement structures;

- index tuning: adding or removing indexes, changing index types or redesigning
existing indexes;

- table redesign: decomposing a table into smaller tables (normalization) or collecting small tables into one large table (de-normalization);

- tuning concurrent access: reducing table and row lock contention and eliminating hot spots;

- physical placement of data on disks: e.g. distributing heavily used objects on different disks or nodes;

- optimizing DBMS data blocks, query optimization and buffer management techniques, and utilizing DBMS specific extensions and features [16];

- evaluating the amount of physical resources available to the database system, e.g. CPUs, disks, main memory, etc.

It is important to note that database performance tuning must go hand in hand with the specification of the application that uses the database; the application may uphold certain constraints depending on the results given by a certain SQL statement or may depend on certain table structures. Any changes to the SQL statements or table structures that are not reflected in the application may produce unexpected results.

The complexity and significance of database system performance tuning has led commercial database vendors to develop a number of database system performance tuning prototypes for the major DBMS [2, 15, 28, 123]. These tools rely on query
optimizers and statistics from production databases to give recommendations for performance improvements, e.g. SQL statement restructuring, index and view recommendations, and table partitioning [2, 15, 28, 123].

As indicated above, optimal performance improvement of the database is achieved through redesigning artefacts of the logical and physical database designs, through either manual procedures or automated tools. Furthermore, the coupling between the application program and the database inevitably means an understanding of the application design is needed for effective database tuning.

This being stated, the logical and physical database design structures are the main contributors to DB system performance problems. Therefore, an early evaluation of their performance at DB system design time, coupled with the knowledge of the application design, would be an effective factor in the reduction of post deployment database tuning.

2.4 Queueing Networks

Queueing networks were initially used in operational research to predict the performance of customer-facing systems. They have been successfully applied for the performance evaluation of computer and telecommunication systems [52-55, 116], and software system designs [9, 65, 97].

Queueing networks are a collection of individual queues or queueing stations; they are based on the concept of modelling contention in resource sharing systems. In a queue,
the shared resource is represented as a *server* that provides services to waiting *customers* or *jobs*. Customers requesting service wait for the server to be free by queueing at the server. Depending on the *queueing (scheduling) discipline*, a customer enters for service after the server completes its last job. After completing service, according to the *routing probabilities*, this customer then leaves the server to queue for another server, re-enters the same server again for more service, or leaves the network completely (Figure 2.5).

![Figure 2.5 An example of a queueing network.](image)

Some common queueing disciplines are [52]: *first-come-first-served (FCFS)*, which serves customers based on the order of arrival; *last-come-first-served (LCFS)*, in which the last arriving customer is served first; *priority queueing*, in which the customer with the highest priority is served first; *processor sharing (PS)*, which shares the server capacity with all the waiting customers in parallel and *random service*, in which customers are chosen for service at random.

The types of queueing networks are: *open* queueing networks, which are characterized
by customers arriving from outside the queueing network and after receiving service depart from the network. In closed queueing networks, customers are resident in the network; the number of customers in the network is always constant. Mixed queueing networks have customers of both types.

The behaviour of the queueing network and the behaviour of customers receiving or waiting to receive service in the queueing network are characterized by: (1) the arrival rate of customers to the queueing network (open queueing network); (2) the client think time – the time it takes a client to send a request to the queueing network (closed queueing network); (3) the statistical distribution of service times for each customer; if this distribution varies between different groups of customers then the customers are grouped into customer classes and service distributions are specified by class; (4) the maximum queue or buffer size, which is the maximum number of customers allowed to wait to be serviced. Kendall’s notation [50] is usually used to specify these characteristics and when using this a queue is represented as $A/S/c/m/N$, where:

- $A$ is the customer inter-arrival time distribution, e.g. $M$ for memoryless/exponential, $G$ for general, and $D$ for deterministic interarrival distributions.

- $S$ is the service time distribution, which can also be $M$, $G$, or $D$ distributions.

- $c$ is the number of servers providing service for the customers in the queue.

- $m$ is the queue or buffer size, which is the maximum number of customers
allowed to wait for service. It can also represent the maximum number of customers allowed in the queue plus the customer in service. The default buffer size is an infinite buffer.

- \( N \) is the maximum number of customers in the system; the default is an infinite number of customers.

Solving a queueing network model analytically or by simulation gives feedback on system performance, e.g. customer response time, throughput and waiting time. These performance measures are calculated on the assumption that the queueing network is in equilibrium (a steady state), i.e. the rate of arrivals to the queueing network is equal to the rate of departures from the network. A large class of queueing networks, known as \textit{product form queueing networks} [8], have direct analytical solutions based on the parameters of the queueing network.

Extended Queueing Network (EQN) models [12] have been introduced to incorporate features of actual computer systems in classical queueing network models, e.g., synchronization, concurrency and simultaneous resource possession. Another extension to classical queueing networks are Layered Queueing Network (LQN) models [87], which allow for the modelling of software and hardware contention in distributed and multiprocessor systems.

\textbf{2.4.1 Queueing Networks for Software Performance Evaluation}

The use of queueing networks and EQNs has been prevalent in software performance evaluation methodologies [4, 5, 9, 25, 27, 79, 80, 97, 98]. This is essentially due to
their widespread use in computer system modelling and to their natural mapping onto software system design phase artefacts, especially software architecture components [4, 5, 9, 26, 79, 119]. However, it has been noted that queueing networks are not powerful enough to model late software lifecycle details [26], e.g. component processing details.

2.5 Performance Evaluation of DBMS Components

Performance evaluation studies of databases concentrate on the fundamental components of the DBMS [104], e.g. the evaluation of storage and buffer management techniques [61, 121], query processing [46] and optimization [22, 93], transaction locking [37] and recovery algorithms [36], and index structures and their utilization [40, 45, 101]. There have also been studies of the performance analysis of unique characteristics of different database models, e.g. object-oriented [14, 100], distributed [70] and active [18] databases, and the assessment of commercial [3] and open source [60] DBMS. Recently, studies have been conducted on the performance evaluation of database system application architectures, i.e. multi-tiered Internet applications with back-end databases [86, 117]. Analytical modelling, simulation or empirical experiments are the main methodologies used for performance evaluation. Thomasian has documented performance evaluation models of database system components [104] and concurrency control mechanisms in database systems [103], while Nicola and Jarke [70] have surveyed performance models of distributed and replicated databases.

Even though the performance of the database system and DBMS components have been extensively studied, studies of overall database system performance are very limited
2.6 A Categorization of Transactions in Database Performance Models

Analytical models of database systems or DBMS components, as is the convention for all analytical models, are basically identified by the phenomena they study, e.g. concurrency control, or the solution method used, e.g. hierarchical or recursive. In an overview of the literature of performance modelling of databases and DBMS components, we have identified a set of techniques in representing databases and database transactions in analytical queueing models.

We have restricted our overview of the literature to performance evaluation studies that model some form of interaction between transactions and a database. Models in which a specific aspect of databases or DBMS, e.g. buffering algorithms, is studied in isolation are outside the scope of this categorization as they do not constitute complete interactions with a database. The objective of this categorization is to define the trends in modelling and defining transactions in queueing models for database and DBMS component performance evaluations. To the best of our knowledge, no such classification currently exists in the literature.

The queueing network performance evaluation studies in the literature have been classified based on the level of detail in which the database transaction's internal design is represented in the models. We have identified four distinct categories of transaction representation which will be referred to as follows: the black box model, the
transaction processing model, the transaction size model and the transaction phase
model. A description of each category follows with the relevant studies from the
literature.

The following convention is used when describing the studies: we assume a database \( D \) with \( d \) database objects and is accessed by \( T = \{T_1, T_2, \ldots, T_n\} \) transaction classes.

### 2.6.1 The Black Box Model

In the black box model, when the database \( D \) is a centralized database it is represented
as a single queueing node. In the case of a distributed database, it is represented as
multiple queueing nodes, where each node represents a distributed database site. The
internal design of the transaction is not represented in the queueing model, as the goal of
the performance evaluation is to represent the workload of the transactions at the system
level. Each transaction class \( T_i \) accessing \( D \) has an arrival rate \( \lambda_i \) and a service demand
\( \mu_i \) on the queueing node \( D \) (Figure 2.6 (a)).

It is common in analytical models for replicated and distributed databases to model the
database site as a single queueing service center and the transaction as a workload class
in the system [70], e.g. Baccelli and Coffman [7] use an M/M/m/FCFS queue to model
\( m \) local sites. Ciciani et al. [23, 24] study the effect of distributed concurrency control
protocols on replicated distributed databases by modelling each local database site as an
M/M/1 queue. For multi-tiered Internet applications with backend databases, Urgaonkar
et al. [117] describe a queueing network model in which each tier, including the
database tier, is represented as a processor sharing queue.

**Figure 2.6** A representation of (a) the black box model and (b) the transaction processing model.

### 2.6.2 The Transaction Processing Model

For the transaction processing model, the database $D$ is represented by the underlying hardware architecture using the central server model [17] or its variations. Each transaction class $T_i$ accessing $D$ is defined by its service demand on the hardware architecture and flows through the system probabilistically (Figure 2.6 (b)). For closed queueing networks, the number of transaction classes in the system is restricted by the maximum multiprogramming level. This model is used to represent a centralized database or a site in a distributed database.

We have found the transaction processing model to be the prevalent category used to represent centralized databases or the database tier in multi-tiered applications as detailed below. Menascé et al. [65] provide many examples of modelling centralized database servers using this model in which transaction classes are represented by their
service demands on the hardware resources.

Sheth et al. [96] evaluate the effect of networks delays on concurrency control protocols of distributed database systems by modelling each distributed site as a central server model with M/M/1/FCFS queues for the CPU, disk and network connections. A replicated database instance is represented as a central server model in [33], with additional delay centers to represent the distributed system functionality.

Menascé et al. [62, 66] represent an $n$-tier multithreaded server system modelled as a software contention model and a hardware contention model. The software contention model is modelled as a Markov chain representing the arrival and completion of jobs. The rate at which the jobs are completed is determined by the solution to the hardware contention model, which is a closed queueing network of a CPU and disk with $k$ jobs in the system. Gijsen et al. [35] model a multi-tiered application as an open queueing network with Poisson arrivals. The front-end application server is assumed to be the CPU and is modelled as a processor-sharing queue, while parallel access to the backend database is modelled as a multi-server FCFS queue with exponential service times.

Ceri et al. [21] study the performance of detached triggers in active databases. The scheduling and execution of jobs and detached triggers is modelled as a queueing network with one CPU and a set of homogeneous disks, where each disk represents a subset of the database. A dispatcher queue represents the scheduling of detached triggers. Triggers are activated probabilistically for each transaction in the system and arrive at the dispatcher when their corresponding transaction completes. The model
assumes Poisson arrivals for transactions and a constant arrival rate for batched jobs.

Moreover, the database system performance models discussed in Section 2.8 [1, 20, 43, 67, 89, 92, 112] all follow the transaction processing model.

2.6.3 The Transaction Size Model

The transaction size model describes each transaction class $T_i$ accessing $D$ based on the number of data objects $n_i$ it accesses from the $d$ database objects. The performance evaluation studies in this category are limited to concurrency control methods.

In [68, 83, 107], modelling of static exclusive locks in centralized databases is considered. A transaction class accesses a constant set $N$ of data items randomly chosen from the total number of data elements in the database. This set $N$, is used to calculate the locking conflict probabilities for transactions entering the system. For the hardware architecture, the CPU is represented with a processing sharing queue, exponential FCFS disk service rates and a maximum multiprogramming level. Thomasian and Ryu [108] extend the model in [107] to represent shared locks in addition to exclusive locks, with a variable number of locks per transaction class. Furthermore, Thomasian and Ryu [109] model two-phase dynamic locking using the same basic modelling assumptions for transaction classes and lock acquisition as in [108].

The static lock acquisition model in [68, 83, 107, 108] is probabilistic, i.e. the locks assigned to a transaction class are determined by a certain probability. However, Thomasian [105] proposes a deterministic lock acquisition model, in which the number of locks acquired by a transaction class are predetermined per class. This is closer to
actual transaction lock requests. Therefore, only compatible transaction classes accessing disjoint data objects can be processed concurrently. The queueing model used by Thomasian [105] is similar to [68, 83, 107, 108], with a fixed number of transaction classes that arrive with random frequencies.

Harder and Harrison [39] present an analytical model that combines traditional locks with Oracle Parallel Server locks. However, the transaction size is based on the number of locks acquired by the transaction class per database tablespace. Parameterization of the model is assumed to be from measurements of a real system.

### 2.6.4 The Transaction Phase Model

For the transaction phase model, each transaction class $T_i$ accessing $D$ is classified based on the number of phases it contains. Movement through the phases is probabilistic. Figure 2.7 illustrates the division of a transaction into $n$ phases.

![Figure 2.7](image)

*Figure 2.7* A representation of the transaction phase model, where $p_i$ is the probability of a transaction moving from phase $i$ to phase $i+1$.

Jenq et al. [49] present an analytical model of a distributed database testbed system in which transactions are divided into phases corresponding to the steps a distributed transaction needs to complete execution. A transaction moves from one phase to the next based on a *phase transition probability*. Transactions access database records
randomly and uniformly. Each transaction issues a fixed number of requests and each request accesses a fixed number of DB records. Lock requests are uniformly distributed over the lifetime of the transaction. Each distributed site is modelled as an extended central server model with delay centres representing synchronization delays.

Yu et al. [122] analyze routing in locally distributed databases. Transactions are divided into two phases: application processing, representing the front-end system and database request processing representing the mean distributed processing of the transaction database requests. The transactions are divided into classes based on their processing times in each phase. Each transaction class $C_i$ has a Poisson arrival rate $\lambda_i$ and an exponential I/O disk access service demand $\mu_i$, for each database request at a site. The transaction class starts in the application phase, then moves to the database request phase with a probability $p_k$ for each distributed site $D_k$ or leaves the network with probability $1- \sum p_k$. Each distributed node is represented by a single server processor-sharing queue for the CPU, and the I/O system is represented by an infinite FCFS queue.

Thomasian [106] studies the effect of checkpointing on transaction performance. Transactions are assumed to access $n$ data objects in $n+1$ steps; each step is preceded by an access to a data object. A transaction moves from one step to the next with a probability $p_i$, i.e. the probability of no data conflict or restart. The conflict probability is proportional to the number of objects the transaction has currently accessed. Access to data objects is uniform and the time to access a data object is assumed to be exponential. An analysis to determine the optimal number of checkpoints and their
position in transaction execution is given.

Sanzo et al. describe a performance analysis of locked-based concurrency control in [90]. The transaction data manipulations are modelled as a sequence of \( m \) execution phases, starting with a \textit{begin phase}, then \( m-2 \) write or read operations on the data items, and finally, a \textit{commit phase}. Each phase is assumed to use an exponentially distributed number of CPU instructions. This is to cater to the underlying M/M/k model for the CPU, where \( k \) is the number of CPU cores. The disk is assumed to have a fixed I/O delay. Transactions arrive according to a Poisson process. Lock holding time for each data item accessed by a transaction depends on the access order of that item, i.e. items accessed first will have a longer lock holding time than objects accessed last; lock holding time is also exponentially distributed. Movement between phases depends on the lock conflict probability of that phase, which is calculated based on the transaction access pattern. The transaction access pattern specifies the probability that the \( k^{\text{th}} \) operation accesses the \( i^{\text{th}} \) data item and that the probability of the operation is a \textit{write}.

### 2.6.5 Discussion

The previous classification mirrors the fact that the goal of the performance analysis dictates the detail at which the transaction is represented in the queueing network model. When the concern is to evaluate performance at the system level, in which transaction internal details are irrelevant or negligible in their effect on performance, the transaction is represented as a workload on the system. This is the main characterization for the black box model studies. Capacity planning performance studies represent the details of the hardware architecture being evaluated; hence the transaction is modelled
as its processing demands on the hardware architecture.

For the transaction size and phase categories, we notice the predominance of models that evaluate and study concurrency control in centralized or distributed databases. Here, the internal structure of the transaction affects the performance of the studied system; hence it must be taken into consideration in the queueing model.

In general, for a queueing model to be a realistic representation of a database system it is imperative that the model represents the details of the transactions that affect the performance.

**2.7 The Exponential Service Time Assumption**

The majority of distributed database queueing network models assume transaction service time is exponentially distributed at the distributed database site [70]. For models of centralized databases, we have found the majority assume transaction disk service time is exponential [33, 35, 62, 65, 66, 68, 83, 105, 107-109]. Transaction time to access a data object is assumed exponentially distributed in [106]. While [90] assumes that the number of CPU instructions per transaction phase and lock holding time are exponentially distributed. However, these models do not provide justification for this assumption in the context of database systems and transactions. In this section, we provide justifications for the exponential service time assumption in the context of database systems.

When transaction internal details and processing are not modelled in the queueing
network model, i.e. the black box category, the justification for the distribution of service times should be directed towards the overall expectation of the transaction workload. Nicola and Jarke [70] provide a justification for the exponential service time assumption for transactions when the database is represented as a single queueing node. They build on the expectation that transactions that access a small or moderate number of data objects occur more frequently than transactions referencing a large number of data objects. For this to hold, the number of data objects accessed by a transaction must follow a geometric distribution. Given that transaction service time is directly related to the number of data objects referenced, then transaction service time can be assumed to be exponentially distributed, which is the continuous equivalent of the geometric distribution.

However, when transaction internal details and processing are modelled, i.e. the transaction processing, size and phase categories, then justification of the exponential service times needs to be directed to disk or data object access time. This, for both cases, is the DB I/O page access time. Below we provide a justification for the exponential service time for transactions when the queueing network model represents the details of transaction processing.

The number of I/O DB pages to access a data object will depend on the type of the data object accessed and its access method. For example, if the data object accessed is a table, e.g. a full table scan, then the number of I/O DB pages can range from one page for a small table to a very large number of pages for a large table. Similarly, if the data object accessed is a row in a table, the number of I/O DB pages will depend on the index used and the type of query. This number can again range from one for a
random record with a precise index; to a small number for an inefficient index or a range search; to a large number when no index exists that satisfies the query. Clearly, a small number of DB pages are more common than a large number DB pages. Thus, the expected number of I/O DB pages will follow a geometric distribution. Hence, disk and data object access service times for a transaction can be approximated by the exponential distribution.

2.8 Database System Performance Evaluation Methodologies

Contrary to the vast amount of performance evaluation studies of individual DBMS components and constructs, there is a lack of research into the overall performance evaluation of database systems [104]. In this Section, we present database system performance evaluation methodologies found in the literature. The majority of these methodologies are based, intentionally or not, on Sevcik’s layered performance evaluation methodology [92], described in the following Section.

2.8.1 A General Framework for Database System Performance Prediction

To the best of our knowledge, the first approach which presented a performance evaluation methodology for database systems using queueing networks was introduced by Kenneth Sevcik [92]. Sevcik describes a framework for estimating workload characteristics of a database system as input parameters to a queueing network model. The framework was not directly related to a certain database model, but catered for the data models at that time – mainly hierarchical and relational data models [92]. The
framework proposed the use of information from the logical and physical designs of the database system, together with the characteristics of the DBMS to map the workload of a database system onto a queueing network model.

Sevcik divides his framework into layers that map directly onto the steps in designing and implementing a database system. The transformation from one layer to the next is based on a database design decision. The framework consists of the following six layers [92]: abstract world, logical database, physical database, data unit access, physical I/O access and device loading layers (Figure 2.8).

In the abstract world layer, the data entities are specified with attribute range, distribution and correlation. The transactions are denoted with frequency of occurrence and relationships to entities. In the logical database layer, the output of the previous phase is transformed to a representation depending on the choice of the logical data model and data manipulation language. In the context of the relational model, this would represent the data entities as relations with attributes and number of rows. For the transactions, the structure and data manipulation language constructs are specified, in addition to rates of occurrence and percentage of database entities used.
Next, in the physical database layer, indexes are specified and a linkage is established between relations and the physical files that hold the relations. Transactions are specified procedurally, and each transaction type is characterized by its pattern of access to the physical files.

In the data unit access layer, the physical database characteristics are transformed to be more similar to the characteristics of the input parameters of queueing network models. Relations and indexes allocated to the same physical data file are considered to be data

Figure 2.8 Sevcik’s database system performance evaluation framework.
units and their storage size is determined. Transactions are specified as task classes, characterized by arrival rates, average number of CPU instructions and average number of accesses to each data unit. In addition, other concurrent workloads on the hardware are classified as task classes. For all task classes, the degree of randomness of access to the data units is specified on a scale of zero to one.

Then, in the physical I/O access layer, data units are distributed over the physical devices and the service demands and arrival rates for each task class per physical device is determined. Next, in the device loading layer, the service times for each task class per physical device are calculated. Finally, the appropriate queueing network model is solved by using the final outputs of the previous layers. Calculations of service demands and service times depend on the environment of the database system, i.e. the hardware and software configurations and the DBMS query optimizer and buffer management strategies. The queueing network service demand distributions depend on the model used to represent the hardware architecture.

2.8.2 Methodologies Based on Sevcik’s Layered Approach

The application of Sevcik’s general framework was extended to other performance evaluation models based on his layering approach. These models are discussed next.

2.8.2.1 The Hierarchical DBMS Evaluation Model

The hierarchical DBMS evaluation model presented by Adams [1], is a five-layer hierarchy that describes workload acquisition and characterization for relational database systems. The five layers are: the enterprise, logical database design, logical
data organization, physical storage organization and underlying machine levels. With transformations similar to those in [92], the layered model presents the steps to gradually transform a database system description into input parameters for a queueing network model. The database system workload characteristics are mapped onto a queueing network model of the underlying hardware devices that the database system will run on.

2.8.2.2 The Prophet Model

Casas and Sevcik [20] describe the Prophet model, which is an extension of [92], as a layered database performance evaluation model proposed for general database data models. The model consists of four levels: semantic representation, schema database, internal database and system hardware. Workload characteristics for the database system are transformed from level to level to provide performance measures to evaluate design decisions. For the internal database layer a buffer management sub-model is used to determine buffer hit rates. It is assumed that database block references are Bradford-Zipf distributed [19] over the database blocks. The final stage is the performance evaluation of the input parameters for a queueing network model of the system hardware devices.

2.8.2.3 The MOSES Model and JOSHUA Prototype

Hyslop and Sevcik [43] propose an extension to [20], which is a layered model for the relational data model, with a tool supporting a relational query optimizer. The model, denoted as MOSES (Model of Sql-Equivalent Systems), was implemented using a prototype tool: JOSHUA. The MOSES model consists of seven layers based on the
layers in [92] and an extension of [20], but tailored to the relational data model. The seven layers are: the semantic model, schema database, query optimizer, internal database, physical I/O, resource allocation and concurrent processes. The tool supports a relational optimizer that transforms the relational algebra representation of the transactions into an optimized access path. The database system is transformed layer by layer to be mapped onto a queueing network model representing the device layout and communication channels. Parameters defining the buffer hit rate, the device speed and network latency are defined or calculated by the model. An analytical model is used to estimate lock conflicts and concurrency.

2.8.2.4 A Relational Database Performance Analysis Tool

Salza and Tomasso [89] present a methodology and tool for the cost and performance analysis of relational database applications. The methodology is based on specifying a static workload and a dynamic workload for a queueing network model of the hardware devices of a database computer system. The static workload consists of the database structure: the tables, their cardinality and attributes. The dynamic workload comprises the set of transaction types, their relative arrival rates, SQL definition, the selectivity of the SQL predicates and the transaction CPU and I/O overhead. The tool has a query optimizer simulator to calculate expected transaction demands and a buffering algorithm to compute transaction buffer hit rates. The resource demands of the transactions in terms of CPU and I/O demands are calculated and used as input to a queueing network model of the computer system. The queueing network model is a multi-class product-form open queueing network, with a customer class for each transaction type and service centers for the CPU and disks. Salza and Renzitti [88] conducted similar work
for parallel database systems.

2.8.2.5 CLISSPE: CLient/Server Software Performance Evaluation Tool

A more recent attempt to evaluate database system performance at a more detailed level is a method for performance evaluation of client/server architectures using queueing networks proposed by Menascé and Gomaa [67]. Client/server system performance is calculated by estimating transaction service demands on the database through calculating the amount of I/O perceived depending on the access path of the DBMS [67].

The workload characterization and service demands for a queueing network model are calculated by modelling the client/server system using the CLISSPE (CLient/Server Software Performance Evaluation) language [63]. The CLISSPE language provides the developer with the ability to specify hardware and software configurations and performance characteristics for clients, servers, networks, commercial DBMSs, relational database tables and transactions in the client/server system, which is systematically translated into service demands on the hardware resources [67]. Figure 2.9 gives an example of a CLISSPE specification for an Ethernet LAN with 100 clients, one server and the EMP table residing on an Oracle DBMS on the server.
Figure 2.9 CLISSPE specification of a simple client/server application.

Database tables are specified in the CLISSPE language, by declaring their structure, and their attribute range, selectivity, and cardinality, as well as the type of index on the table. Transactions are specified by describing their functionality, with specific commands for accessing and modifying database tables. The service demand for a transaction is the sum of the service demands of all the database commands it contains,
in addition to the service demands of the procedural commands. A built-in model of a DBMS query optimizer allows the CLISSPE compiler to estimate the service demands for database transactions [67].

In this method, the performance characteristics of the database are taken from the logical and physical designs. The CLISSPE language is able to model a simple database system: tables without hierarchical relationships and SELECT and simple UPDATE statements with indexes and access path calculations. Lock contention, active database rules and referential integrity constraints are not covered in the specification of the language.

2.8.2.6 Discussion

The previous performance evaluation methodologies are based on the same methodology for estimating workload characteristics from a database system for use as input parameters to a queueing network model, which represents the physical hardware configuration of the final system. These methodologies provide for simple database designs, mainly due to historical reasons [1, 20, 43, 92] (the methodologies were proposed before the maturity of the relational model), or due to lack of representation of modern DBMS functionality [67, 89], e.g. referential integrity and active database rules.

The main objective that defines the aforementioned performance evaluation tools and methodologies is the definition of a technique to extract performance parameters for queueing network models from the characteristics of database transactions and tables. The concern is in providing these parameters for a queueing network model that represents the hardware architecture of the evaluated system. The consequence is that
performance problems are identified on hardware devices, thereby giving a general indication to the database designer of where the problem is: e.g. the bottleneck is on Disk2, therefore review all transactions that access Disk2. This information, however, is not beneficial to the database designer during the actual design process.

Furthermore, to explain these concepts, assume we have a database system composed of two classes of transactions: $t_1$ and $t_2$. Transaction class $t_1$ SELECTs from table $r_1$ and INSERTs into table $r_2$. Transaction class $t_2$ SELECTs from table $r_2$ only. The transactions arrive for execution at the database server with rate $\lambda_1$ for $t_1$ and $\lambda_2$ for $t_2$. The database server is composed of a single CPU and two identical disks. The previous methodologies apply a systematic transformation of the database system design, i.e. the transaction and table specifications, to map the transaction class service demands on the CPU and disks. The service demands are calculated depending on the amount of CPU processing needed by each transaction to execute its SQL statements, and depending on which disk the physical files for tables $r_1$ and $r_2$ reside. These transformations will result in the queueing network model of Figure 2.10. For this queueing model, each transaction class will have a specified service demand on the CPU and disks, in addition to probabilities for visiting the CPU and disks.
As can be seen from Figure 2.10, the transaction is the smallest evaluated granularity which is represented as a customer class in a multi-class queueing network. The service demands of the transaction are divided between the disks and the CPUs in the hardware architecture. Response times will be evaluated at the level of the transaction; hence, the models are incapable of evaluating at a smaller granularity, e.g. which SQL statement caused the overall delay of the transaction. Moreover, the abstraction of the details of a transaction as service demands on the hardware devices does not lend to the straightforward representation and assessment of modern DBMS functionality, e.g. referential integrity and active database rules. In particular, that these functionalities affect the performance of other transactions in the database system.

In addition, performance evaluation is conducted at the final stage of the overall system design process, after all design decisions have been bound at the upper layers of the
database design process, even though enough information is available in the early
design stages to permit performance assessment. Hence, bottlenecks that are identified
on hardware devices are resolved through a reverse process, by backtracking to the early
software and database design artifacts to identify the cause of the performance problem,
and then redesigning and re-evaluating the performance model once more. This leads to
delays in feedback, complicates the performance evaluation methods and affects their
accurate application. Moreover, it questions the applicability of these methods in an
industrial setting.

2.8.3 An Approach for Parallel Relational Database System
Performance Evaluation

Tomov et al. [112] describe an analytical performance evaluation model for relational
parallel databases, targeted at DB administrators and not DB developers. In this method,
the authors categorize transactions based on their pattern of resource consumption on
the hardware architecture in a typical parallel DBMS. The methodology assumes the
availability of a partial query execution plan for each transaction accessing the DBMS.
This execution plan can be obtained from a DBMS query optimizer, and in that case, the
methodology takes advantage of the optimization strategy of the DBMS.

The method consists of three stages. The first stage is the preparation stage, in which
query execution plans are transformed into a set of low-level resource consumption
specifications that represent the execution of the query on the hardware architecture.
The output of this stage is a query resource usage profile for every node in the query
execution plan of a transaction, with regard to all hardware resources of the system, e.g.
number of visits, usage per visit and specification of parallel execution points. The resource usage profile contains control structures and lower-level operations that describe the sequential and parallel usage of hardware resources for each node in the query execution plan. Figure 2.11(a) shows an example of a simple resource usage profile for two transaction classes. The keyword \textit{loop} represents the repeated sequential usage of the set of resources, i.e. the expected number of visits.

\begin{align*}
\text{T}_1 & \quad \text{arrival rate} = \lambda_1 \\
\text{loop 5} & \\
\text{resCPU} & \quad 0.32 \text{ sec} \\
\text{resDisk1} & \quad 0.6 \text{ sec} \\
\text{resDisk2} & \quad 1.46 \text{ sec} \\
\text{T}_2 & \quad \text{arrival rate} = \lambda_2 \\
\text{loop 10} & \\
\text{resCPU} & \quad 0.5 \text{ sec} \\
\text{resDisk1} & \quad 1.0 \text{ sec} \\
\end{align*}

(a)

\textbf{Figure 2.11} An example of (a) a resource usage profile and (b) the corresponding queueing network for two transaction classes.

The second stage is the \textit{estimation of the mean resource response time}. In this stage the prediction of the response time for the hardware resources of the system, e.g. CPUs and disks, is based on all the query resource usage profiles identified in the previous step. This is conducted through the evaluation of an open, multi-class queueing network in
which the hardware resources represent FCFS servers. Each query resource usage profile represents a customer class in the network, with the service demands and transitions among the servers determined from the resource usage profile. From Figure 2.11(a), the transition probabilities for the customer classes are calculated from the structure of the resource usage profiles for each transaction class, resulting in the queueing network model of Figure 2.11(b).

The servers in the multi-class queueing network are considered a mixture of M/M/1 and M/G/1 queues. A heuristic rule was formulated in [110] in which the dominant resource in terms of utilization and relative visit ratio is designated as an M/G/1 queue. This combination of M/M/1 and M/G/1 queues with the application of the heuristic rule was shown to give similar results to that of more complicated approximation techniques for non-product form queueing networks [110, 111]. Solving the queueing network gives the waiting time for each hardware resource.

The final stage is the estimation of the mean query response time using the query resource usage profile and the estimated response times of individual hardware resources from the previous stage. This is accomplished by accumulating usage time as the query resource usage profile is traversed, while taking into account intra-operator parallelism.

A tool was developed to implement this methodology: STEADY (System Throughput Estimator for Advanced Database Systems) [120]. The method was validated against measurements from actual DBMS systems running on parallel machines using simple
queries [30, 112].

2.8.3.1 Discussion

The authors explicitly stated that the approach is not intended for database developers, but for database administrators. This is evident in that the modelling of the transactions start with the execution plan for the queries that make up the transaction. This, in regard to pre-implementation performance evaluation, implies at best a database system design at its final stages or at worst a completed system. Nonetheless, we shall discuss this approach based on its suitability for design time performance evaluation.

Unlike the previous methodologies, it allows for detailed representation of transaction processing, thus allowing the determination of SQL statements that cause bottlenecks or have unrealistic response times. Moreover, the transaction response times are calculated in steps, separating the underlying hardware architecture response time from the calculation of transaction response time. This allows for the evaluation of parallel transaction execution without complicating the underlying queueing network model of the hardware architecture.

However, the methodology suffers from the same drawbacks as the methodologies in the previous section. Mainly, performance evaluation is conducted very late in the development cycle and the methodology has been shown to apply to simple queries, as query execution plans do not depict active database rules or referential integrity checks [56, 73, 102].
2.9 Summary and Contribution

In this Chapter, we have contributed a categorization of the modelling of transactions in database and DBMS queueing network models. We have shown that the majority of queueing network models for databases and DBMS components fall into the transaction processing category, which implies the ability of the performance analyst to be able to determine the service demand of a transaction on the CPU and disk. This is not straightforward, nor is it easily measureable, while at the same time constituting a different domain for database designers. This reiterates the fact that the majority of models target capacity planning or overall system properties in generic systems. Work in detailed database transaction processing and behaviour is rarely studied.

In addition, we have contributed a justification for the exponential service time assumption for transactions in queueing network models, when transaction details are modelled.

With regard to the overview of the analytical performance evaluation methodologies developed for database system performance evaluation, we have identified the main shortcomings of these performance methodologies: the evaluation of overall database system performance is modelled by mapping the database system workload onto the hardware architecture of the system. Given that database systems are a major category of software systems, more detailed performance evaluation models are needed to correspond to the performance models available for different hardware architectures and software components.
The contribution of this work, in contrast to [1, 20, 43, 67, 89, 92, 112], is in (1) the representation of the transaction in the queueing model, and (2) the level at which the performance evaluation is conducted. In our method, we model the transactions as a set of phases – each phase corresponds to an access to a table as the transaction interacts with the database. The probability of accessing a table depends on the procedural structure of the transaction. Each table is modelled as a server in the queueing network. Transaction service times are assumed to be exponentially distributed, with a mean corresponding to the average number of I/O DB pages needed by the transaction on the table. Other properties of the queueing network model depend on the modelled database system characteristics. This work is similar to Tomov et al. [112], in that the sequential procedural structure of the transaction is used to decide the routing of the transaction in the queueing network. However, Tomov et al. assume service demands are on the hardware devices, while we assume service demands are on the tables.

The work in this thesis is an improvement over [1, 20, 43, 67, 89, 92] in that the transaction is modelled at a finer granularity, thus providing for feedback that is more relevant and useful to the database designer. Moreover, unlike [1, 20, 43, 67, 89, 92, 112], detailed knowledge and modelling of the hardware architecture is not required. Hence, database designs can easily be mapped onto the queueing network model. This simplifies the approach for database designers and allows the application of the method in early DB system design phases. Furthermore, this method provides for the explicit representation of active database rules and referential integrity in the queueing network models.
Chapter 3

A Queueing Networks Approach for the Performance Modelling of Database Designs

3.1 Introduction

Database system performance is measured in terms of query and transaction response time – the major indicator of a system capacity problem. After a database system has exhibited a performance problem, the main effort of post-deployment performance tuning is concentrated on the revision of the design of the database and the transactions running against the database [51, 84, 95, 123]. Hence, if the flaws of the database design had been discovered before system implementation and deployment, some of the post-deployment performance problems would have been avoided.

In addition to the general acceptance of the high impact of the performance evaluation of software systems in early development lifecycle phases, a performance evaluation method for database designs has the following benefits:
• prevents the propagation of design problems to the detailed design and implementation stages of a database system;

• simplifies work for database designers as well as application developers; performance evaluation feedback is relevant to the current state of the development process, thereby preventing costly backtracking to change requirements or application design;

• integrates performance evaluation in the database design process as well as the software development process;

• contributes in minimizing post-deployment database system performance tuning.

Performance modelling of database designs is possible because transaction execution costs can be estimated from the procedural structure of the transaction design, i.e. from the SQL statements, the procedural statements and the structure and relationships between tables, by using database query optimization and costing techniques [32, 51, 84].

At the design stage of database development, query optimization techniques are used as guidelines in designing efficient queries and transactions. These techniques can be adapted by a database designer to optimize a given SQL statement, at design time, in isolation, but are very cumbersome to use when considering the effect of a query on the performance of other transactions, or the effect of concurrent access to the database of different transactions or different invocations of the same transaction. Being so, the trend is to wait until database system deployment, when the effect of concurrency and
the interaction of different transactions will be clearer, to optimize the performance of problematic queries and updates [32, 84].

By using a performance model to evaluate the *dynamic* behaviour of the database design, the database designer can assess the expected performance of the design, *before* the physical deployment of the database system.

This Chapter describes the database design queueing network performance evaluation model. The steps in building the model are introduced and the transaction service demand calculation method is detailed. In addition, a formal specification of a database design, a queueing network and the transformation mechanism from a database design to the corresponding queueing network model is presented. The material in this Chapter has been outlined in [77].

### 3.2 The Database Design Queueing Network Model

Consider a database design composed of a set of tables and the transactions that access these tables. For each table the following is defined:

- the attribute data types and selectivity,
- the expected number of rows and row length,
- the index types and structure.

For each transaction the following is known:
• the rate of occurrence or its percentage of the total transactions,

• the SQL statements that make up the transaction, i.e. the tables that are accessed,
  the joined/retained attributes and their sequence and selectivity,

• the transaction structure, i.e. the procedural statements which enclose the SQL
  statements.

In our database design performance model, we represent the interaction between tables
and transactions as a queueing system. In the queueing system, the tables will represent
the shared resources, i.e. the servers, and the transactions that use these resources are the
customers. The total time for a client to process procedural statements can be
aggregated for each transaction as the client think time. Network latency can be
represented as a delay resource in the queueing network.

Disk I/O cost is the dominant factor in query execution costs [42, 84], especially for
large databases [38]; this is the cost criteria used to calculate the service demands for
transactions for our queueing network models. We currently ignore SQL processing
times, SQL aggregate function processing times and temporary table in-memory
operations. Referential integrity processing and active database rule invocations are
covered in Chapters 5 and 6, respectively. Other performance evaluation inputs, e.g. the
number of transaction invocations and user population are available, or can be
calculated from the database system design [98].

To model relational algebra JOIN operations between tables, we represent transaction
access to these tables as sequential access, based on the order of access defined on the
optimized query tree for the JOIN statement. Assuming DBMS query optimizers use left-deep query trees [84] to decide on an execution plan for a transaction, the order of table accesses for the JOIN operation will be the left-deep traversal of the JOIN operation’s optimized query tree.

In the following sections, the process of modelling a database design using queueing networks will be described.

3.2.1 Specifying Service Demands

The table and transaction specifications are used to calculate the number of DB I/O pages accessed by the transaction, by applying query optimization and costing techniques [32, 84]. A DBMS query cost optimizer builds a query tree to represent the SQL query based on the most efficient method to evaluate the query and, in turn, implement the relational operators. Efficiency is measured in DB I/O pages [32, 84]. Therefore, the optimized query tree provides the optimal access plan for the SQL query, in terms of the most efficient order to access the tables, as well as the number of I/O DB pages needed to retrieve the data.

A transaction may access more than one table or the same table more than once. Since DB I/O pages are the cost factor of the transactions, in our model we use the worst case scenario: all data pages are flushed from memory after a transaction completes its operations on the data; i.e. buffering is on a transaction-by-transaction basis only and skewed access to the data or large buffers are not accounted for. The consequence is that the service demand calculated for a transaction will use the number of unique DB pages accessed by the transaction.
To complete the calculation of the service demands on the tables, table physical structure (e.g. clustering and partitioning) and index types are used to calculate the final service demands, i.e. the total time to access the calculated number of I/O DB pages, using the formulae of the cost model described in [84] which is detailed in the next Section.

### 3.2.1.1 The Service Demand Cost Model

The cost model we use to estimate the expected execution time of DB I/O for a SQL statement is the cost model specified in [84]. The cost model is based on the underlying file organization of the DB table: heap file with no index, sorted file, clustered B+ tree file, clustered hash index file, heap file with an unclustered B+ index and heap file with an unclustered hash index.

The operations that can be executed by a SQL statement on a table are:

- **Sequential scans**: fetch all rows of a table, i.e. fetch all the DB pages for a table into the DB buffer.

- **Search with equality selection**: fetch all rows that satisfy an equality condition on an index key field. This encompasses fetching the DB pages of a table that contain the qualifying rows. However, we ignore the processing time to locate the correct row within the fetched page.

- **Search with range selection**: fetch all rows satisfying a range condition on the index key fields, once more ignoring processing times to locate the rows within fetched pages.
• **Insert a new row**: locate the DB page in which the row will be inserted, fetch the DB page from disk, modify it to include the new row, then write it back to disk.

• **Update/delete an existing row**: identify the DB page that contains the specific row, fetch it from disk, delete/update the row, then write the DB page back to disk.

For the cost model we use the following notations:

• $B$: denotes the total number of DB pages in a table, neglecting all header information, i.e. DB pages are assumed to be fully loaded with no space considerations for page or row headers.

• $D$: the average time to read or write a DB page to/from disk, i.e. average DB page I/O time;

• $F$: the tree index fan-out, i.e. the average number of children for a non-leaf node;

• $R$: ratio of the index entry size to the table row size.

The cost model is summarized in Table 3.1. In the next Section, the formulae are derived for a heap file with an unclustered B+ tree index, which is the file organization used in the experiments in the following Chapters. Details for the other file organizations can be found in [84].
Table 3.1 I/O DB page cost model for SQL operations.

<table>
<thead>
<tr>
<th>Table Type</th>
<th>Scan</th>
<th>Equality Search</th>
<th>Range Search</th>
<th>Insert</th>
<th>Update/Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>BD</td>
<td>0.5BD</td>
<td>BD</td>
<td>2D</td>
<td>Search + D</td>
</tr>
<tr>
<td>Sorted</td>
<td>BD</td>
<td>Dlog₂B</td>
<td>D( log₂B + # of matching pages)</td>
<td>Search + BD</td>
<td>Search + BD</td>
</tr>
<tr>
<td>Clustered tree index</td>
<td>BD</td>
<td>Dlog₂B</td>
<td>D( log₂B + # of matching pages)</td>
<td>Search + D</td>
<td>Search + D</td>
</tr>
<tr>
<td>Clustered hash index</td>
<td>BD</td>
<td>1.2D</td>
<td>1.2D( # of hash keys in range)</td>
<td>Search + D</td>
<td>Search + D</td>
</tr>
<tr>
<td>Unclustered tree index</td>
<td>BD( # of records per page + R)</td>
<td>D(1+ log₂RB)</td>
<td>D( log₂RB + # of matching records)</td>
<td>Search + 2D</td>
<td></td>
</tr>
<tr>
<td>Unclustered hash index</td>
<td>BD( # of records per page + R)</td>
<td>2D</td>
<td>BD</td>
<td>4D</td>
<td>Search + 2D</td>
</tr>
</tbody>
</table>

*B* denotes the number of DB pages in a table neglecting header information, i.e. pages are fully loaded, *D*: the average time to read or write a DB page, *F*: the tree index fan-out, *R*: ratio of the index entry size to the table row size
Calculating I/O Cost for a Heap File with an Unclustered Tree Index [84]

**Scan:** To perform a full table scan, scan the leaf level of the index and fetch the corresponding row for each index entry. The cost of reading all index entries is $RBD$. Then we have to fetch all the corresponding rows. Given that this is an unclustered tree index, each leaf entry can point to a different DB page. Therefore, the cost of fetching all the rows is one I/O disk access per row, i.e. the number of rows per page $\times BD$. Thus, the total cost of a full table scan is $BD(\text{the number of records per page} + R)$.

**Search with equality on the index key:** Locate the first page containing the desired index entries by fetching all index pages from the root to the appropriate leaf; this takes $\log_{RB}$ steps. Each step costs a disk I/O, thus the cost is: $D\log_{RB}$. The qualifying data row will cost an additional disk I/O. Hence the total cost is: $D(1+ \log_{RB})$.

**Search with range selection:** The first qualifying row in the range costs the same as a search for a row with equality on the index key: $D(1+ \log_{RB})$. Then index data entries are retrieved sequentially until an entry that does not satisfy the range selection is found. Each retrieved index entry incurs one I/O to fetch the corresponding row. This will cost: (the total number of matching records $- 1$). Therefore, the total cost is: $D(\log_{RB} + \text{number of matching records})$.

**Insert:** When inserting a row, it is first inserted into the heap file at a cost of one disk access to read the DB page and another to write the modified page to disk, i.e. $2D$. In addition, the corresponding data entry must be inserted into the index. The cost of finding the correct leaf page is $D\log_{RB}$ and writing the modified index page to disk costs another I/O disk access. Hence, the total cost is: $D(3 + \log_{RB})$. 

58
**Update/Delete:** The cost of locating the row in the table and locating the index entry is: \( D \log_{10} RB + D \). To write the modified table and index pages to disk costs: \( 2D \). Therefore the total cost is: \( D(3 + \log_{10} RB) \): which is the cost of the equality search plus \( 2D \).

### 3.2.2 Building the Queueing Network Model

The steps to build the queueing network model are:

**Step 1: Specify queueing network model structure:**

(a) **Servers:** each table in the database design is a server in the queueing network; partitioned or replicated tables are represented as separate servers.

(b) **Customer classes:** each transaction type is considered as a different customer class: transaction types that access identical tables with equal service demands may be considered as one class.

(c) **Scheduling discipline is FCFS:** DBMS use queues to control access to data objects; a new transaction is given access to a data object depending on the state of the current transactions waiting to access or currently accessing the data object. Depending on the concurrency control mechanism implemented by the DBMS, access is either granted immediately to the new transaction, or it is forced to wait behind the current transactions [84]. The effect of FCFS is in forcing all transactions to wait.

Given that the queueing network model represents the whole database, a transaction still inside the queueing network is analogous to a transaction still
accessing the database, i.e. has not committed or aborted. In this scenario, when
transaction A finishes service at table X and enters table Y, any transaction
entering table X, is in fact accessing table X in parallel with transaction A.
Therefore, FCFS gives serial access at the transaction statement level (i.e. at the
lowest granularity of access: the row level in this case), but the model gives
parallel access at the transaction level.

(d) **Queue length is infinite:** this is based on the assumption that aborts due to
deadlocks are rare in DBMS [84] and system overload causes long response
times instead of transaction aborts.

**Step 2: Specify performance characteristics for the customer classes:**

(a) **Transaction service demands on each server:** the total cost of executing the
SQL statements in terms of I/O DB pages. Service times are assumed to be
exponentially distributed, with the mean being the service demands calculated
in the previous section. A justification for exponential service times was
provided in Chapter 2.

(b) **Transaction rate:** for open queueing networks: arrival rates, and for closed
queueing networks: transaction think times and number in system.

**Step 3: Specify the routing table for the customer classes:** i.e. the order in which the
transactions access their tables. This is derived from the procedural structure of the
transaction. In addition, for tables in JOIN statements the order of the left-deep traversal
of the optimized query tree is used.
Step 4: Solve the queueing network model: depending on the complexity of the queueing network model and the solution method used, the queueing network model can specify the bottleneck tables, total access compared to other tables, and transaction response times and response time distributions [74-76].

Table 3.2 summarizes the mapping between database design entities and queueing network models.

Table 3.2  Mapping between database designs and queueing network models.

<table>
<thead>
<tr>
<th>Database Design</th>
<th>Queueing Network Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>server</td>
</tr>
<tr>
<td>transaction type</td>
<td>customer class</td>
</tr>
<tr>
<td>transaction rate of occurrence or percentage of total transactions</td>
<td>arrival rate or number in system</td>
</tr>
<tr>
<td>cost of I/O DB pages needed to execute the SQL statements of the transaction on a table</td>
<td>customer class service demand on a server</td>
</tr>
<tr>
<td>order of SQL statements in the transaction</td>
<td>traversal path of the customer class</td>
</tr>
</tbody>
</table>

3.2.3 An Example

Consider a simple database design that consists of two heap organized tables: EMP(emp_no, emp_name, dept_no) and DEPT (dept_no, dept_name) and two transactions, New_Emp and List_Emp (Figure 3.1). Each transaction is composed of a number of SQL statements and procedural statements. A queueing network model of this database system design can be built, using the steps detailed in the previous section and the cost model of Table 3.1.
Transaction **New_Emp**
Input Parameters:
   (:emp_no_var, :emp_name_var, :dept_name_var)
Body
   SELECT dept_no INTO :dept_no_var
   FROM DEPT
   WHERE dept_name=:dept_name_var;
   INSERT INTO EMP
   VALUES
   (:emp_no_var, :emp_name_var,:dept_no_var);
   show_message('Operation Successful');
End Transaction;

Transaction **List_Emp**
Input Parameters:
   (:dept_no_var)
Body
   SELECT emp_no,emp_name
   FROM EMP
   WHERE dept_no=:dept_no_var;
   Print_List_Procedure;
End Transaction;

**Figure 3.1** Details of New_Emp and List_Emp transactions.

Each table in the database design is a server in the queueing network – from the example, EMP and DEPT are the servers in the queueing network model. Each transaction represents a customer class in the queueing network. The order in which the tables are accessed by each transaction is: for the New_Emp transaction, DEPT then EMP. The List_Emp accesses the EMP table only. This gives the queueing network model of Figure 3.2.

**Figure 3.2** A queueing network model for the database design example.

The service demand on each server (table) is equal to the total time to access the data, which is the cost of the DB pages needed to be retrieved from disk. Neither table has an index; therefore, we will use the heap table cost model. Assume the size of the EMP and
DEPT tables in DB pages are twelve and two pages, respectively, and that the duration of one disk access is arbitrarily chosen as one second. Table 3.3 shows the service demands for each transaction on each table. Arrival rates (open network) or number in system (closed network) can be derived from the database system design specifications and the model can be solved.

Using the information from the results of the performance evaluation, the database designer can decide whether the expected response time of the transactions is suitable. If not, the other results, like the residence time, can be traced back to specific transactions or tables. Changing the design properties will change the number of DB I/O pages accessed by each transaction, thereby changing the result of the model: hence, it is possible to experiment with different indexes, table sizes, table structures, etc.

**Table 3.3** Service demands for the New_Emp and List_Emp transactions.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Service Demand (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>EMP</strong></td>
</tr>
<tr>
<td>New_Emp</td>
<td>12D=12</td>
</tr>
<tr>
<td>List_Emp</td>
<td>0.5BD=6</td>
</tr>
</tbody>
</table>

### 3.3 The Formal Specification

In this section, we present a formal specification of the process of modelling a database design using queueing networks. To formally describe the queueing network database design performance evaluation technique we will use the notation described in Table 3.4.
Table 3.4  Formal specification notation.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x l</td>
<td>cardinality</td>
</tr>
<tr>
<td>l</td>
<td>choice</td>
</tr>
<tr>
<td>C</td>
<td>class type</td>
</tr>
<tr>
<td>c_i</td>
<td>class instance</td>
</tr>
<tr>
<td>.</td>
<td>class member access</td>
</tr>
<tr>
<td>&lt;…&gt;</td>
<td>comments</td>
</tr>
<tr>
<td>( ... )</td>
<td>grouping</td>
</tr>
<tr>
<td>A[i]</td>
<td>vector</td>
</tr>
<tr>
<td>A[i,j]</td>
<td>matrix</td>
</tr>
<tr>
<td>[ ... ]*</td>
<td>repetition: 0 or more</td>
</tr>
<tr>
<td>[ ... ]+</td>
<td>repetition: 1 or more</td>
</tr>
<tr>
<td>[ ... ]^n</td>
<td>repetition: n times</td>
</tr>
</tbody>
</table>

3.3.1 Database Design Formal Specification

A database design can be formally described as \( DBDesign = (R, T) \), where \( R \) is the set of relations or tables and \( T \) is the set of transactions that access these tables. Define each table \( r_i \) in \( R \) as:

\[
r_i = (\text{<ordered set of>} \ A, I, [\text{uniqueness constraint}]^*, \text{expected number of rows}, \text{average row length})
\]

where

- \( A \) is the set of attributes of \( r_i \). For each attribute \( a_j \) in \( A \), the data type, range and selectivity or probability distribution is defined.
• \( I \) is the set of indexes of \( r_i \). For each index \( i_n \) in \( I \), its type and structure is defined.

If a table is denormalized into \( n \) partitions, it is represented as \( n \) different tables. If \( n \) tables are clustered, they are represented as one table.

Define each transaction \( t_j \) in \( T \) as:

\[
t_j = (\text{<ordered set of> } S, \text{Rate}, \text{tranDBpages}[r_i \in \mathcal{R}])
\]

where \( S \) is the ordered set of statements of the transaction, and \( s_k \) in \( S \) is defined as:

\[
s_k = (q, \text{loop} | \text{branch}, \text{statDBpages}[r_i \in \mathcal{R}])
\]

such that:

• \( q \) is a SQL statement and can be described as:

\[
q = (\text{type}, \text{<ordered set of> Access, DBpages}[r_i \in \mathcal{R}]),
\]

and

- \( \text{type} \) is either a SELECT, UPDATE, INSERT or DELETE SQL statement including the conditional clause parameters;
Access is the ordered set of tables \((r_i \in \mathcal{R})\) that are accessed by \(q\), including the joined/retained attributes \((a_j \in \mathcal{A}, r_i)\) and their sequence and selectivity;

- DBpages\([r_i \in \mathcal{R}]\) can be described as: Let \(B_i\) be the number of DB I/O pages calculated for the given SQL statement \(q\) on each table \(r_i\) it accesses; then the cost of accessing these DB pages is:

\[
\text{DBpages}[r_i \in \mathcal{R}] = B_i \times D,
\]

where \(D\) is the mean time to read or write a DB page. The value \(B_i \times D\) is also known as the service demand. Details of the calculation method for \(B_i \times D\) were stated in Section 3.2.1.

- **loop** is a loop statement and is described as:

\[
\text{loop} = (\text{<ordered sequence of>} q^+, \text{total-iterations}).
\]

- **branch** is a conditional branch statement, which is described as:

\[
\text{branch} = [ (\text{<ordered sequence of>} q^+, p_i) ]^{\text{total number of branches}},
\]

where \(p_i = \text{probability of accessing branch}_i\), \(\sum_{i=1}^{\text{branch}} p_i = 1\).

The service demand or total cost of accessing the DB I/O pages calculated for the statement \(s_k\), for all accessed tables, is \(\text{statDBpages}[r_i \in \mathcal{R}]\). This calculation depends on the type of \(s_k\).
• If \( s_k = q \), then \( \text{statDBpages}[r_i \in \mathcal{R}] \) can be described as: \( \forall r_i \in \mathcal{R}, \text{statDBpages}[r_i \in \mathcal{R}] = q \cdot \text{DBpages}[r_i] \).

• If \( s_k = \text{loop} \), then \( \text{statDBpages}[r_i \in \mathcal{R}] \) is:

\[
\forall r_i \in \mathcal{R}, \text{statDBpages}[r_i \in \mathcal{R}] = n \sum_{m=1}^{\text{loop,q}_n} q_m \cdot \text{DBpages}[r_i],
\]

where \( n \) is the number of loop iterations and \( q_m \) is the \( m^{\text{th}} \) SQL statement in the loop.

• If \( s_k = \text{branch} \), then \( \text{statDBpages}[r_i \in \mathcal{R}] \) can be described as:

\[
\forall r_i \in \mathcal{R}, \text{statDBpages}[r_i \in \mathcal{R}] = \sum_{j=1}^{\text{branch}[j]} \sum_{m=1}^{\text{branch}[j],\#} q_m \cdot \text{DBpages}[r_i],
\]

where \( n \) is the total number of branches and \( q_m \) is the \( m^{\text{th}} \) SQL statement in the branch. For simplicity, we assume that each SQL statement in a branch accesses a different table.

For the transaction \( t_j \), Rate can be defined as:

\[
\text{Rate} = \text{arrival rate} \times (\text{think time}, [\% \text{ of total transactions}] \times \text{average number in system}),
\]

and \( \text{tranDBpages}[r_i \in \mathcal{R}] \), which is the service demand of \( t_j \) on each \( r_i \in \mathcal{R} \), can be defined as:
\[ \forall r_i \in R, \text{tranDBpages}[r_i \in R] = \sum_{k=1}^{\text{numstat}s} s_k \cdot \text{statDBpages}[r_i], \]

where \( s_k \) is the \( k^{th} \) statement in the transaction.

For simplicity, in this specification, we are assuming that if a transaction accesses a table in multiple SQL statements, these statements are in sequence. This is to accommodate for a simple routing path algorithm (see Section 3.3.3) in which a transaction class visits a server (table) only once. A general routing algorithm, in which multiple visits to the same table are allowed can be specified by changing the customer type for each distinct visit to a table, as it would be expected that the service demand, i.e. the number of DB pages, would be different for each visit.

### 3.3.2 Queueing Network Model Formal Specification

The queueing network model can be formally described as:

\[
\text{QN} = (\text{Server}, C, \lambda[C], D[\text{Server}, C], \mathcal{P}[C, \text{Server}, \text{Server}])
\]

where:

- \( \text{Server} \) is the set of resources of the queueing network. Each \( server_i \in \text{Server} \) is a FCFS service center with exponential service time and infinite queue capacity.

- \( C \) is the set of customer classes seeking service in the queueing network.
• $\lambda[C]$ is defined as: $\forall c_i \in C$:

$$\lambda[i] = (<\text{open queueing network}> \textit{arrival rate} | <\text{closed queueing network}> ( \textit{think time}, [\% \text{ of total transactions} \mid \text{average number in system} )$$

• $\mathcal{D}[Server, C]$ is a $|Server| \times |C|$ matrix of the mean service demands of the customers on the queueing network. $\mathcal{D}[Server, C]$ is defined as $\forall server_i \in Server$, $\forall c_j \in C$, $\mathcal{D}[i, j]$ is the service demand of $c_j$ on $server_i$.

• $\mathcal{P}[C, Server, Server]$ is a $|C| \times |Server| \times |Server|$ matrix of the path a customer class traverses through the queueing network. $\mathcal{P}[C, Server, Server]$ is defined as: $\forall c_i \in C$, $\forall server_i \in Server$, $\forall server_k \in Server$, $\mathcal{P}[i, j, k]$ is the probability of $c_i$ moving to $server_k$ when leaving $server_j$. In addition, $\forall c_i \in C$, for each $server_j \in Server$, $\sum_{k=1, k \neq j}^{\lfloor Server \rfloor} \mathcal{P}[c_i, j, k] = 1$.

### 3.3.3 Building the Queueing Network Model from the Database Design

To transform the database design $DBDesign = (R, T)$ into the queueing network $QN = (Server, C, \lambda[C], \mathcal{D}[Server, C], \mathcal{P}[C, Server, Server])$ apply the following:

• $server_i = r_i$, $\forall server_i \in Server$, $\forall r_i \in R$, where $|Server| = |R|$, partitioned or replicated tables are represented as separate servers.
• \( c_i = t_i, \forall c_i \in C, \forall t_i \in \mathcal{T} \), where \(|C| = |\mathcal{T}|\), transaction types that access identical tables with equal service demands may be considered as one class.

• \( \lambda[i] = t_i.\text{Rate}, \forall \lambda[i] \in \lambda[C], \forall t_i \in \mathcal{T}. \)

• \( D[i, j] = t_j.\text{tranDBpages}[r_i]. \)

• \( P[i, j, k] \) is calculated using Algorithm 3.1. The algorithm takes as input the formal description of each transaction \( t_j \) in \( \mathcal{T} \) and outputs the routing probabilities for the corresponding customer class \( c_i. \)

---

**Algorithm 3.1: Calculating Customer Class Path**

1: \( \forall t_i \in \mathcal{T} \)

2: Let \( current\_table \) be the current table in the path of \( t_i \)

3: \( current\_table \leftarrow 0 \)

4: Let \( \text{branch}[n] \) be a vector of \( r_k \in \mathcal{R} \), that holds the last table accessed by a branch[i] of a branch statement, where \( n \) is the number of branches

5: \( \text{branch}[\_] \leftarrow \text{nil} \) (element by element assignment)

6: Let \( \text{bran}\_table \) be the current table of a branch statement

7: \( bran\_table \leftarrow 0 \)

8: Let \( \text{prev}\_branch[\_] \) be a vector that holds the initial value of \( \text{branch}[\_] \)

9: \( \text{prev}\_branch \leftarrow \text{nil} \)

10: \( \forall s_j \in t_i. S \)

11: \( \text{case } s_j = q \)

12: \( \forall r_k \in q.\text{Access} \)

13: \( \text{if } (r_k \text{ is first table accessed by } q) \text{ and } \text{branch}[\_] \neq \text{nil} \text{ then} \)

\( \) (connect the last tables accessed by the previous branch statement to the first table of this SQL statement)

14: \( \text{for } \text{branch}[1] \text{ to } |\text{branch}[\_]| \text{ do} \)

15: \( P[c_i, \text{branch}[1], r_1] \leftarrow 1 \)

16: \( \text{end for} \)

17: \( \text{branch}[\_] \leftarrow \text{nil} \)

18: \( current\_table \leftarrow r_k \)
else
    $P[c, c_i, current_table, r_k]^{1} \leftarrow 1$
    $current_table \leftarrow r_k$
end if

case $s = loop$

for $q_m \in loop$
    for $r_k \in q_m$.Access$
        if ($q_m$ is first SQL statement) and ($r_k$ is first table accessed by $q_m$) and $branch[] \neq nil$ then
            (connect the last tables accessed by the previous branch statement to the first table of this SQL statement)
            $P[c, c_i, branch[], r_k] \leftarrow 1$
end for
        $branch[] \leftarrow nil$
        $current_table \leftarrow r_k$
    else
        $P[c, c_i, current_table, r_k] \leftarrow 1$
        $current_table \leftarrow r_k$
end if

case $s = branch$

for $i$ in $n$ (total number of branches)
    $bran_table \leftarrow current_table$
    for $q_m \in branch_i$
        for $r_k \in q_m$.Access$
            if ($q_m$ is first SQL statement) and ($r_k$ is first table accessed by $q_m$) then
                Let $p_i$ be the probability of accessing $branch_i$
                if $prev_branch[] \neq nil$ then
                    (connect the last tables accessed by the previous branch statement to the first table of this branch’s SQL statement)
                    $P[c, c_i, prev_branch[], r_k] \leftarrow p_i$
                end if
            else
                $P[c, c_i, bran_table, r_k] \leftarrow p_i$
                $bran_table \leftarrow r_k$
            end if
        else
            $P[c, c_i, bran_table, r_k] \leftarrow 1$
            $bran_table \leftarrow r_k$
        end if
    end for
end for
$prev_branch \leftarrow nil$
end case

$P[c, 0, r_k]$ gives the entry server for $t$. Unassigned values take the value zero.
3.4 Summary

In this Chapter, the database design queueing network performance evaluation model was introduced. The cost model for calculating service demands for the transactions was presented. A formal specification of the model and the transformation between database designs and corresponding queueing network models was described. The formal specification and its related algorithms can form the basis upon which to develop a tool for implementing this performance evaluation technique.
Chapter 4

Modelling the TPC-C Benchmark

4.1 Introduction

In order to validate and evaluate our database performance evaluation technique, we compare the results of a queueing network model to the performance of an actual database system. We have chosen the Transaction Processing Performance Council (TPC) benchmark [113] as the database design specification. In this Chapter, we model the TPC-C benchmark and compare the results of the queueing network database performance model with the TPCC-UVA open source implementation of the TPC-C benchmark developed at the University of Valladolid, Spain [59]. The purpose of the database performance evaluation model is to provide the database designer with the ability to compare between different database designs at database system design time. In Section 4.6, we conduct a comparison between three different designs for the TPCC-UVA system using the queueing network performance evaluation model. The results in this Chapter have been published in [76, 77].

4.2 The TPC-C Benchmark

The Transaction Processing Performance Council (TPC) [114] TPC-C benchmark [113] is an on-line transaction processing (OLTP) benchmark. It is written to be as representative as possible to actual production applications and environments. However,
the TPC-C benchmark has some shortcomings in its ability to represent actual OLTP database applications and workloads: the benchmark’s accommodation of known optimal memory buffering techniques cannot be replicated on real workloads [57], its workload is considerably different from actual production workloads [41] and its I/O reference behaviour does not replicate that of actual production systems [13, 42].

In spite of the aforementioned shortcomings, the TPC-C benchmark is still the de facto standard benchmark for OLTP systems in industry, as well as being the only database system benchmark with published results for different software and hardware configurations. Moreover, the purpose of this work is to establish the ability to model database designs using queueing networks; thereby, for our purposes, and in this context, we believe that the TPC-C benchmark fulfils our needs and its shortcomings can be viewed as particular properties or specifications of the database system under evaluation.

The TPC-C benchmark revision 5.8.0 [113, 115] is used as an example of a database system design. The TPC-C benchmark is a design specification of an order-entry system. The benchmark portrays a wholesale supply company with a set of sales districts and associated warehouses. Each warehouse covers 10 districts, each district serves 3,000 customers. Warehouses hold stock of 100,000 items. Customers can place new orders or enquire about the status of existing orders. Orders have an average of 10 items (order lines). For all order lines 1% are from items not in stock at the district’s warehouse and must be supplied by another warehouse. The order-entry system provides for the entering of customer payments, the processing of orders for delivery and the identification of shortages in stock levels.
The TPC-C specification is composed of [113]:

- 9 tables (WAREHOUSE, DISTRICT, CUSTOMER, HISTORY, ORDER, NEW-ORDER, ORDER-LINE, STOCK, ITEM) and
- 5 transactions (New-Order, Payment, Order-Status, Delivery, Stock-Level).

A brief description of the transactions is in Table 4.1. The details of the design of the tables, the relationships between them, the restrictions on the random data generation for populating the database and the details of transaction functionality can be found in [113]. Appendix A summarizes the transaction descriptions and Appendix B shows table structure and data population specifications.

**Table 4.1 Summary of the TPC-C benchmark transactions.**

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Description</th>
<th>Min. % of the total number of transactions</th>
<th>Min Mean of Think Time Distribution (seconds)</th>
<th>Min Keying Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>New-Order</td>
<td>Initiates a new order</td>
<td>No minimum</td>
<td>12</td>
<td>18</td>
</tr>
<tr>
<td>Payment</td>
<td>Updates the customer’s balance and reflects the payment on the district and warehouse sales statistics</td>
<td>43</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>Order-Status</td>
<td>Queries the status of a customer’s last order</td>
<td>4</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>Delivery</td>
<td>Processes a batch of 10 new orders, one for each district for a given warehouse</td>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Stock-Level</td>
<td>Counts the number of items in the last 20 orders in a district that fall below the stock threshold</td>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

The mean keying and think times for the different transactions are specified in Table 4.1. In view of the fact that the TPC-C benchmark is emulating a real user environment, it states that after transaction \( i \) finishes executing and returns the result to the user, the
user processes that data (think time of transaction $i$) before choosing a new transaction and keying in its parameters (keying time for transaction $i+1$).

The TPC-C benchmark also includes performance specifications related to the implementation of the database system, such as [113]:

- regulation of the transaction mix during the measurement period (Table 4.1);

- database population and scaling requirements: Table 4.2 shows the scaling requirements based on the number of warehouses in the database;

- randomness and probabilities of values for the initial database loading;

- the probability of operations on the database and the probability of choosing the values of the parameters for the transactions;

- the required performance results.

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Cardinality (in rows)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAREHOUSE</td>
<td>1</td>
</tr>
<tr>
<td>DISTRICT</td>
<td>10</td>
</tr>
<tr>
<td>CUSTOMER</td>
<td>30,000</td>
</tr>
<tr>
<td>HISTORY</td>
<td>30,000</td>
</tr>
<tr>
<td>ORDER</td>
<td>30,000</td>
</tr>
<tr>
<td>NEW-ORDER</td>
<td>9,000</td>
</tr>
<tr>
<td>ORDER-LINE</td>
<td>300,000</td>
</tr>
<tr>
<td>STOCK</td>
<td>100,000</td>
</tr>
<tr>
<td>ITEM</td>
<td>100,000</td>
</tr>
</tbody>
</table>

Table 4.2 Scaling requirements for the TPC-C database.
However, we incorporated the following assumptions to the TPC-C benchmark specifications when modelling our queueing network performance models:

- The TPC-C benchmark specification states that:

  - 1% of all New-Order transactions rollback, we assume that no transaction rolls back;

  - the Payment and Order-Status transactions are invoked 60% of the time using the customer’s last name and 40% of the time using customer_id. We calculated the I/O costs based on the average number of pages needed for access by customer last name and customer_id.

- We use the average value for all parameters, e.g. the number of items in an order is randomly selected between 5 and 15, we assume 10 items to an order.

4.3 The TPCC-UVA Implementation

The TPCC-UVA [59] is an open source implementation of the TPC-C benchmark for the PostgreSQL database. TPCC-UVA is written in C language for Linux systems. It is composed of a set of remote terminal emulators that simulate the behaviour of users based on the TPC-C benchmark specifications. Figure 4.1 details the TPCC-UVA architecture.
The TPCC-UVA implementation is composed of modules that implement the TPC-C benchmark system. They provide for all of the processing needed to measure the performance of the system. These modules are [59]:

- **The benchmark controller**: this is the user interface of the TPCC-UVA system, it allows for (1) the initial population of the database based on the selected number of warehouses, (2) the launch of different experiments on the populated database for different combinations of warehouses and districts by specifying the ramp-up and measurement intervals and (3) provides the results summary in report and graphical formats.

- **The remote terminal emulator**: each district specified in an experiment represents a remote system terminal according to the TPC-C specifications. There is one remote terminal emulator process per active terminal in the benchmark execution. The remote terminal emulator simulates the activity of a remote terminal (transaction generation, waiting times, keying times, etc) as
specified in the TPC-C benchmark. In addition, each remote terminal emulator logs the response times for all transactions executed by the terminal.

- **The transaction monitor**: all database requests from the remote terminal emulators are sent to the transaction monitor, which in turn executes the queries on the underlying database system.

The communication between the transaction monitor and the remote terminal emulators is by a shared memory queue of pending transaction requests. The execution order of transaction requests is FCFS. Semaphores are used to synchronize the read and writes of the remote terminal emulator and transaction monitor to the queue. When a transaction completes execution on the database the results are transmitted from the transaction monitor to the issuing remote terminal emulator through a semaphore synchronized shared-memory data structure.

Appendix B details the TPCC-UVA database table structures with the TPC-C data population specifications. Appendix C illustrates the TPCC-UVA transactions’ SQL source code.

For our performance evaluation experiments we have used the TPCC-UVA system as provided. However, we incorporated the following modifications to the TPCC-UVA implementation:

- we modified the implementation of the nonuniform random function used for data generation [113] in the TPCC-UVA to use the parameter value C=1, to simplify transaction service demand calculations (see Appendix B);
• foreign key references in all tables were removed; this prevents the processing overhead of foreign keys which is currently not represented in the model;

• the initial database check which read the whole database into the database buffer was removed. This allowed the actual transactions to fill the buffer as needed, hence the simulated model and the implementation begin from the same initial state: an empty buffer;

• the implementation of the New-Order transaction was edited to place SQL statements accessing the same table in sequence (this affected only one SQL query accessing one table); this did not change the functionality of the transaction, but simplified the design of the queueing network model.

The TPCC-UVA experimental platform was a Pentium 4 Dual Core Processor at 2.4 GHz with 2GB RAM and 150 GB HD running Linux. All software has the default configuration and the TPCC-UVA and PostgreSQL database version 8.3.3 [102] were installed as stated in [58], with the modifications stated above.

4.4 Building the Performance Evaluation Model

To build the queueing network model for the TPCC-UVA database design, the design specification of the TPC-C benchmark was used to specify the probability of operations on the database and the distribution of the parameter values for the transactions.
4.4.1 Measuring DB Page Access Time

In order to collect information on the time it takes the kernel to fulfil a DB page request we employ the Linux strace utility to trace read and write system calls to database files between the PostgreSQL database engine and the Linux kernel. The strace utility provides the time duration to fulfil these system calls. The arithmetic mean is taken of the times to fulfil all the read and write system calls to database files during the experiment measurement interval. This gives the mean DB page access time, which accounts for actual DB page requests; any pages already in the DB buffer before the beginning of the measurement interval will not be accounted for.

Given that the mean DB page access time is calculated during the measurement interval only, it will give the mean kernel response time when the TPCC-UVA system is in the steady state.

4.4.2 Calculating Transaction Service Demands

The database initial loading size is based on the database population specification of the TPC-C benchmark [113]. We have used data for 100 warehouses, each with 10 districts, i.e. 100x10 clients (Table 4.3). This is the initial configuration for all our experiments irrespective of the actual number of clients used in an experiment and is used to calculate service demands for the transactions for the queueing network model. TPCC-UVA actual data will vary slightly due to random generation.
Table 4.3 Initial loading size for the TPCC-UVA queueing network model.

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Cardinality (in rows)</th>
<th>Rows Per Page* (in rows)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAREHOUSE</td>
<td>100</td>
<td>93</td>
</tr>
<tr>
<td>DISTRICT</td>
<td>1,000</td>
<td>87</td>
</tr>
<tr>
<td>CUSTOMER</td>
<td>3,000,000</td>
<td>13</td>
</tr>
<tr>
<td>HISTORY</td>
<td>3,000,000</td>
<td>179</td>
</tr>
<tr>
<td>ORDER</td>
<td>3,000,000</td>
<td>342</td>
</tr>
<tr>
<td>NEW-ORDER</td>
<td>900,000</td>
<td>1,024</td>
</tr>
<tr>
<td>ORDER-LINE</td>
<td>30,000,000</td>
<td>152</td>
</tr>
<tr>
<td>STOCK</td>
<td>10,000,000</td>
<td>27</td>
</tr>
<tr>
<td>ITEM</td>
<td>100,000</td>
<td>100</td>
</tr>
</tbody>
</table>

*PostgreSQL DB page size is 8 Kbytes. DB pages are fully loaded.

Using query optimization techniques and the cost model in Table 3.1, the number of DB pages needed by each TPCC-UVA transaction is calculated from the tables, index structures and SQL statements described in the source code. In addition, from the TPCC-UVA implementation, the process in which the data was initially generated and loaded into the database was taken into account, e.g. some tables were loaded in key sort order. This gives the values in Table 4.4. Appendix C details the transaction SQL statements and the corresponding formulas used to derive the values in Table 4.4.

The values in Table 4.4 will be used for all our experiments regardless of the number of clients or the length of the execution run. This is due to the fact that the TPCC-UVA transaction access to data does not depend on the table size.

Table 4.4 Number of I/O DB pages for the TPCC-UVA transactions.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>number of I/O DB pages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I</td>
</tr>
<tr>
<td>New-Order</td>
<td>0.75</td>
</tr>
<tr>
<td>Payment</td>
<td>2.75</td>
</tr>
<tr>
<td>Order-Status</td>
<td>-</td>
</tr>
<tr>
<td>Delivery</td>
<td>-</td>
</tr>
<tr>
<td>Stock-Level</td>
<td>-</td>
</tr>
</tbody>
</table>

I = WAREHOUSE, II= DISTRICT, III= CUSTOMER, IV= HISTORY, V= ORDER, VI= NEW-ORDER, VII= ORDER-LINE, VIII= STOCK, IX= ITEM
One exception is the Order-Status transaction, in which the number of DB pages accessed on the ORDER table depends on the number of New-Order transactions executed; we have incorporated this in the queueing network model simulation (details in Appendix D). The value shown in Table 4.4 for the Order-Status transaction on the ORDER table is the initial value.

The service demand of a transaction on the relevant table is the calculated number of I/O DB pages needed by the transaction on that table × the mean time to access a DB page. Therefore, for the TPCC-UVA transactions, their service demands will be the values in Table 4.4 multiplied by the mean DB page access time calculated in the previous Section.

### 4.4.3 Building the Queueing Network Model

Applying the steps described in Section 3.2, the queueing network model for the TPCC-UVA database system has 9 servers (tables) and 5 customer classes (transactions) with service demands on each server, as calculated in the previous Sections.

From the TPCC-UVA transaction structure, the order in which each transaction accesses its tables is used to define how it will traverse the queueing network. In addition, the TPCC-UVA architecture has one transaction monitor that receives all requests from the remote terminal emulators, which are queued for service by order of arrival [59]. As a consequence, there is only one transaction being processed in the DB at a time. The transaction monitor is represented as a queue without a service center in the queueing network model. A customer leaves the transaction monitor queue and begins service in the database only after the last customer finishes service. Hence the database acts as the
service center for the transaction monitor queue. This gives us the multi-class queueing network of Figure 4.2.

For all our experiments, the queueing network model was solved using simulation using QNAP2, a discrete-event simulator for queueing networks [82]. The details of the QNAP2 model descriptions are in Appendix D.

![TPCC-UVA queueing network model](image)

**Figure 4.2** TPCC-UVA queueing network model.

### 4.5 Experimental Results

The TPCC-UVA system was configured to run with 100 warehouses, each with 2 districts, i.e. 100x2 clients. The ramp-up period was 20 minutes and the measurement interval 2 hours, as specified by the TPC-C benchmark [113]. The database was initialized with data for 100x10 clients, as stated in Section 4.4.2. To measure the mean
DB page access time, the TPCC-UVA was run 5 different times (using the strace utility as stated in Section 4.4.1). The mean DB page access time of all 5 runs was used to parameterize the queueing network model.

To measure the TPCC-UVA transaction performance metrics the system was run another 5 times to collect response times for the transactions; these were averaged and compared to the simulation results. The 95% confidence intervals were obtained for the system and simulation results, but these were too tight to show on the graphs.

### 4.5.1 Transaction Mean Response Time and Mean Throughput

For the overall mean transaction response time the model underestimated the mean transaction response times by an average of 18.4% and hence, overestimated the performance. However, for mean response times per minute the model gave a better approximation. Figure 4.3(a) details the measured and modelled mean response times per minute during the measurement interval for the New-Order transaction. It can be seen from Figure 4.3(a) that the model underestimates the mean response time for the New-Order transaction; however, towards the end of the measurement interval, the measured response time slowly approached the modelled response time. This is apparent in Figure 4.3(b), in which the measurement interval was extended to 4 hours for one test run. In Figure 4.3(b) the measured system has become stable demonstrating good agreement between measured and modelled response times per minute.
The convergence of the measured system to the model is due to the fact that initially the system buffer is empty and as time passes it is populated by the transactions. Therefore, after a certain time, frequently accessed pages are resident in the buffer for all transactions, e.g. the WAREHOUSE and DISTRICT tables, which is when the system starts to stabilize and converge to the model.

Figure 4.3(c) compares the mean throughput per minute for the New-Order transaction during the 2 hour measurement interval. Since the model expressed shorter response times, it shows higher throughput than the measured throughput, giving an overestimation for the measured throughput. In Figure 4.3(d), in which the measurement interval was extended to 4 hours for one test run, the modelled throughput per minute
gives a better approximation of the measured throughput per minute. Results for the other transactions are similar.

### 4.5.2 Scalability

We have shown that the model is able to capture the steady-state performance of the TPCC-UVA system, giving a lower bound on the mean response time per minute of the transactions. From the results of the previous section, the TPCC-UVA system begins to stabilize about 120 minutes into the measurement interval. Therefore, for the following experiments, the ramp-up period was increased from 20 to 140 minutes and the measurement interval was one hour.

To establish the scalability of the model for different workloads the TPCC-UVA system was run 3 times to measure the mean DB page access time, and then it was run an additional 3 times to collect response times for the transactions. The experiment was conducted for 100 (100x1), 200 (100x2) and 300 (100x3) clients.

Figures 4.4 to 4.8 show the mean response time per minute for the one-hour measurement interval for all the transactions for these different workloads. It can be seen as the workload increases the system takes longer to stabilize. This is due to the increase in I/O activity of the TPCC-UVA database with the increase in the workload. The TPCC-UVA database index design forces a transaction to read large amounts of data into the buffer. This data is inadequate for other transactions due to the data distribution, e.g. customer data is unique for each district in each warehouse. Therefore, as the number of clients increases the amount of distinct data for each transaction increases, thereby decreasing the buffer hit rate per transaction.
Prior to system stability, the modelled mean response time per minute gives the lower bound on transaction response time per minute, irrespective of the workload. However, as the system shows signs of stability, the measured mean response time per minute approaches the modelled mean response time per minute. Therefore, the model scales to capture the steady state performance of the TPCC-UVA transactions.

Table 4.5 shows the measured and modelled mean response time per transaction for the different workloads calculated during the measurement interval. From Table 4.5 the model underestimates the mean response time for small workloads; this is due to the fact that processing time is the principal cost factor for transaction response time for small workloads. However, as the workload increases, and consequently the DB size increases, disk I/O time becomes the dominant cost factor, hence the model gives more accurate approximations. This is evident for the Stock-Level transaction. The Stock-Level transaction performs an SQL JOIN that is processed by the database using temporary tables [84], this is not considered in the cost metric for the model when calculating transaction service demands. Therefore, when processing time was prevalent, the model gave a high error rate for the Stock-Level transaction, relative to other transactions, however when disk I/O became prevalent the model accuracy rate increased for the Stock-Level transaction.
Figure 4.4 Comparison of the New-Order transaction mean response time per minute for different number of clients.

Figure 4.5 Comparison of the Payment transaction mean response time per minute for different number of clients.
Figure 4.6 Comparison of the Order-Status transaction mean response time per minute for different number of clients.

Figure 4.7 Comparison of the Delivery transaction mean response time per minute for different number of clients.
Figure 4.8 Comparison of the Stock-Level transaction mean response time per minute for different number of clients.

Table 4.5 Comparison of transaction mean response times for different number of clients.

<table>
<thead>
<tr>
<th># of Clients</th>
<th>Trans</th>
<th>Response Time (sec)</th>
<th>% Error per trans</th>
<th>% Overall Error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Measured</td>
<td>Modeled</td>
<td></td>
</tr>
<tr>
<td>100x1</td>
<td>New-Order</td>
<td>0.57</td>
<td>0.37</td>
<td>33.93</td>
</tr>
<tr>
<td></td>
<td>Payment</td>
<td>0.47</td>
<td>0.41</td>
<td>12.01</td>
</tr>
<tr>
<td></td>
<td>Order-Status</td>
<td>0.58</td>
<td>0.46</td>
<td>20.42</td>
</tr>
<tr>
<td></td>
<td>Delivery</td>
<td>0.73</td>
<td>0.43</td>
<td>40.63</td>
</tr>
<tr>
<td></td>
<td>Stock-Level</td>
<td>0.86</td>
<td>0.46</td>
<td>46.68</td>
</tr>
<tr>
<td>100x2</td>
<td>New-Order</td>
<td>18.28</td>
<td>16.25</td>
<td>11.08</td>
</tr>
<tr>
<td></td>
<td>Payment</td>
<td>18.20</td>
<td>16.29</td>
<td>10.50</td>
</tr>
<tr>
<td></td>
<td>Order-Status</td>
<td>18.31</td>
<td>16.27</td>
<td>11.13</td>
</tr>
<tr>
<td></td>
<td>Delivery</td>
<td>18.36</td>
<td>16.11</td>
<td>12.26</td>
</tr>
<tr>
<td></td>
<td>Stock-Level</td>
<td>18.79</td>
<td>16.24</td>
<td>13.59</td>
</tr>
<tr>
<td>100x3</td>
<td>New-Order</td>
<td>41.56</td>
<td>38.26</td>
<td>7.95</td>
</tr>
<tr>
<td></td>
<td>Payment</td>
<td>41.49</td>
<td>38.29</td>
<td>7.72</td>
</tr>
<tr>
<td></td>
<td>Order-Status</td>
<td>41.56</td>
<td>37.91</td>
<td>8.8</td>
</tr>
<tr>
<td></td>
<td>Delivery</td>
<td>41.62</td>
<td>38.01</td>
<td>8.69</td>
</tr>
<tr>
<td></td>
<td>Stock-Level</td>
<td>42.05</td>
<td>38.34</td>
<td>8.82</td>
</tr>
</tbody>
</table>
4.6 A Performance Comparison of Different Database Designs

The purpose of the database performance evaluation model is to provide the database designer with the ability to compare different database designs at database system design time. In this Section, we compare three different designs for the TPCC-UVA system.

4.6.1 The Database Design Descriptions

Using the database design of the TPCC-UVA application, we configured three different database designs to achieve different DB I/O page activity. This was conducted by changing the indexes on the CUSTOMER table given that it is the most accessed table. The three designs are:

- $I_1$: primary B-tree index on (warehouse_id, district_id, customer_id), and secondary b-tree index on (warehouse_id, district_id, customer_lastname);

- $I_2$: B-tree index on (warehouse_id, district_id, customer_id), this is the original design of the TPCC-UVA;

- $I_3$: B-tree index on (warehouse_id, district_id).

The indexes were chosen with regard to the way the transactions accessed the CUSTOMER table; the Payment and Order-Status transactions are invoked 60% of the time using the customer’s last name and 40% of the time using customer_id, the rest of
the transactions access the CUSTOMER table by customer_id, while the Stock-Level transaction does not access the CUSTOMER table.

These changes seem simple, but as can be seen in the following Section, they have a profound effect on the performance of the overall system. In the following Sections, we show how the TPCC-UVA database design was modelled for the different database designs: I₁, I₂, and I₃. The measurement of DB page access time was conducted as specified in Section 4.4.1.

To calculate the service demands for the queueing network models for the I₁, I₂, and I₃ database designs, we have used the same assumption as in Section 4.4.2 including the database initial loading size. The number of DB I/O pages for the designs I₁, I₂, and I₃ differ from those in Table 4.4 only for the CUSTOMER table; this is shown in Table 4.6. The values in Table 4.6 will be used for all our experiments regardless of the number of clients or the length of the execution run, as stated in Section 4.4.2. The service demand of a transaction on the relevant table is the calculated number of I/O DB pages needed by the transaction on that table × the mean time to access a DB page. Therefore, for the three designs, the transaction service demands will be the values in Table 4.6 multiplied by the mean DB page access time.

The queueing network performance model has the same structure as that of Figure 4.2 since the designs I₁, I₂, and I₃ differ from the original TPCC-UVA design in service demands, not in transaction processing or order of access to the tables.

For all these experiments, the queueing network model was solved using simulation using QNAP2. In addition, the experimental setup was that of Section 4.3.
Table 4.6 Number of I/O DB pages for the TPCC-UVA transactions.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
<th>V</th>
<th>VI</th>
<th>VII</th>
<th>VIII</th>
<th>IX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I₁</td>
<td>I₂</td>
<td>I₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>New-Order</td>
<td>0.75</td>
<td>3.04</td>
<td>2.33</td>
<td>2.33</td>
<td>251.26</td>
<td>-</td>
<td>4.34</td>
<td>3.98</td>
<td>47.6</td>
</tr>
<tr>
<td>Payment</td>
<td>2.75</td>
<td>3.04</td>
<td>6.89</td>
<td>152.93</td>
<td>253.26</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Order-Status</td>
<td>-</td>
<td>-</td>
<td>4.89</td>
<td>151.73</td>
<td>251.26</td>
<td>-</td>
<td>10.34</td>
<td>-</td>
<td>2.76</td>
</tr>
<tr>
<td>Delivery</td>
<td>-</td>
<td>-</td>
<td>43.3</td>
<td>43.3</td>
<td>253.26</td>
<td>-</td>
<td>43.4</td>
<td>39.8</td>
<td>47.6</td>
</tr>
<tr>
<td>Stock-Level</td>
<td>-</td>
<td>1.04</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>21.76</td>
</tr>
</tbody>
</table>

I = WAREHOUSE, II= DISTRICT, III= CUSTOMER, IV= HISTORY, V= ORDER, VI= NEW-ORDER, VII= ORDER-LINE, VIII= STOCK, IX= ITEM

4.6.2 Experimental Results

The TPCC-UVA system was configured to run with 100 warehouses, each with 2 districts, i.e. 100x2 clients for each design. The measurement interval was 120 minutes as specified by the TPC-C benchmark in which the system is in steady state. To determine the steady state for each design, the system was run with a ramp-up period of 20 minutes and a measurement interval of 6 hours for the I₁ design and a ramp-up period of 30 minutes and a measurement interval of 7 hours for the I₃ design. The mean response time per minute was plotted for the New-Order transaction. Figures 4.9 and 4.10 show the resulting graphs.

From Figures 4.9 and 4.10 the steady state for the two designs I₁ and I₃, is reached with a ramp-up period of 100 and 170 minutes, respectively. For I₂, the steady state is based on the results of the previous Section in which the steady state is reached with a ramp-up period of 140 minutes. The database was initialized with data for 100x10 clients, as stated in Section 4.4.2. To measure the mean DB page access time, the TPCC-UVA was run 3 different times for each design (using the strace utility as stated in Section 4.4.1). The mean DB page access time of all 3 runs was used to parameterize the queueing network model for each design.
Figure 4.9 New-Order transaction mean response time per minute for a ramp-up period of 20 minutes and measurement interval of 4 hours for 100x2 clients for the $I_1$ database design. The TPCC-UVA system starts to stabilize 80 minutes into the measurement interval, i.e. 100 minutes from the beginning of the system run.

Figure 4.10 New-Order transaction mean response time per minute for a ramp-up period of 30 minutes and measurement interval of 7 hours for 100x2 clients for the $I_3$ database design. The TPCC-UVA system starts to stabilize 140 minutes into the measurement interval, i.e. 170 minutes from the beginning of the system run.

To measure the TPCC-UVA transaction performance metrics the system was run another 3 times, for each design, to collect response times for the transactions. The response times were averaged and compared to the simulation results.
Figures 4.11 – 4.15 detail the measured and modelled mean response times per minute during the measurement interval for all the transactions for these three designs. It can be seen from Figures 4.11 – 4.15 that the best design, in terms of response time, is $I_1$ and the worst design is $I_3$. This is an intuitive result, since $I_1$ uses indexes that are tailored to the transaction usage. $I_2$ uses only one index; this forces the transactions that access the CUSTOMER table by customer_lastname to read all the DB pages of the relevant district from the CUSTOMER table. This is due to the fact that customer data is unique for each district in each warehouse. $I_3$ forces the transactions to access all the customer pages for the relevant district on any access to the table, whether by customer_id or customer_lastname.

From Figures 4.11 – 4.15, as well as from Table 4.7, it can be seen that the performance model gives an excellent approximation for the mean response time per minute for the transactions for $I_1$ (8% prediction error), but fails to achieve the same accuracy for $I_2$ (23% prediction error) and $I_3$ (25% prediction error), in which it gave the lower bound on response time. Nonetheless, these results are acceptable at design time.

**Figure 4.11** Comparison of the New-Order transaction mean response time per minute for 100x2 clients for the design (a) $I_1$ (b) $I_2$ (c) $I_3$.
Figure 4.12 Comparison of the Payment transaction mean response time per minute for 100x2 clients for the design (a) I₁ (b) I₂ (c) I₃.

Figure 4.13 Comparison of the Order-Status transaction mean response time per minute for 100x2 clients for the design (a) I₁ (b) I₂ (c) I₃.

Figure 4.14 Comparison of the Delivery transaction mean response time per minute for 100x2 clients for the design (a) I₁ (b) I₂ (c) I₃.

Figure 4.15 Comparison of the Stock-Level transaction mean response time per minute for 100x2 clients for the design (a) I₁ (b) I₂ (c) I₃.
4.6.3 Analysis

The performance model uses the mean DB page access time as a metric to calculate transaction response times, this is based on assuming that transaction access to DB pages is random; i.e. sequential access is rare. The response time of a transaction depends on the sequence of its DB page access requests and the time needed to fulfil these requests.

In order to investigate the effect of the TPCC-UVA database design change, we analyzed the DB page access trace (from the strace utility) for each of the three designs during the measurement interval. For each design, we looked at a trace for one run, in which we took a random sample of DB page access times for 4500 DB pages in sequence. Given that the TPCC-UVA has one transaction in the database at a time, this trace represents a sequence of transaction requests for table and index DB pages from disk. This is illustrated in Figure 4.16. In Figure 4.16 (and Figure 4.17), a DB page with
a long access time represents random I/O, while very short access times represent sequential I/O.

Figure 4.16 DB page access trace for 100x2 clients for the design (a) I₁ (b) I₂ (c) I₃.
Figure 4.17 CUSTOMER table DB page access trace for 100x2 clients for the design (a) I_1, (b) I_2, (c) I_3.

As can be seen from Figure 4.16(a), transaction access to DB pages is random, with few sequential accesses for I_1, this is expected due to the index design. For I_2, Figure 4.16(b), access is more evenly divided between random and sequential access. However, from Figure 4.16(c), for I_3, access is mostly sequential with few random accesses. This is apparent in Figure 4.17, in which a random sample of 4200 DB page access times from a trace of the transaction requests to the CUSTOMER table and its indexes for the three designs is shown. The effect of index design change can be seen, in which for I_1 access to the CUSTOMER table is random with rare sequential access (Figure 4.17(a)), while for I_3 it is sequential with rare random access (Figure 4.17(c)). The reason that I_3 displays such behaviour is due to the fact that the TPCC-UVA loads the CUSTOMER table in key sort order, therefore pages of customers of a certain
district are ordered logically and physically, causing sequential access. This is not a feature of real systems, customer data would be expected to be randomly distributed through the whole table, and therefore, lead to large random access when conducting a partial table scan, not large sequential access.

From Figure 4.16 and Figure 4.17, for I_3 short DB page response times are dominant per transaction: the disk head will move to the first page of the scan in the longest time and then sequentially scan the rest of the table in physical disk order. Thus, the access of the following DB pages will take significantly less time than the initial page. Since short response times are dominant per transaction, and therefore overall, the calculation of the DB page mean access time will favour the short responses. Hence, the calculated DB page mean access time will not accurately represent the effect of the initial random access to DB pages on transaction response time (this is formulated mathematically in Figure 4.18). Therefore, the calculated DB page mean access time will not accurately approximate the transaction mean DB page access behaviour. Consequently, the performance model will underestimate the transaction response time.

For I_2, sequential scans are not dominant, so the calculated DB page mean access time will give a better approximation of the transaction mean DB page access behaviour. Thus, the performance model gives a better estimate. For I_1, random access is dominant, and therefore, the performance model gives excellent results for transaction mean response times per minute.
Figure 4.18 The effect of large values on the mean of a population.

In conclusion, when access was overwhelmingly random with rare sequential access the performance model gives an excellent approximation of the mean response time. When the database design exhibited less random access and more sequential access the model tends to underestimate the mean response time, giving a lower bound on the mean response time. A good design, in general, will always consider more random access and less sequential access.

In general, good database designs favour random access to sequential access [84]. Full and partial table scans are avoided except when the table is very small and frequently
accessed, in which the performance lost is negligible in comparison to random I/O. A situation like $I_3$ is extremely rare and well beyond what is expected in actual DB systems. Hence, the use of the DB page mean access time as a metric in the performance model is suitable for realistic designs. Given that, if the rows of the CUSTOMER table were randomly distributed, the performance model would give results for $I_3$ similar to that of $I_1$.

### 4.7 Summary

In this Chapter, we have modelled the TPC-C benchmark using the queueing networks database design performance evaluation model. The performance model was validated against actual system runs of the TPCC-UVA open source implementation of the TPC-C benchmark. The experimental results indicate that this modelling technique has the ability to evaluate expected database system performance from database designs. It has been shown that the model was able to give the upper bound of system performance in the steady state for the TPCC-UVA implementation of the TPC-C benchmark for different workloads, with accuracy improving as the workload increased.

In addition, we have utilized the queueing network performance evaluation model in the performance comparison of different database designs for the TPCC-UVA system. The experimental results indicate that this modelling technique was able to give an excellent approximation of the system response time in the steady state for the TPCC-UVA implementation of the TPC-C benchmark for database designs with dominant random I/O DB page access.
In the next Chapter, we extend the database design queueing network performance evaluation model to incorporate database designs with active database rules.
Chapter 5

Modelling Active Database Rules

5.1 Introduction

In this Chapter, we model active database rules or triggers. Our definition of triggers is based on the SQL: 2003 standard [47]. The significance of representing triggers in a database performance model is important due to:

- the complexity of designing triggers in database systems [84],
- the fact that poor trigger designs are a cause of database performance problems [34], and
- it is difficult for a database designer to visualize the execution of triggers [85].

It is our belief that modelling database system design performance is not complete without the ability to represent triggers in the database design. The extension of our database design performance evaluation model to incorporate triggers is an improvement over previous modelling methods.

In the following sections, we extend the database design performance evaluation model to incorporate database triggers. We show the calculation of service demands for transactions that invoke triggers and illustrate the extended algorithm for calculating the
transaction path through the queueing network. Finally, we validate our model by comparing the results with a modified TPCC-UVA database design that incorporates an invocation of a trigger. The work in this Chapter has been described in [78].

5.2 Modelling Active Database Rules

An active rule or a trigger is a procedure that is run or activated by the DBMS when a certain event happens in the database [84]. Triggers are associated with events that occur in the form of INSERT, DELETE or UPDATE SQL statements on the tables of the database. A trigger is only activated when the event meets the condition of the trigger, i.e. a test condition or a query that evaluates to true (the result set is nonempty). When a trigger is run it performs an action that can be any set of SQL statements or procedural computations, depending on the DBMS implementation.

A trigger can be configured to execute before the event that applies changes to the database or after the changes are applied, these are referred to as BEFORE or AFTER triggers. In addition, the rate at which a trigger executes its action when activated can be defined. If an action is to be executed for each row modified by the event, then it is a row-level trigger. However, if it is defined to execute only once per activating event, then it is a statement-level trigger.

A transaction that contains a statement that will lead to trigger activation and execution is blocked until the trigger finishes successfully. Another option is to allow the execution of the trigger to be deferred to the end of the transaction execution or to execute instead of the activating statement, or asynchronously as part of another
transaction. Given that triggers execute in response to other actions on the database, they are considered part of the transaction that activates them. Hence, the activating transaction does not commit unless the trigger completes successfully (and all other triggers that are implicitly fired due to the actions of the initial trigger). We will only consider modelling blocking triggers. Deferred triggers can be modelled as blocking triggers at the end of the transaction. The model can be easily extended to instead of and asynchronous triggers.

Based on this, we represent triggers in our performance model as sub-transactions of the original transaction that invoked the trigger. The invoked trigger must complete first before the transaction can proceed with processing, i.e. we are modelling blocking triggers. Thus, a trigger’s service demands and traversal of the queueing network are calculated in the same manner as transactions. However, any transaction that invokes a trigger will have its path through the queueing network altered by the addition of the path of the activated trigger. For example, if a transaction accesses three tables, A, B, and C and the statements that access table B activate a BEFORE and AFTER trigger on that table, then the queueing network model for this transaction will be altered to represent the access of the BEFORE and AFTER triggers to table B as detailed in Figure 5.1.

Based on the SQL:2003 standard, we assume a trigger can have the same functionality as a transaction, i.e. there are no restrictions on the control statements or the SQL statements executed in a trigger. PostgreSQL allows this [102], however other DBMSs have some restrictions [71].
In the following Section, the formal specification of the queueing network model for database designs is modified to reflect the addition of triggers to the database design.

### 5.3 Extension of the Formal Specification for Triggers

The modifications to the formal definition presented in Section 3.3 are as follows:

- The definition of a table is modified by adding a `Trigger` attribute, which is defined accordingly.

- The algorithm to calculate the customer queueing network traversal path is modified to incorporate the invocation of BEFORE and AFTER triggers in the path. The algorithm was redesigned from that of Section 3.3 into a main algorithm that invokes a second recursive algorithm. All variables are global and parameters are assumed to be passed by reference. The recursive design of the second algorithm allows it to take into account triggers that activate triggers.
5.3.1 Trigger Formal Specification

As stated in Section 3.3, a database design can be formally described as $\text{DBDesign} = (\mathcal{R}, \mathcal{T})$, where $\mathcal{R}$ is the set of relations or tables and $\mathcal{T}$ is the set of transactions that access these tables. Define each table $r_i$ in $\mathcal{R}$ as:

$$r_i = (<\text{ordered set of}> A, I, [\text{uniqueness constraint}], \text{expected number of rows}, \text{average row length}, Trigger)$$

where:

- $A$ (set of attributes of $r_i$) and $I$ (the set of indexes of $r_i$) are as defined in Section 3.3.

- $Trigger$ is the set of triggers associated with the table.

Define each trigger $j$ in $\mathcal{T}$ as:

$$trigger_j = (\text{event}, \text{time}, \text{level}, <\text{ordered set of}> S, \text{trigDBpages}[r_i \in \mathcal{R}])$$

where:

- $\text{event}$ is the activating event: [UPDATE | INSERT | DELETE] SQL statement. There can only be one such triggering event per $trigger_j$. In addition, we assume that each table cannot have more than one trigger with the same event.

- $\text{time} = [\text{BEFORE} | \text{AFTER}]$, which specifies when the trigger action should execute, before or after the triggering event.
• \( level \) = [row | statement], which specifies if the trigger will execute for each row accessed by the triggering event or once after the triggering event. The value of \( level \) affects the service demand of the trigger.

• \( S \) is the ordered set of statements of the trigger and is defined as that of the transaction; this corresponds to the trigger action.

• \( \text{trigDBpages}[r_i \in \mathcal{R}] \), is the service demand of \( \text{trigger}_j \) on each \( r_i \in \mathcal{R} \), which can be defined as:

\[
\forall r_i \in \mathcal{R}, \text{trigDBpages}[r_i \in \mathcal{R}] = \sum_{k=1}^{|[S]|} s_k \cdot \text{statDBpages}[r_i],
\]

where \( s_k \) is the \( k^{th} \) statement in the trigger. This formula calculates the expected number of database pages that the trigger will use, in isolation. The final number depends on the invoking transaction and whether the trigger is a row-level or a statement-level trigger.

5.3.2 Calculating Service Demands for Transactions that Invoke Triggers

The service demand for a trigger depends on the invoking transaction and whether the trigger is a row-level or statement-level trigger. Therefore, if a transaction \( t_j \) fires a trigger \( \text{trigger}_j \) that in turn accesses a table \( r_i \), then the service demands of \( \text{trigger}_i \) on table \( r_i \) depend on the query \( q \) firing the trigger and the statement \( s_k \) it resides in.
The calculation of the service demands for trigger$_i$ that is invoked by $t_j$ are based on our initial assumption that if a transaction accesses a table in multiple SQL statements, these statements are in sequence (see Section 3.3.1). The consequence of this assumption is that if a transaction $t_j$ accesses table $r_i$, then accesses table $r_{i+1}$, and in the process fires a BEFORE trigger, trigger$_i$, on table $r_{i+1}$, and trigger$_i$ in turn accesses table $r_{k}$, (Figure 5.2), then one of the following must hold:

$$r_k = r_i \text{ or } r_k = r_{i+1} \text{ or } r_k = r_n \text{ where } r_n \in R \text{ is not accessed by } t_j.$$ 

The same applies if $r_i$ has an AFTER trigger that accesses table $r_k$ and is invoked by $t_j$.

![Figure 5.2 A BEFORE trigger invocation.](image)

In addition, the assumption that all DB pages of a transaction will be in the buffer until the transaction commits results in that the first access to a table’s data is the significant access, which will be the service demand on the table. Any subsequent access during the execution of the transaction will have a service demand of zero. However, if subsequent accesses access different pages, than that will be added to the initial service demand. The general formulas given below assume that the trigger will access different pages than that of the invoking transaction.
This assumption simplifies the calculation of the service demands, thus serving to explain the concept without the complicated details. In addition, it simplifies the calculation of the routing path of the transaction for the queueing network model. The general formula, in which access of a trigger to a table is not restricted, can be modelled by changing the customer type when a trigger is invoked and returning to the original customer type after the trigger completes execution. This will not affect the queueing network routing table with regard to the overall table access, nor will it affect the overall service demands on the tables. However, the calculation of the transaction service demands for tables accessed by both the transaction and the trigger will now be divided between them.

For each transaction $t_j$, the trigger service demands depend on the type of statement $s_k$ in $S$ that make up $t_j$. The algorithm for calculating the transaction service demand when the trigger is invoked from a query $q$ is:

\[
\begin{align*}
\text{if } s_k &= q \text{ then} \\
&\forall r_i \in q.\text{Access} \\
&\forall \text{trigger}_n \in r_i \\
&\quad \text{if (r}_i.\text{Trigger is not NULL) and (r}_i.\text{trigger}_n.\text{event} = q.\text{type} ) \text{ then} \\
&\quad\quad \text{if r}_i.\text{trigger}_n.\text{level} = \text{statement} \text{ then} \\
&\quad\quad\quad t_j.\text{tranDBpages}[r_i \in R] = t_j.\text{tranDBpages}[r_i \in R] + \text{trigger}_n.\text{trigDBpages}[r_i \in R] \\
&\quad\quad\quad (\# \text{ of rows accessed by } q) \times \text{trigger}_n.\text{trigDBpages}[r_i \in R] \\
&\quad\quad\quad \text{else (row-level trigger)} \\
&\quad\quad\quad\quad t_j.\text{tranDBpages}[r_i \in R] = t_j.\text{tranDBpages}[r_i \in R] + \\
&\quad\quad\quad\quad (\# \text{ of rows accessed by } q) \times \text{trigger}_n.\text{trigDBpages}[r_i \in R] \\
&\quad\quad\quad\quad \text{end if} \\
&\quad\quad\quad \text{end if} \\
&\quad\quad \text{end if} \\
&\quad \text{end if}
\end{align*}
\]

The algorithm for calculating the transaction service demand when the trigger is invoked from within a loop is:
if $s_k = \text{loop}$, and $N$ is the number of loop iterations, then

$$\forall q_m \in \text{loop}
\forall r_i \in q_m, \text{Access}
\forall \text{trigger}_n \in r_i$$

if ($r_i$ . Trigger is not NULL) and ($r_i$ . trigger$'_n$ . event = $q_m$ . type )

then

if ($r_i$ . trigger$'_n$ . level = statement )

then

$$t_j . \text{tranDBpages}[r_i \in R] = t_j . \text{tranDBpages}[r_i \in R] + N \times \text{trigger}_n . \text{trigDBpages}[r_i \in R]$$

else (row-level trigger)

$$t_j . \text{tranDBpages}[r_i \in R] = t_j . \text{tranDBpages}[r_i \in R] + N \times \text{(# of rows accessed by } q_m) \times \text{trigger}_n . \text{trigDBpages}[r_i \in R]$$

end if

end if

end if

end if

end if

The algorithm for calculating the service demand when the trigger is invoked from within a branch statement is:

if $s_k = \text{branch}$, then

$$\forall \text{branch}_i
\forall q_m \in \text{branch}_i
\forall r_i \in q_m, \text{Access}
\forall \text{trigger}_n \in r_i$$

if ($r_i$ . Trigger is not NULL) and ($r_i$ . trigger$'_n$ . event = $q_m$ . type )

then

if ($r_i$ . trigger$'_n$ . level = statement )

then

$$t_j . \text{tranDBpages}[r_i \in R] = t_j . \text{tranDBpages}[r_i \in R] + \text{trigger}_n . \text{trigDBpages}[r_i \in R]$$

else (row-level trigger)

$$t_j . \text{tranDBpages}[r_i \in R] = t_j . \text{tranDBpages}[r_i \in R] + \text{(# of rows accessed by } q_m) \times \text{trigger}_n . \text{trigDBpages}[r_i \in R]$$

end if

end if

end if
5.3.3 Calculating the Routing Path

In order to simplify the routing path algorithm, it is assumed that when $s_k = \text{branch}$, i.e. a branch statement, that $\forall \ branch_i \in \text{branch}$ the following holds:

- the first table accessed in $branch_i$ cannot activate a BEFORE trigger and
- the last table accessed in $branch_i$ cannot activate an AFTER trigger.

Consequently if $branch_i$ accesses only one table, than that table cannot activate any triggers.

- All other tables in $branch_i$ cannot activate a branch statement as a BEFORE trigger or as an AFTER trigger.

The algorithm can be easily extended to include these previous cases. Algorithm 5.1 and 5.2 detail the calculation of the queueing network traversal path for a database design with BEFORE and AFTER triggers. The additions to the original algorithm in Section 3.3 are highlighted.
Algorithm 5.1: Calculating Customer Class Path

1: Let $current_table$ be the current table in the path of $t_i$
2: $current_table \leftarrow 0$
3: Let $branch[n]$ be a vector of $r_k \in R$, that holds the last table accessed by a branch statement, where $n$ is the number of branches
4: $branch[] \leftarrow nil$ (element by element assignment)
5: Let $bran_table$ be the current table of a branch statement
6: $bran_table \leftarrow 0$
7: Let prev_branch[] be a vector that holds the initial value of $branch[]$
8: prev_branch[] ← nil
9: $\forall t_i \in T$
10: $\forall s_j \in t_i.S$
11: ConnectPath($s_j$, $current_table$, $branch[]$)
12: if $s_j$ is the last statement in $t_i$ then
13: if $branch[] \neq nil$ then
14: for branch[1] to $\mid branch[] \mid$ do
15: $P[c_i, branch[], 0] \leftarrow 1$
16: end for
17: else
18: $P[c_i, current_table, 0] \leftarrow 1$
19: end if
20: end if
21: (the transaction leaves the network after leaving the last table accessed by the last SQL statement of the final statement)
22: end algorithm

Algorithm 5.2: Function ConnectPath

Function ConnectPath($s_j$, $current_table$, $branch[]$)
1: case $s_j = q$
2: $\forall r_k \in q.Access$
3: if ($r_k.Trigger$ is not NULL) and
4: ($\exists r_k.trigger.time = BEFORE$) and
5: ($r_k.trigger.event = q.type$)
6: then
7: $\forall s_i \in trigger_i.S$
8: ConnectPath($s_j$, $current_table$, $branch[]$)
9: end if –before trigger
10:
11: if ($r_k$ is first table accessed by $q$) and $branch[] \neq nil$ then
12: (connect the last tables accessed by the previous branch statement to the first table of this SQL statement)
13: for branch[1] to $\mid branch[] \mid$ do
14: $P[c_i, branch[, r_k] \leftarrow 1$
15: end for
16: $branch[] \leftarrow nil$
17: $current_table \leftarrow r_k$
else
    \[ P[c_i, current_table, r_k] \] ← 1
    current_table ← \[ r_k \]
end if

if (\( r_k \).Trigger is not NULL) and
    \( (\exists \ r_k \cdot \text{trigger}.time = \text{AFTER}) \) and
    \( (r_k \cdot \text{trigger}.event = q.m\cdot\text{type}) \)
then
    \( \forall s_j \in \text{trigger} \cdot S \)
    ConnectPath(s_j, current_table, branch[])
end if –after trigger

case \( s_j = \text{loop} \)
\( \forall q_m \in \text{loop} \)
\( \forall r_k \in q_m\cdot\text{Access} \)
if (\( r_k \).Trigger is not NULL) and
    \( (\exists r_k \cdot \text{trigger}.time = \text{BEFORE}) \) and
    \( (r_k \cdot \text{trigger}.event = q_m\cdot\text{type}) \)
then
    \( \forall s_j \in \text{trigger} \cdot S \)
    ConnectPath(s_j, current_table, branch[])
end if –before trigger

if (\( q_m \) is first SQL statement) and (\( r_k \) is first table accessed by \( q_m \)) and \( \text{branch}[] \neq \text{nil} \)
(\( \text{connect the last tables accessed by the previous branch statement to the first table of this SQL statement} \))

for \( \text{branch}[] \) to | \( \text{branch}[] \) | do
    \[ P[c, \text{branch}[], r_k] \] ← 1
end for

\( \text{branch}[] \leftarrow \text{nil} \)
\( \text{current_table} \leftarrow r_k \)
else
    \[ P[c_i, current_table, r_k] \] ← 1
    current_table ← \[ r_k \]
end if

if (\( r_k \).Trigger is not NULL) and
    \( (\exists r_k \cdot \text{trigger}.time = \text{AFTER}) \) and
    \( (r_k \cdot \text{trigger}.event = q_m\cdot\text{type}) \)
then
    \( \forall s_j \in \text{trigger} \cdot S \)
    ConnectPath(s_j, current_table, branch[])
end if –after trigger

\( P[c_i, 0, r_k] \) gives the entry server for \( t_i \). Unassigned values take the value zero.
60:  case \( s_j = \text{branch} \)
61:    \( \text{prev}_branch[i] \leftarrow \text{branch}[i] \)
62:  for \( i \) in \( n \) do (total number of branches)
63:    \( \text{bran}_table \leftarrow \text{current}_table \)
64:    \( \forall q_m \in \text{branch}_i \)
65:    \( \forall r_k \in q_m.\text{Access} \)
66:      if \( \text{r}_k.\text{Trigger} \) is not NULL and
67:        ( \( \exists r_k.\text{trigger}_i.\text{time} = \text{BEFORE} \) and
68:           ( \( r_k.\text{trigger}_i.\text{event} = q_m.\text{type} \))
69:        then
70:          \( \forall s_j \in \text{trigger}_i.\text{S} \)
71:          \( \text{ConnectPath}(s_j, \text{bran}_table, \text{null}) \)
72:        \( \text{branch}[i] \) is null due to the assumption that the first
73:         table in \( \text{branch}_i \) does NOT have a \text{BEFORE} trigger)
74:      end if –before trigger
75:  else
76:    if \( q_m \) is first SQL statement and \( \text{r}_k \) is first table accessed by \( q_m \) then
77:      Let \( p_i \) be the probability of accessing \( \text{branch}_i \)
78:      if \( \text{prev}_branch[i] \neq \text{nil} \) then
79:        (connect the last tables accessed by the previous branch
80:         statement to the first table of this branch’s SQL statement)
81:          for \( \text{prev}_branch[i] \) to \( |\text{prev}_branch[i]| \) do
82:            \( P[c_i, \text{prev}_branch[i], r_k] \leftarrow p_i \)
83:          end for
84:          \( \text{bran}_table \leftarrow r_k \)
85:        else
86:          \( P[c_i, \text{bran}_table, r_k] \leftarrow 1 \)
87:          \( \text{bran}_table \leftarrow r_k \)
88:        end if
89:      else
90:        if \( \text{r}_k.\text{Trigger} \) is not NULL and
91:          ( \( \exists r_k.\text{trigger}_i.\text{time} = \text{AFTER} \) and
92:            ( \( r_k.\text{trigger}_i.\text{event} = q_m.\text{type} \))
93:          then
94:            \( \forall s_j \in \text{trigger}_i.\text{S} \)
95:            \( \text{ConnectPath}(s_j, \text{bran}_table, \text{null}) \)
96:          end if –after trigger
97:        \( \text{branch}[i] \leftarrow r_k \)
98:      end for
99:    end case
100:  end function algorithm
5.4 TPCC-UVA Trigger Performance Modelling

Due to the limitations of the experimental setup, we have seen that designs with triggers whose actions lead to the invocation of other triggers lead to rapid system saturation and stability is never achieved. This is demonstrated in Figure 5.3 in which an AFTER INSERT trigger on the HISTORY table invokes an AFTER UPDATE trigger on the ORDERCopy table (see below). Figure 5.3 shows the response time of the New-Order transaction for 100x1 clients. Therefore, the experiments were restricted to modelling simple trigger designs.

![Graph showing response time for New-Order transaction](image)

**Figure 5.3** New-Order transaction mean response time per minute for a ramp-up period of 20 minutes and measurement interval of 480 minutes for 100x1 clients.

For this experiment, we changed the design of the TPCC-UVA system by adding an AFTER INSERT trigger on the HISTORY table. This trigger can only be invoked by the Payment transaction, which is the only transaction that inserts rows into the HISTORY table.
In order not to affect the data in the other tables, a new table was created, ORDERCopy, which is an exact duplicate of the original ORDER table including index and keys. There are two scenarios for the design of the AFTER INSERT trigger on the HISTORY table, trigger1 and trigger2, respectively.

**Trigger1** updates the o_carrier_id field of the ORDERCopy table for 300 orders of the corresponding district of the triggering INSERT statement on HISTORY. Figure 5.4 shows the details of trigger1, where new.h_w_id and new.h_d_id correspond to the values inserted into the HISTORY table by the Payment transaction.

```sql
CREATE OR REPLACE FUNCTION update_ORDERCopy() RETURNS TRIGGER AS $trigger1$
BEGIN
    update ORDERCopy set o_carrier_id=1 where o_w_id = new.h_w_id and o_d_id = new.h_d_id and o_id between 1500 and 1800;
    RETURN new;
END;

Figure 5.4 Details of trigger1: AFTER UPDATE trigger on HISTORY.
```

**Trigger2** counts the number of orders of the corresponding district of the triggering INSERT statement on HISTORY using a SELECT statement. Figure 5.5 details trigger2. In addition, new.h_w_id and new.h_d_id correspond to the values inserted into the HISTORY table by the Payment transaction.
Figure 5.5 Details of trigger2: AFTER INSERT trigger on HISTORY.

Table 5.1 shows the number of DB pages for the queueing network model for the TPCC-UVA database design. The values in Table 5.1 are calculated in the same manner as those in Section 4.4.2, the difference is in the DB pages used by trigger1 and trigger2 (shown as t1 and t2 in Table 5.1) on the ORDERCopy table. The calculation methods for the service demands for trigger1 and trigger2 are similar to those presented in Appendix C.

Using the algorithm in the previous section, the corresponding queueing network model for the TPCC-UVA design with a trigger access to ORDERCopy table is in Figure 5.6. The queueing network model is identical for the TPCC-UVA design with trigger1 or trigger2. The difference between the designs is in the service demands on the ORDERCopy table; however, the traversal of the queueing network is similar for both designs.
Table 5.1 Number of I/O DB pages for the TPCC-UVA transactions.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
<th>V</th>
<th>VI</th>
<th>VII</th>
<th>VIII</th>
<th>IX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>New-Order</td>
<td>0.75</td>
<td>3.04</td>
<td>2.33</td>
<td>-</td>
<td>4.34</td>
<td>3.98</td>
<td>47.6</td>
<td>44.7</td>
<td>17.1</td>
<td>-</td>
</tr>
<tr>
<td>Payment</td>
<td>2.75</td>
<td>3.04</td>
<td>152.93</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4.34</td>
<td>10.34</td>
</tr>
<tr>
<td>Order-Status</td>
<td>-</td>
<td>-</td>
<td>151.73</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Delivery</td>
<td>-</td>
<td>-</td>
<td>43.3</td>
<td>-</td>
<td>43.4</td>
<td>39.8</td>
<td>47.6</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Stock-Level</td>
<td>-</td>
<td>1.04</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>21.76</td>
<td>201.47</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

I = WAREHOUSE, II= DISTRICT, III= CUSTOMER, IV= HISTORY, V= ORDER, VI= NEW-ORDER, VII= ORDER-LINE, VIII= STOCK, IX= ITEM, X= ORDERCopy

Figure 5.6 TPCC-UVA queueing network model with ORDERCopy table.

5.4.1 Experimental Results

The TPCC-UVA system was configured to run with 100 warehouses, each with 2 districts, i.e. 100x2 clients for each design: trigger1 and trigger2. The measurement interval was 120 minutes, as specified by the TPC-C benchmark in which the system is in the steady state. To determine the steady state for the designs trigger1 and trigger2, the system was run with a ramp-up period of 20 minutes and a measurement interval of 8 hours for the trigger1 design and a ramp-up period of 20 minutes, and a measurement
interval of 5 hours for the \textit{trigger2} design. The mean response time per minute was plotted for the New-Order transaction. Figures 5.7 and 5.8 show the resulting graphs.

To reach the steady state for the designs \textit{trigger1} and \textit{trigger2} ramp-up periods of 160 and 150 minutes, respectively, were used, as can be seen from Figures 5.7 and 5.8. The database was initialized with data for 100x10 clients, as stated in Section 4.4.2. To measure the mean DB page access time, the TPCC-UVA was run 3 different times for each design (using the strace utility). The mean DB page access time of all 3 runs was used to parameterize the queueing network model for each design.

To measure the TPCC-UVA transaction performance metrics the system was run another 3 times, for each design, to collect response times for the transactions. The response times were averaged and compared to the simulation results. The 95\% confidence intervals were obtained for the system and simulation results, but these were too tight to show on the graphs.
Figure 5.7 New-Order transaction mean response time per minute for a ramp-up period of 20 minutes and measurement interval of 480 minutes for 100x2 clients. The TPCC-UVA system with trigger1 starts to stabilize 140 minutes into the measurement interval, i.e. 160 minutes from the beginning of the system run.

Figure 5.8 New-Order transaction mean response time per minute for a ramp-up period of 20 minutes and measurement interval of 300 minutes for 100x2 clients. The TPCC-UVA system with trigger2 starts to stabilize 130 minutes into the measurement interval, i.e. 150 minutes from the beginning of the system run.
Figures 5.9 – 5.13 detail the measured and modelled mean response times per minute during the measurement interval for the five transactions for the TPCC-UVA trigger1 and trigger2 designs. Table 5.2 shows the measured and modelled mean response time per transaction for the two designs calculated during the measurement interval. We would expect trigger1 to have better performance than trigger2, given that the design for trigger2 accesses more DB pages than trigger1 on the ORDERCopy table (see Table 5.1). However, the performance of trigger2 was 20% better than that of trigger1. This is due to the fact that processing time increases for the trigger response time when executing an UPDATE statement in relation to when executing a SELECT statement.

It can be seen from Figures 5.9 – 5.13 and Table 5.2 that the performance model gives an excellent approximation of the mean response time per minute for the transactions for trigger2 (approximately 18% prediction error), but fails to achieve the same accuracy for trigger1 (approximately 39% prediction error). The improved prediction for trigger2 is related to the predominance of DB I/O time in the overall trigger response time. However, for trigger1, where processing time is predominant in the overall trigger response time, the performance model deviated from giving an accurate estimation. Given that the performance model does not take processing demands into consideration this result was to be expected.
Figure 5.9 Comparison of the New-Order transaction mean response time per minute for the TPCC-UVA with (a) trigger1 and (b) trigger2 designs for 100x2 clients and measurement interval of 120 minutes.

Figure 5.10 Comparison of the Payment transaction mean response time per minute for the TPCC-UVA with (a) trigger1 and (b) trigger2 designs for 100x2 clients and measurement interval of 120 minutes.

Figure 5.11 Comparison of the Order-Status transaction mean response time per minute for the TPCC-UVA with (a) trigger1 and (b) trigger2 designs for 100x2 clients and measurement interval of 120 minutes.
Figure 5.12 Comparison of the Delivery transaction mean response time per minute for the TPCC-UVA with (a) trigger1 and (b) trigger2 designs for 100x2 clients and measurement interval of 120 minutes.

Figure 5.13 Comparison of the Stock transaction mean response time per minute for the TPCC-UVA with (a) trigger1 and (b) trigger2 designs for 100x2 clients and measurement interval of 120 minutes.

Table 5.2 Comparison of transaction mean response times for TPCC-UVA with trigger1 and trigger2 designs.

<table>
<thead>
<tr>
<th>Design</th>
<th>Transaction</th>
<th>Response Time (sec)</th>
<th>Response Time (sec)</th>
<th>% Error per trans</th>
<th>% Overall Error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Measured</td>
<td>Modelled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>trigger1</td>
<td>New-Order</td>
<td>33.32</td>
<td>20.52</td>
<td>38.41</td>
<td>38.46</td>
</tr>
<tr>
<td></td>
<td>Payment</td>
<td>33.19</td>
<td>20.55</td>
<td>38.08</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Order-Status</td>
<td>33.28</td>
<td>20.65</td>
<td>37.96</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Delivery</td>
<td>33.36</td>
<td>20.5</td>
<td>38.55</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stock-Level</td>
<td>33.96</td>
<td>20.61</td>
<td>39.3</td>
<td></td>
</tr>
<tr>
<td>trigger2</td>
<td>New-Order</td>
<td>26.85</td>
<td>22.14</td>
<td>17.56</td>
<td>17.75</td>
</tr>
<tr>
<td></td>
<td>Payment</td>
<td>26.71</td>
<td>22.17</td>
<td>16.99</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Order-Status</td>
<td>26.88</td>
<td>22.21</td>
<td>17.36</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Delivery</td>
<td>26.93</td>
<td>22.14</td>
<td>17.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stock-Level</td>
<td>27.42</td>
<td>22.21</td>
<td>19.02</td>
<td></td>
</tr>
</tbody>
</table>
5.5 Summary

In this Chapter, an extension of the database design queueing network performance evaluation model for active database rules was presented. The formal specification for database triggers was given. In addition, a calculation of the service demands for triggers and transactions that activate triggers was also presented. A modified algorithm to calculate the path of a transaction that invokes a trigger through the queueing network was given.

The experimental results have shown that the performance model can give an accurate estimation of the mean response time for database designs in which triggers have predominant I/O processing. This is in agreement with the results previously discussed in Chapter 4, where it was shown that the performance model is applicable to designs of large databases where random disk I/O is the dominant cost factor and in which processing is negligible.

In the next Chapter, the database design queueing network model is extended to incorporate referential integrity checking.
Chapter 6

Modelling Referential Integrity

6.1 Introduction

In this Chapter, we model referential integrity or foreign key checking in database systems. Foreign keys are used to maintain a parent/child relationship between tables. Referential integrity checking is implemented in a way very similar to triggers in DBMS except that referential integrity checks are system invoked. The importance of modelling referential integrity checks in a database performance model is due to the fact that such checks incur performance costs on the database [10, 72].

In the following sections, we extend the database design performance evaluation model to incorporate referential integrity checking. We show the calculation of service demands for transactions that invoke foreign key checks and illustrate the extended algorithm for calculating the transaction path through the queueing network. To validate our performance evaluation model we modify the TPCC-UVA database design by including a parent/child relationship and compare the system performance to our model results.
6.2 Modelling Referential Integrity Checking

Referential integrity checks are implemented in DBMS as system invoked procedures. A referential integrity check means that another table or tables are read by the DBMS to check the existence of the value of the referenced field. Due to the similarity in execution between triggers and referential integrity checks, we model referential integrity checks similarly to modelling AFTER triggers. The main difference is that in the majority of DBMS a referential integrity check from the child table to the parent table is an index scan on the primary key index of the parent table [10, 44, 72], as the foreign key must match a value of the primary key. Hence, referential integrity checks incur no table access. However, in PostgreSQL, we have noticed when looking at the TPCC-UVA system statistics with foreign key references, that the actual DB page was read when PostgreSQL performed a referential integrity check.

Referential integrity maintains a parent/child relationship between the referenced table and the referencing table. A DBMS handles referential integrity enforcement depending on the operations that cause the foreign key checks. These are [84]:

(a) operations on the parent table: which would be a DELETE/UPDATE of the referenced field. The options provided by the DBMS are:

- CASCADE: DELETE/UPDATE all child references,

- DISALLOW: prevent the operation as long as a child row exists,

and
- DEFAULT VALUE: update the foreign key of the child rows with a default value, including NULL.

For the CASCADE and DEFAULT VALUE cases, a table access will happen in order to execute the operation on the child rows. However, for the DISALLOW operation a table access will be needed only if the foreign key column is not indexed.

(b) operations on the child table: an INSERT/UPDATE of a foreign key field. The inserted/updated foreign key value is checked against the referenced field in the parent table. The operation is rejected if the inserted/updated field value does not exist in the parent table.

The time when the DBMS checks the referential integrity constraints can be specified when defining a foreign key on a table. The options are:

- IMMEDIATE: check immediately after SQL statement execution, or

- DEFERRED: defer checking until transaction commit time.

To model foreign key referencing, we model them as we have modelled triggers, i.e. as sub-transactions that are part of the invoking transaction. For simplicity, we assume the mode on the parent is to DISALLOW an operation violating referential integrity, if a child row exists. Therefore, we do not consider the effect of this in our model. However, the model can be easily extended to allow for such conditions.

To model a foreign key check in the IMMEDIATE execution mode, we model the referential integrity check on the parent table as another table access after leaving the
invoking child server. For example, if a transaction accesses three tables, A, B and C and the statements that access tables A and B both cause a referential integrity check to parent tables A’ and B’. Then the queueing network model for this transaction will be altered to represent the referential integrity check, i.e. the access of the parent tables A’ and B’, as detailed in Figure 6.1.

**Figure 6.1** A queueing network model with IMMEDIATE referential integrity checking.

For referential integrity checking in the DEFERRED mode, the referential integrity check for all parent tables will be in sequence after the last table accessed in the transaction. Using the previous example, the referential integrity check to the parent tables A’ and B’ will happen before transaction commit, i.e. before the transaction leaves the queueing network. Thus the queueing network model for the transaction will be altered to represent the referential integrity check, i.e. access of the parent tables A’ and B’, as detailed in Figure 6.2.

In the following section, the formal specification of the queueing network model of Section 3.3 is modified to reflect the addition of foreign keys to the database design.
6.3 Extension of the Formal Specification for Foreign Keys

The modifications to the formal definition presented in Section 3.3 are as follows:

- The definition of a table is modified by adding a \(FK\) (foreign key) attribute, which is defined in the next Section.

- The algorithm to calculate the customer queueing network traversal path is modified to incorporate the invocation of referential integrity checks in the path. The algorithm was redesigned from that of Section 3.3 into a main algorithm that invokes a second recursive algorithm. All variables are global and parameters are assumed to be passed by reference.

6.3.1 Referential Integrity Formal Specification

As stated in Section 3.3, a database design can be formally described as DBDesign = \((R, T)\), where \(R\) is the set of relations or tables and \(T\) is the set of transactions that access these tables. Define each table \(r_i\) in \(R\) as:
\[ r_i = (\langle \text{ordered set of} \rangle A, I, [\text{uniqueness constraint}]^*, \text{expected number of rows}, \]

\[ \text{average row length}, FK \rangle ) \]

where:

- \( FK \) is the set of referential integrity constraints associated with the table.

Define each \( f_k \) in \( FK \) as:

\[ f_k = (r_k, \text{ref\_attribute}, \text{mode}, \langle \text{ordered set of} \rangle S, \text{FKDBpages}[r_k] ) \]

where:

- \( r_k \in R \) is the parent table.

- \( \text{ref\_attribute} \) is the uniqueness constraint of \( r_k \), i.e. the referenced column or attribute.

- \( \text{mode} \) is the time the referential integrity check is made: \([\text{IMMEDIATE} \mid \text{DEFERRED}]\).

- \( S \) is the ordered set of statements that will be executed on the parent table, in this case \( S = s_1 = q \), such that:

  o \( q \) is a SQL statement and can be described as:

  \[ q = (\text{type}, \langle \text{ordered set of} \rangle \text{Access}, \text{DBpages}[r_k] ) , \]
and given that a referential integrity check is an implicit SELECT:

- **type** is a SELECT statement where the condition clause and the retained attribute refer to the referenced column;

- **Access** is the parent table \( r_k \in \mathcal{R} \);

- \( \text{DBpages}[r_k] \) is calculated as previously stated in reference to the parent table \( r_k \) it accesses; which will be a primary index scan for the foreign key.

Even though a reference check is not an actual query, we use this notation in order to be compatible with the routing algorithm and for calculation purposes (implicit SELECT resolved by index scan).

For the special case of referential integrity:

\[
\text{FKDBpages}[r_k] = s_j.\text{statDBpages}[r_k] = q.\text{DBpages}[r_k]
\]

This formula calculates the expected number of DBpages that a referential integrity check will use, in isolation. The final number depends on the invoking transaction and the number of checks needed.
6.3.2 Calculating Service Demands for Transactions that Invoke Referential Integrity Checks

If a transaction \( t_j \) has a foreign key \( fk_i \) on a parent table \( r_i \), then the service demands for the referential check of \( fk_i \) on table \( r_i \) depend on the number of rows of the query \( q \) that invoke the check, as well as the statement \( s_k \) that \( q \) resides in. In addition, the calculation of referential integrity service demands is based on the assumption that if a transaction accesses a table in multiple SQL statements, these statements are in sequence (see Section 3.3.1) and that buffering is on a transaction-by-transaction basis.

Consider that we have two tables: A the parent table and B the child table which references A. Based on the previous assumptions, if a transaction accesses table A and B, then a referential check is allowed only in the following scenarios:

- **A primary index access to A, then an UPDATE/INSERT to B causing a reference check to rows in A.** Given that the referential integrity check will need an index scan only and since the index of A will already be in the buffer, this scenario will not add any extra service demands for the transaction, i.e. extra service demands on A due to the reference check is nil. Hence, the queueing network model will not change due to the addition of a referential integrity check, nor will the service demands of the transaction. In consequence, the SQL statements that access B do not have to strictly be in sequence to SQL statements that access A.
• An access to B that causes a reference check on A, that is followed by a primary index access to A. This will only incur the cost of allocating the rows of A since the index will already be in the buffer. Therefore, the total service demand on A will be the index access service demand plus the row access service demand. In this scenario, the SQL statements that access A must strictly be in sequence to the SQL statements that access B.

• Access to B which invokes a referential integrity check to a table A and A is never directly accessed by the transaction. In this case, the referential check will add a new server to the queueing network with new service demands to the transaction. In this case the referential integrity check on A will automatically follow the access to B.

• Access to B where the foreign key reference is recursive, i.e. B references columns in the same table B. In this case the index is already in the buffer, the reference check will not add an additional service demand to table B, and there will be no changes to the queueing network.

The implication of the above scenarios is that if a transaction $t_j$ accesses a set of tables, and $\exists r_k \in R$ which references a parent table $r_i$, then one of the following must hold:

$$r_k = r_{i-1} \text{ or } r_k = r_i \text{ or } r_k \geq r_{i+1} \text{ or } r_i = r_n \text{ where } r_n \in R \text{ is not accessed by } t_j.$$

In addition, if the referential integrity is in DEFERRED mode then only the following holds:
\[ r_i = r_n \text{ where } r_n \in R \text{ is not accessed by } t_j \]

Building on the previous scenarios, for each transaction \( t_j \) that accesses a set of tables \( R' \), the reference check service demand does not differ depending on the type of \( s_k \) in \( S \) that make up \( t_j \) when \( r_i \in R' \). The algorithm for calculating the service demand when \( s_k = q \) is:

\[
\begin{align*}
\text{if } s_k = q, \text{ then} \\
\forall r_k \in q.\text{Access} \\
\forall f_{k_i} \in r_k \\
\text{if } (q.\text{type} \text{ in } (\text{INSERT|UPDATE})) \text{ and } \\
(\exists q.a_j = f_{k_i}.\text{ref_attribute}) \text{ then} \\
\text{if } (r_i \not\in R') \text{ then} \\
\begin{equation*}
t_j.\text{tranDBpages}[r_i \in R] = f_{k_i}.\text{FKDBpages}[r_i \in R] \tag{1}
\end{equation*} \\
\text{-- where } q.a_j \text{ is the retained attributes of } q \text{ and } r_i \text{ is the parent table} \\
\text{else} \\
\text{-- when } r_i \in R' \\
\text{use the original calculation for } r_i \text{ as the reference check does not change the service demand} \\
\text{end if} \\
\text{end if} \\
\text{end if}
\end{align*}
\]

For the loop structure, since access is to the index, which will be in the buffer for each iteration of the loop, the service demand does not depend on the number of loop iterations.

\[
\begin{align*}
\text{if } s_k = \text{loop}, \text{ and } n \text{ is the number of loop iterations, then} \\
\forall q_m \in \text{loop} \\
\forall r_k \in q_m.\text{Access} \\
\forall f_{k_i} \in r_k \\
\text{if } (q_m.\text{type} \text{ in } (\text{INSERT|UPDATE})) \text{ and } \\
(\exists q_m.a_j = f_{k_i}.\text{ref_attribute}) \text{ then}
\end{align*}
\]
if \((r_i \not\in R)\) then
\[ t_j.tranDBpages[r_i \in R] = f_k^i.FKDBpages[r_i \in R] \]
--where \(q_m.a_j\) is the retained attributes of \(q_m\) and \(r_i\) is the parent table

else
-- when \(r_i \in R\)
use the original calculation for \(r_i\) as the reference check does not change the service demand
end if
end if

The same applies for the branch structure:

\[
\text{if } s_k = \text{branch}, \text{ then} \]
\[
\forall \text{branch}_i \]
\[
\forall q_m \in \text{branch}_i \]
\[
\forall r_k \in q_m.A_\text{ccess} \]
\[
\forall f_k^i \in r_k \]
\[
\text{if } (q_m.type \text{ in } (\text{INSERT|UPDATE})) \text{ and } \exists q_m.a_j = f_k^i.ref\_\text{attribute}) \text{ then} \]
\[
\text{if } (r_i \not\in R) \text{ then} \]
\[
 t_j.tranDBpages[r_i \in R] = f_k^i.FKDBpages[r_i \in R] \]
--where \(q_m.a_j\) is the retained attributes of \(q_m\) and \(r_i\) is the parent table

else
-- when \(r_i \in R\)
use the original calculation for \(r_i\) as the reference check does not change the service demand
end if
end if

end if

\[6.3.3 \text{ Calculating the Routing Path}\]

In order to simplify the routing path algorithm, it is assumed when \(s_k = \text{branch}\), i.e. a branch statement, \(\forall \text{branch}_i \in \text{branch}\) that:

- the last table accessed in \(\text{branch}_i\) cannot invoke a foreign key reference check.
Consequently, if \( branch_i \) accesses only one table, then that table cannot invoke any reference checks.

The algorithm can be easily extended to include these cases. Algorithm 6.1 and 6.2 detail the calculation of the queueing network traversal path for a database design with foreign key constraints. The additions to the original algorithm in Section 3.3 are highlighted.

**Algorithm 6.1: Calculating Customer Class Path**

1: Let \( current_table \) be the current table in the path of \( t_i \)
2: \( current_table \leftarrow 0 \)
3: Let \( branch[n] \) be a vector of \( r_k \in R \), that holds the last table accessed by a branch[\( i \)] of a branch statement, where \( n \) is the number of branches
4: \( branch[] \leftarrow \text{nil} \) (element by element assignment)
5: Let \( bran_table \) be the current table of a branch statement
6: \( bran_table \leftarrow 0 \)
7: Let \( prev_branch[] \) be a vector that holds the initial value of \( branch[] \)
8: \( prev_branch \leftarrow \text{nil} \)
9: Let \( D \) be the set of deferred referential integrity checks
10: \( \forall t_i \in T \)
11: \( \forall s_j \in t_i.S \)
12: ConnectPath(\( s_j \), \( current_table \), \( branch[] \))
13: if \( s_j \) is the last statement in \( t_i \) then
14: if \( D \) is empty then
15: \[ \text{no DEFERRED FK} \]
16: if \( branch[] \neq \text{nil} \) then
17: for \( branch[1] \) to \( \text{|branch[]|} \) do
18: \( P[c, branch[], 0] \leftarrow 1 \)
19: end for
20: else
21: \( P[c, current_table, 0] \leftarrow 1 \)
22: end if
23: else
24: \( \forall r_k.f_k \in D \)
25: ConnectPath(\( f_k.s_j \), \( current_table \), \( branch[] \))
26: \( P[c, current_table, 0] \leftarrow 1 \)
27: end if
28: (the transaction leaves the network after leaving the last table accessed by the last SQL statement of the final statement)
29: end algorithm

139
Algorithm 6.2 : Function ConnectPath

Function ConnectPath($s_j$, $current_table$, $branch[]$)
1: case $s_j = q$
2: $\forall r_k \in q$.Access
3: if ($r_k$ is first table accessed by $q$) and $branch[] \neq$ nil then
   (connect the last tables accessed by the previous branch statement to
   the first table of this SQL statement)

5: for $branch[1]$ to $|branch[]|$ do
6: $P[c, branch[1], r_k] \leftarrow 1$
7: end for
8: $branch[] \leftarrow$ nil
9: $current_table \leftarrow r_k$
10: else
11: $P[c, current_table, r_k] \leftarrow 1$
12: $current_table \leftarrow r_k$
13: end if
14: if ($q$.type in (INSERT| UPDATE) and
    ($r_k$.FK is not NULL)) then
15: $\forall fk_i \in r_k$.FK where
    ($\exists q.a_j = r_k.fk_i$.ref_attribute)
16: if $fk_i$.mode = IMMEDIATE then
17: ConnectPath($fk_i.s_1$, $current_table$, $branch[]$)
18: else --DEFERRED
19: Add $r_k.fk_i$ to $D$
20: end if
21: end if --FK reference
22: end if
23: case $s_j = loop$
24: $\forall q_m \in loop$
25: $\forall r_k \in q_m$.Access
26: if ($q_m$ is first SQL statement) and ($r_k$ is first table accessed by $q_m$)
    and $branch[] \neq$ nil then
    (connect the last tables accessed by the previous branch statement to
    the first table of this SQL statement)
27: for $branch[1]$ to $|branch[]|$ do
28: $P[c, branch[1], r_k] \leftarrow 1$
29: end for
30: $branch[] \leftarrow$ nil
31: $current_table \leftarrow r_k$
32: else
33: $P[c, current_table, r_k] \leftarrow 1$
34: $current_table \leftarrow r_k$
35: end if

$P[c, 0, r_k]$ gives the entry server for $t_i$. Unassigned values take the value zero.
if \( q_m.\text{type} \in \{ \text{INSERT}, \text{UPDATE} \} \) and 
\( r_k.\text{FK} \) is not NULL

\[
\forall \text{fk}_i \in r_k.\text{FK} \text{ where } \\
(\exists \ q_m.a_j = \text{fk}_i.\text{ref_attribute}) \\
\text{if } \text{fk}_i.\text{mode} = \text{IMMEDIATE then} \\
\text{ConnectPath}(\text{fk}_i, s_i, \text{current_table}, \text{branch[i]}) \\
\text{else } \text{DEFERRED} \\
\text{Add } r_k.\text{fk}_i \text{ to } D \\
\text{end if}
\]

end if –FK reference

\[
\text{case } s_i = \text{branch} \\
\text{prev_branch[i] } \leftarrow \text{branch[i]} \\
\text{for } i \text{ in } n \text{ do (total number of branches)} \\
\text{bran_table } \leftarrow \text{current_table} \\
\forall q_m \in \text{branch}_i, \\
\forall r_k \in q_m.\text{Access} \\
\text{if } (q_m \text{ is first SQL statement}) \text{ and } \\
(\text{r}_k \text{ is first table accessed by } q_m) \text{ then} \\
\text{Let } p_i \text{ be the probability of accessing } \text{branch}_i \\
\text{if prev_branch[i] } \neq \text{nil then} \\
\text{(connect the last tables accessed by the previous branch statement to } \\
\text{the first table of this branch’s SQL statement)} \\
\text{for prev_branch[i] to } \mid \text{prev_branch[i]} \text{ do} \\
\text{P}[c_i, \text{prev_branch[i]}, r_k] \leftarrow p_i \\
\text{end for} \\
\text{bran_table } \leftarrow r_k \\
\text{else} \\
\text{P}[c_i, \text{bran_table}, r_k] \leftarrow p_i \\
\text{bran_table } \leftarrow r_k \\
\text{end if} \\
\text{else} \\
\text{P}[c_i, \text{bran_table}, r_k] \leftarrow 1 \\
\text{bran_table } \leftarrow r_k \\
\text{end if} \\
\text{end if} \text{–FK reference}
\]

branch[i] \leftarrow r_k

end for
6.4 TPCC-UVA Foreign Key Performance Modelling

Due to the constraints of the experimental setup, in this Section, we model a simple referential integrity check. In order not to affect the data in the other tables, we modified the design of the TPCC-UVA system by adding a new table, ITEMCopy, which is an exact copy of the original ITEM table including index and keys. We defined a foreign key constraint on the ORDER_LINE table that references the primary key on the ITEMCopy table. The referential integrity check is DEFERRABLE, as that is the least process intensive in PostgreSQL [102].

Figure 6.3 shows the details of the foreign key constraint definition. The resulting referential integrity checks will affect the New-Order transaction only: it is the only transaction that INSERTs items into the ORDER_LINE table. The other transactions SELECT from the ORDER_LINE table (Order-Status and Stock) or DELETE from it (Delivery).

```sql
alter table ORDER LINE add CONSTRAINT fk1 FOREIGN KEY
(ol_i_id) REFERENCES ITEMCopy (i_id) DEFERRABLE;
```

**Figure 6.3** Details of the foreign key constraint on the ORDER-LINE table.

Table 6.1 shows the number of DB pages for the queueing network model for the TPCC-UVA database design. The values in Table 6.1 are calculated in the same manner
as those in Section 4.4.2, the difference is in the DB pages used for the foreign key referential check on the ITEMCopy table. The calculation methods for the service demands are similar to those presented in Appendix C. It would be expected that the referential integrity check would only read the index to check the inserted field; however, we have noticed from the collected statistics that PostgreSQL does a complete row fetch for each referential integrity check. Therefore, the number of pages needed for the referential integrity check is calculated in the same way as that of a SELECT statement.

Using the algorithm in the previous section, the corresponding queueing network model for the TPCC-UVA design with a referential integrity check to the ITEMCopy table is in Figure 6.4. Given that the referential integrity check is DEFERRED, the ITEMCopy table will be the last table accessed by the New-Order transaction.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>number of I/O DB pages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I</td>
</tr>
<tr>
<td>New-Order</td>
<td>0.75</td>
</tr>
<tr>
<td>Payment</td>
<td>2.75</td>
</tr>
<tr>
<td>Order-Status</td>
<td>-</td>
</tr>
<tr>
<td>Delivery</td>
<td>-</td>
</tr>
<tr>
<td>Stock-Level</td>
<td>-</td>
</tr>
</tbody>
</table>

I = WAREHOUSE, II= DISTRICT, III= CUSTOMER, IV= HISTORY, V= ORDER, VI= NEW-ORDER, VII= ORDER-LINE, VIII= STOCK, IX= ITEM, X= ITEMCopy
6.4.1 Experimental Results

The TPCC-UVA system was configured to run with 100 warehouses, each with 2 districts, i.e. 100x2 clients with the addition of the foreign key references mentioned previously. The measurement interval was 120 minutes, as specified by the TPC-C benchmark in which the system is in the steady state. The steady state for the TPCC-UVA design was determined by running the system with a ramp-up period of 20 minutes and a measurement interval of 8 hours. The mean response time per minute was plotted for the New-Order transaction, as detailed in Figure 6.5.
To reach the steady state for the TPCC-UVA design, a ramp-up period of 140 minutes was used. The database was initialized with data for 100x10 clients, as stated in Section 4.4.2. To measure the mean DB page access time, the TPCC-UVA was run 3 different times (using the strace utility). The mean DB page access time of all 3 runs was used to parameterize the queueing network model for the design.

To measure the TPCC-UVA transaction performance metrics the system was run another 3 times, to collect response times for the transactions. The response times were averaged and compared to the simulation results. The 95% confidence intervals were obtained for the system and simulation results, but these were too tight to show on the graphs.
Figures 6.6(a) to 6.6(e) detail the measured and modelled mean response times per minute during the measurement interval for the five transactions for the TPCC-UVA design. Table 6.2 shows the measured and modelled mean response time per transaction calculated during the measurement interval. We would expect the model to give an accurate prediction of the mean response time per minute for the design. However, as can be seen from Figures 6.6(a) to 6.6(e) and Table 6.2 the model has an approximately 36% prediction error.

To investigate the reason for this, we modified our original TPCC-UVA design by adding a foreign key reference on the ORDER-LINE table to the ITEM table. In this design, referential integrity checking for the New-Order transaction would not incur an I/O disk access; since the referenced rows of the ITEM table would already be in the DB buffer due to the execution of a SELECT statement prior to the INSERT statement in the New-Order transaction (see Appendix C). To measure the transaction response times the system was run 3 times, and the transaction response times were averaged.

As can be seen from Table 6.3 the measured mean transaction response times for the design referencing the ITEM table are similar to the mean transaction response times for the design referencing the ITEMCopy table. In addition, in Table 6.3, the transaction mean response times for the original TPCC-UVA design without referential integrity (from Table 4.7) are shown. The new design differs from the original TPCC-UVA design in the addition of the referential integrity check on the ITEM table. Therefore, the increased transaction response time for the new design is due to the processing of the referential integrity checks. Given that the model does not consider processing time, the error rate is justified.
Figure 6.6 Comparison of the (a) New-Order (b) Payment (c) Order-Status (d) Delivery (e) Stock-Level transactions mean response time per minute for a measurement interval of 120 minutes for 100x2 clients.
This processing overhead is due to the effect of referential integrity checks on INSERT statements which cause processing similar to a JOIN on the foreign key [95]. In addition, it has been reported that PostgreSQL has known performance problems when issuing INSERTs to tables with foreign key references [99].

Table 6.2 Comparison of transaction mean response times for the TPCC-UVA design with foreign key referencing.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Response Time (sec)</th>
<th>Response Time (sec)</th>
<th>% Error per trans</th>
<th>% Overall Error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Measured</td>
<td>Modeled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>New-Order</td>
<td>25.47</td>
<td>16.35</td>
<td>35.8</td>
<td>35.84</td>
</tr>
<tr>
<td>Payment</td>
<td>25.31</td>
<td>16.37</td>
<td>35.34</td>
<td></td>
</tr>
<tr>
<td>Order-Status</td>
<td>25.47</td>
<td>16.43</td>
<td>35.48</td>
<td></td>
</tr>
<tr>
<td>Delivery</td>
<td>25.47</td>
<td>16.35</td>
<td>35.81</td>
<td></td>
</tr>
<tr>
<td>Stock-Level</td>
<td>26</td>
<td>16.44</td>
<td>36.77</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.3 Transaction mean response times for the TPCC-UVA design with foreign key referencing the ITEM table and TPCC-UVA original design.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>TPCC-UVA design with referential integrity</th>
<th>Original TPCC-UVA design</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Measured Response Time (sec)</td>
<td>Measured Response Time (sec)</td>
</tr>
<tr>
<td>New-Order</td>
<td>24.3</td>
<td>19.10</td>
</tr>
<tr>
<td>Payment</td>
<td>24.17</td>
<td>19.00</td>
</tr>
<tr>
<td>Order-Status</td>
<td>24.38</td>
<td>19.11</td>
</tr>
<tr>
<td>Delivery</td>
<td>24.59</td>
<td>19.15</td>
</tr>
<tr>
<td>Stock-Level</td>
<td>25.2</td>
<td>19.71</td>
</tr>
</tbody>
</table>

6.5 Summary

In this Chapter, an extension of the database design queueing network performance evaluation model for referential integrity was presented. The formal specification for database foreign keys was given. In addition, a calculation of the service demands for transactions that invoke foreign key reference checks was also presented. A modified
algorithm to calculate a transaction’s path through the queueing network with referential integrity checking was given.

The experimental results have shown that the performance model is able to give an accurate estimation of the mean response time for database designs with referential integrity checks only if the DBMS is efficient in handling foreign key referential integrity processing.
Chapter 7 Conclusions and Future Work

The main contribution of this thesis is the development of a novel performance evaluation method for database designs based on queueing networks. We have provided a formalism that captures the essential database design features while keeping the performance model sufficiently simple to be accessible to database designers who are unlikely to be specialists in queueing theory. This contribution is significant in that the majority of performance evaluation models for database systems target capacity planning or overall system properties, with limited work in detailed database transaction processing and behaviour.

In this Chapter, a summary of the main contributions of the thesis is provided along with directions for future work.

7.1 Main Contributions

This thesis contributes a novel performance evaluation method using queueing networks for database design performance evaluation. This work is considered to be an improvement over previous methodologies in that the transaction is modelled at a finer granularity, thus providing for feedback at an early stage in the design process that is more relevant and useful to the database designer. Moreover, detailed knowledge and modelling of the hardware architecture is not required. In addition, the method provides
for the explicit representation of active database rules and referential integrity in the queueing network models.

We have introduced the database design queueing network performance evaluation model with a formal specification describing the transformation between database designs and queueing network models. The accuracy of this model has been validated by modelling the TPCC-UVA open source implementation of the TPC-C benchmark. Through experimentation with different database designs, results have shown that the database design queueing network model is applicable to designs of large databases where random disk I/O is the dominant cost factor and in which processing costs are negligible.

The simplicity of the modelling algorithms permits the direct mapping between database design entities and queueing networks. Thus, its application is straightforward for database designers. This allows for easy integration of our modelling technique into early database system development phases. The model is useful in providing what if comparisons of database designs before database system implementation. Furthermore, the method is suitable for post-deployment database system performance tuning, and in such a case, the parameterization of the queueing model can be extracted from traces of the database system or from DBMS statistics.

The queueing network models presented in this thesis were for centralized databases. The modelling technique can be applied to distributed databases, in which each distributed node can be modelled as a database design queueing model. For multi-tier applications, the database tier can be represented as a database design queueing model.
Another contribution of the thesis is a classification of the modelling of transactions in database and DBMS queueing network models. This classification is based on the level of detail of the representation of the database transaction’s internal design in the queueing network models. We have identified four main categories: the black box, the transaction processing, the transaction size and the transaction phase models. We have shown that the majority of queueing network models for databases and DBMS components fall into the transaction processing category. While the transaction size and phase category is predominated by studies of DBMS concurrency control mechanisms.

From this categorization, we have identified that the main assumption for transaction service demand is that of exponentially distributed service times. However, justification for this assumption in the context of database systems and transactions was only provided for models that fall into the black box category. In this thesis, we have contributed a justification for the exponential service time assumption for transactions in queueing network models for the other categories, i.e. when transaction details are modelled.

7.2 Future Work

For future work, the formal specification and its related algorithms can form the basis on which to develop a database design analysis tool for implementing this performance evaluation technique. The algorithms for calculating the service demands and routing paths for transactions would need to be extended to include the cases that were excluded in the thesis. This would lead into an investigation of database designs for distributed,
replicated and multi-tier database applications to research their detailed performance behaviour.

The effect of processing in referential integrity checks and active database rules needs to be addressed by extending the cost model to incorporate processing costs. In addition, the costing method can be extended with commercial DBMS specific constraints, e.g. page and row header sizes, which will allow for more accurate estimations. To provide for more realistic database designs and workloads, locking contention will need to be incorporated in the queueing network model. Moreover, more complex access methods can be integrated into the cost model, e.g. bitmap and R-tree indexes.

Another direction would be investigating the extension of the database design queueing network model beyond relational databases, e.g. document-oriented, object-oriented or XML databases.

Finally, an interesting direction would involve investigating the integration of the database design queueing network model with currently available queueing network models for different hardware architectures.
Appendix A: The TPC-C Transaction

Specification

This Appendix is summarized from [113].

A.1 The New-Order Transaction

<table>
<thead>
<tr>
<th>New Order Transaction</th>
<th>The New-Order business transaction consists of entering a complete order through a single database transaction. It represents a mid-weight, read-write transaction with a high frequency of execution and stringent response time requirements to satisfy on-line users. This transaction is the backbone of the workload. It is designed to place a variable load on the system to reflect on-line database activity as typically found in production environments.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td>For a given warehouse number (W_ID), For any given terminal, the home warehouse number (W_ID) is constant</td>
</tr>
<tr>
<td>district number (D_W_ID, D_ID), randomly selected within [1 .. 10] from the home warehouse (D_W_ID = W_ID)</td>
<td></td>
</tr>
<tr>
<td>customer number (C_W_ID, C_D_ID, C_ID), non-uniform random customer number (C_ID) is selected using the NURand(1023,1,3000) function from the selected district number (C_D_ID = D_ID) and the home warehouse number (C_W_ID = W_ID).</td>
<td></td>
</tr>
<tr>
<td>count of items (ol_cnt, not communicated to the SUT), randomly selected within [5 .. 15] (an average of 10),</td>
<td></td>
</tr>
<tr>
<td>and for a given set of items (OL_I_ID), A fixed 1% of the New-Order transactions are chosen at random to simulate user data entry errors and exercise the performance of rolling back update transactions. This must be implemented by generating a random number rbk within [1 .. 100].</td>
<td></td>
</tr>
<tr>
<td>(OL_I_ID), A non-uniform random item number (OL_I_ID) is selected using the NURand(8191,1,100000) function. If this is the last item on the order and rbk = 1 (see Clause 2.4.1.4), then the item number is set to an unused value causing rollback.</td>
<td></td>
</tr>
</tbody>
</table>
supplying warehouses
(OL_SUPPLY_W_ID),
A supplying warehouse number
(OL_SUPPLY_W_ID) is selected as
the home warehouse 99% of the time
and as a remote warehouse 1% of the
time.
This can be implemented by generating
a random number \( x \) within [1 .. 100]:
- If \( x > 1 \), the item is supplied from the
  home warehouse (OL_SUPPLY_W_ID = W_ID).
- If \( x = 1 \), the item is supplied from a
  remote warehouse
  (OL_SUPPLY_W_ID is randomly
  selected within the range of active
  warehouses other than W_ID).

Comment 1: With an average of 10
items per order, approximately 90% of
all orders can be supplied in full by
stocks from the home warehouse.

Comment 2: If the system is
configured for a single warehouse, then
all items are supplied from that single
home warehouse.

| quantities
(OL_QUANTITY): | is randomly selected within [1 .. 10]. |
| S_remote | Set to 1 if remote order-line |
| \( \alpha_{\text{all\_local}} \) | If the order includes only home order-lines, then \( \alpha_{\text{ALL\_LOCAL}} \) is set to 1, otherwise \( \alpha_{\text{ALL\_LOCAL}} \) is set to 0. |

A.2 The Payment Transaction

<p>| Payment Transaction | The Payment business transaction updates the customer's balance and reflects the payment on the district and warehouse sales statistics. It represents a light-weight, read-write transaction with a high frequency of execution and stringent response time requirements to satisfy on-line users. In addition, this transaction includes non-primary key access to the CUSTOMER table. |</p>
<table>
<thead>
<tr>
<th>Body</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>The home warehouse number (W_ID) For any given terminal, is constant over the whole measurement</td>
</tr>
<tr>
<td>The district number (D_W_ID, D_ID)</td>
<td>is randomly selected within [1 ..10] from the home warehouse (D_W_ID) = W_ID.</td>
</tr>
<tr>
<td>The customer id (C_W_ID, C_D_ID, C_LAST)</td>
<td>The customer is randomly selected 60% of the time by last name This can be implemented by generating a random numbers ( y ) within [1 .. 100]; • If ( y &lt;= 60 ) a customer last name (C_LAST) is</td>
</tr>
<tr>
<td>(C_W_ID, C_D_ID, C_ID),</td>
<td>40% of the time by</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| C_ID  | Generated according to **Clause 4.3.2.3** from a non-uniform random value using the NURand(255,0,999) function. The customer is using his/her last name and is one of the possibly several customers with that last name.  
  - If \( y > 60 \) a non-uniform random customer number (C_ID) is selected using the NURand(1023,1,3000) function. The customer is using his/her customer number. |
|  | Independent of the mode of selection, the customer resident warehouse is the home warehouse 85% of the time and is a randomly selected remote warehouse 15% of the time. This can be implemented by generating a random numbers \( x \) within \([1\ldots100]\);  
  - If \( x \leq 85 \) a customer is selected from the selected district number (C_D_ID = D_ID) and the home warehouse number (C_W_ID = W_ID). The customer is paying through his/her own warehouse.  
  - If \( x > 85 \) a customer is selected from a random district number (C_D_ID is randomly selected within \([1\ldots10]\)), and a random remote warehouse number (C_W_ID is randomly selected within the range of active warehouses (see Clause 4.2.2), and \( C_W_ID \neq W_ID \). The customer is paying through a warehouse and a district other than his/her own. |
| H_AMOUNT | The payment amount is randomly selected within \([1.00\ldots5,000.00]\). |
| H_DATE | The payment date cr_date is generated within the SUT by using the current system date and time. |
A.3 The Order-Status Transaction

<table>
<thead>
<tr>
<th>Order-Status Transaction</th>
<th>The Order-Status business transaction queries the status of a customer’s last order. It represents a mid-weight read-only database transaction with a low frequency of execution and response time requirement to satisfy on-line users. In addition, this table includes non-primary key access to the CUSTOMER table.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Body</strong></td>
<td><strong>Constraints</strong></td>
</tr>
<tr>
<td>Input</td>
<td>Home warehouse number (W_ID)</td>
</tr>
<tr>
<td>The district number (D_ID)</td>
<td>Is randomly selected within [1..10] from the home warehouse.</td>
</tr>
<tr>
<td>The customer id (C_W_ID, C_D_ID, C_LAST)</td>
<td>The customer is randomly selected 60% of the time by last name from the selected district (C_D_ID = D_ID) and the home warehouse number (C_W_ID = W_ID).</td>
</tr>
<tr>
<td>This can be implemented by generating a random number y within [1..100];</td>
<td></td>
</tr>
<tr>
<td>• If y &lt;= 60 a customer last name (C_LAST) is generated according to Clause 4.3.2.3 from a non-uniform random value using the NURand(255,0,999) function. The customer is using his/her last name and is one of the, possibly several, customers with that last name.</td>
<td></td>
</tr>
<tr>
<td>• If y &gt; 60 a non-uniform random customer number (C_ID) is selected using the NURand(1023,1,3000) function. The customer is using his/her customer number.</td>
<td></td>
</tr>
<tr>
<td>(C_W_ID, C_D_ID, C_ID)</td>
<td>And 40% of the time by number from the selected district (C_D_ID = D_ID) and the home warehouse number (C_W_ID = W_ID).</td>
</tr>
</tbody>
</table>
A.4 The Delivery Transaction

The Delivery business transaction consists of processing a batch of 10 new (not yet delivered) orders. Each order is processed (delivered) in full within the scope of a read-write database transaction. The number of orders delivered as a group (or batched) within the same database transaction is implementation specific. The business transaction, comprised of one or more (up to 10) database transactions, has a low frequency of execution and must complete within a relaxed response time requirement. The Delivery transaction is intended to be executed in deferred mode through a queueing mechanism, rather than interactively, with terminal response indicating transaction completion. The result of the deferred execution is recorded into a result file.

<table>
<thead>
<tr>
<th>Delivery Transaction</th>
<th>The delivery date (OL_DELIVERY_D) is generated within the SUT by using the current system date and time.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Body Constraints</td>
<td>Constraints</td>
</tr>
<tr>
<td>Input</td>
<td>Body</td>
</tr>
<tr>
<td></td>
<td>the home warehouse number (W_ID) For any given terminal, is constant over the whole measurement interval.</td>
</tr>
<tr>
<td></td>
<td>the carrier number (O_CARRIER_ID) is randomly selected within [1 .. 10].</td>
</tr>
<tr>
<td></td>
<td>The delivery date (OL_DELIVERY_D) is generated within the SUT by using the current system date and time.</td>
</tr>
</tbody>
</table>

A.5 The Stock-Level Transaction

The Stock-Level business transaction determines the number of recently sold items that have a stock level below a specified threshold. It represents a heavy read-only database transaction with a low frequency of execution, a relaxed response time requirement, and relaxed consistency requirements.

<table>
<thead>
<tr>
<th>Stock-Level Transaction</th>
<th>The district number (D_ID) is selected at random within [10 .. 20].</th>
</tr>
</thead>
<tbody>
<tr>
<td>Body</td>
<td>Constraints</td>
</tr>
<tr>
<td>Input</td>
<td>Body</td>
</tr>
<tr>
<td>the home warehouse number (W_ID)</td>
<td>Each terminal must use a unique value of (W_ID, D_ID) that is constant over the whole measurement, i.e., D_IDs cannot be re-used within a warehouse</td>
</tr>
<tr>
<td>The district number (D_ID)</td>
<td></td>
</tr>
<tr>
<td>the threshold of minimum quantity in stock (threshold)</td>
<td>is selected at random within [10 .. 20].</td>
</tr>
</tbody>
</table>
Appendix B: The TPCC-UVA Table

Specifications

The TPCC-UVA database table specification provided here are taken from the source code. The information regarding the population of the tables is taken from the TPC-C benchmark [113]. This design represents the table used in our performance experiments, e.g. foreign key referencing is not shown.
### TABLE NAME: WAREHOUSE

<table>
<thead>
<tr>
<th>Columns</th>
<th>Distribution</th>
<th>Database Population</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>w_id int4</td>
<td>2*W unique IDs</td>
<td>unique within [W]</td>
<td>wareh1 PRIMARY KEY (w_id)</td>
</tr>
<tr>
<td>w_name varchar(10)</td>
<td>random a-string [6 .. 10]¹</td>
<td></td>
<td>populated sequentially sorted by w_id</td>
</tr>
<tr>
<td>w_street_1 varchar(20)</td>
<td>random a-string [10 .. 20]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>w_street_2 varchar(20)</td>
<td>random a-string [10 .. 20]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>w_city varchar(20)</td>
<td>random a-string [10 .. 20]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>w_state char(2)</td>
<td>random a-string of 2 letters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>w_zip char(9)</td>
<td>generated according to ²</td>
<td></td>
<td></td>
</tr>
<tr>
<td>w_tax float4</td>
<td>random within [0.0000 .. 0.2000]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>w_ytd float8</td>
<td>300,000.00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

¹ The notation **random a-string [x .. y]** (respectively, **n-string [x .. y]**) represents a string of random alphanumeric (respectively, numeric) characters of a random length of minimum x, maximum y, and mean (y+x)/2.

**Comment 1:** The character set used must be able to represent a minimum of 128 different characters.

**Comment 2:** Generating such strings can be implemented by the concatenation of two strings selected at random from two separate arrays of strings, and where:
1. Both arrays contain a minimum of 10 different strings of characters.
2. The first array contains strings of x characters.
3. The second array contains strings of lengths uniformly distributed between zero and (y - x) characters.
4. Both arrays may contain strings that are pertinent to the row and the attribute (e.g., use an actual first name for C_FIRST) instead of strings of random characters, as long as this does not bring any improvement to the reported metrics.

² The warehouse zip code (W_ZIP), the district zip code (D_ZIP) and the customer zip code (C_ZIP) must be generated by the concatenation of:
1. A random n-string of 4 numbers, and
2. The constant string '11111'.

Given a random n-string between 0 and 9999, the zip codes are determined by concatenating the n-string and the constant '11111'. This will create 10,000 unique zip codes. For example, the n-string 0503 concatenated with 11111, will make the zip code 050311111.

**Comment:** With 30,000 customers per warehouse and 10,000 zip codes available, there will be an average of 3 customers per warehouse with the same zip code.
<table>
<thead>
<tr>
<th><strong>DISTRICT</strong></th>
<th><strong>d_id int4</strong></th>
<th>20 unique IDs - 10 are populated per warehouse</th>
<th>unique within [10]</th>
<th>dist1 PRIMARY KEY (d_w_id, d_id), populated sequentially sorted by d_id, w_id</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>d_w_id int4</td>
<td>2*W unique IDs</td>
<td>= W_ID</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d_name varchar(10)</td>
<td>random a-string [6 .. 10]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d_street_1 varchar(20)</td>
<td>random a-string [10 .. 20]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d_street_2 varchar(20)</td>
<td>random a-string [10 .. 20]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d_city varchar(20)</td>
<td>random a-string [10 .. 20]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d_state char(2)</td>
<td>random a-string of 2 letters</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d_zip char(9)</td>
<td>generated according to</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d_tax float4</td>
<td>random within [0.0000 .. 0.2000]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d_ytd float8</td>
<td>30,000.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d_next_o_id int4</td>
<td>10,000,000 unique IDs</td>
<td>3,001</td>
<td></td>
</tr>
<tr>
<td><strong>CUSTOMER</strong></td>
<td>c_id int4</td>
<td>96,000 unique IDs - 3,000 are populated per district</td>
<td>unique within [3,000]</td>
<td>custom1 PRIMARY KEY (c_w_id, c_d_id, c_id), populated sequentially sorted by c_id, c_d_id, c_w_id</td>
</tr>
<tr>
<td></td>
<td>c_d_id int4</td>
<td>20 unique IDs</td>
<td>= D_ID</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c_w_id int4</td>
<td>2*W unique IDs</td>
<td>D_W_ID</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c_first varchar(16)</td>
<td>random a-string [8 .. 16]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c.middle char(2)</td>
<td>‘OE’</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All customers of 1st dist, 1st ware, 2nd dist, 1st ware, ...... 10th dist, 1st ware, ...... 10th dist, nth ware
The customer last name (C_LAST) must be generated by the concatenation of three variable length syllables selected from the following list:

```
BAR        OUGHT       ABLE       PRI        PRES       ESE     ANTI     CALLY    ATION    EING
```

Given a number between 0 and 999, each of the three syllables is determined by the corresponding digit in the three digit representation of the number. For example, the number 371 generates the name PRICALLYOUGHT, and the number 40 generates the name BARPRESBAR.

The term non-uniform random, used only for generating customer numbers, customer last names, and item numbers, means an independently selected and non-uniformly distributed random number over the specified range of values \([x .. y]\). This number must be generated by using the function \text{NURand} which produces positions within the range \([x .. y]\). The results of NURand might have to be converted to produce a name or a number valid for the implementation.

\[
\text{NURand}(A, x, y) = (((\text{random}(0, A) \mid \text{random}(x, y)) + C) \% (y - x + 1)) + x
\]

where:

- \(\text{exp-1} \mid \text{exp-2}\) stands for the bitwise logical OR operation between \text{exp-1} and \text{exp-2}
- \(\text{exp-1} \% \text{exp-2}\) stands for \text{exp-1} modulo \text{exp-2}
- \text{random}(x, y)\) stands for randomly selected within \([x .. y]\)
- \(A\) is a constant chosen according to the size of the range \([x .. y]\)
  - for C_LAST, the range is \([0 .. 999]\) and \(A = 255\)
  - for C_ID, the range is \([1 .. 3000]\) and \(A = 1023\)
  - for OL_I_ID, the range is \([1 .. 100000]\) and \(A = 8191\)
- \(C\) is a run-time constant randomly chosen within \([0 .. A]\) that can be varied without altering performance. The same \(C\) value, per field (C_LAST, C_ID, and OL_I_ID), must be used by all emulated terminals.

In order that the value of \(C\) used for C_LAST does not alter performance the following must be true:

- Let C-Load be the value of \(C\) used to generate C_LAST when populating the database. C-Load is a value in the range of \([0..255]\) including 0 and 255.
- Let C-Run be the value of \(C\) used to generate C_LAST for the measurement run.
- Let C-Delta be the absolute value of the difference between C-Load and C-Run. C-Delta must be a value in the range of \([65..119]\) including the values of 65 and 119 and excluding the value of 96 and 112.
using the function NURand(255,0,999) for each of the remaining 2,000 customers. The run-time constant C used for the database population must be randomly chosen independently from the test run(s).

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_street_1 varchar(20)</td>
<td>random a-string [10 .. 20]</td>
</tr>
<tr>
<td>c_street_2 varchar(20)</td>
<td>random a-string [10 .. 20]</td>
</tr>
<tr>
<td>c_city varchar(20)</td>
<td>random a-string [10 .. 20]</td>
</tr>
<tr>
<td>c_state char(2)</td>
<td>random a-string of 2 letters</td>
</tr>
<tr>
<td>c_zip char(9)</td>
<td>generated according to +</td>
</tr>
<tr>
<td>c_phone char(16)</td>
<td>random n-string of 16 numbers</td>
</tr>
<tr>
<td>c_since timestamp</td>
<td>date/time given by the operating system when the CUSTOMER table was populated.</td>
</tr>
<tr>
<td>c_credit char(2)</td>
<td>'GC' or 'BC'</td>
</tr>
<tr>
<td>c_credit_lim float8</td>
<td>50,000.00</td>
</tr>
<tr>
<td>c_discount float4</td>
<td>random within [0.0000 .. 0.5000]</td>
</tr>
<tr>
<td>c_balance float8</td>
<td>-10.00</td>
</tr>
<tr>
<td>c_ytd_payment float8</td>
<td>10.00</td>
</tr>
<tr>
<td>c_payment_cnt int2</td>
<td>1</td>
</tr>
<tr>
<td>c_delivery_cnt int2</td>
<td>0</td>
</tr>
<tr>
<td>c_data varchar(500)</td>
<td>random a-string [300 .. 500]</td>
</tr>
</tbody>
</table>

**HISTORY**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>h_c_id int4</td>
<td>96,000 unique IDs = C_ID</td>
</tr>
<tr>
<td>h_c_d_id int4</td>
<td>20 unique IDs = H_D_ID = D_ID</td>
</tr>
<tr>
<td>h_c_w_id number</td>
<td>2*W unique IDs = H_W_ID = W_ID</td>
</tr>
<tr>
<td>h_d_id int4</td>
<td>20 unique IDs = H_D_ID = D_ID</td>
</tr>
<tr>
<td>h_w_id int4</td>
<td>2*W unique IDs = H_W_ID = W_ID</td>
</tr>
<tr>
<td>h_date timestamp</td>
<td>current date and time</td>
</tr>
<tr>
<td>h_amount float4</td>
<td>10.00</td>
</tr>
<tr>
<td>h_data varchar(24)</td>
<td>random a-string [12 .. 24]</td>
</tr>
</tbody>
</table>

populated sequentially sorted by h_c_id, h_c_d_id, h_c_w_id: All customers of 1st dist, 1st ware, 2nd dist, 1st ware, 10th dist, 1st ware, 10th dist, nth ware
<table>
<thead>
<tr>
<th>ORDER</th>
<th></th>
<th>10,000,000 unique IDs</th>
<th>unique within [3,000]</th>
</tr>
</thead>
<tbody>
<tr>
<td>o_id int4</td>
<td>int4</td>
<td>10,000,000 unique IDs</td>
<td>unique within [3,000]</td>
</tr>
<tr>
<td>o_w_id int4</td>
<td>int4</td>
<td>2*W unique IDs</td>
<td>= W_ID</td>
</tr>
<tr>
<td>o_d_id int4</td>
<td>int4</td>
<td>20 unique IDs</td>
<td>= D_ID</td>
</tr>
<tr>
<td>o_c_id int4</td>
<td>int4</td>
<td>96,000 unique IDs</td>
<td>selected sequentially from a random permutation of [1 .. 3,000]</td>
</tr>
<tr>
<td>o_entry_d timestamp</td>
<td>TIMESTAMP</td>
<td>DEFAULT '1970-01-01'</td>
<td>current date/time given by the operating system</td>
</tr>
<tr>
<td>o_carrier_id int2</td>
<td>int2</td>
<td>10 unique IDs or null</td>
<td>random within [1 .. 10] if O_ID &lt; 2,101, null otherwise</td>
</tr>
<tr>
<td>o_all_local int2</td>
<td>int2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

ORDER PRIMARY KEY (o_w_id, o_d_id, o_id) populated sequentially sorted by o_id, o_d_id, o_w_id:

All orders of 1st dist, 1st ware, 2nd dist, 1st ware, ….. 10th dist, 1st ware, ….. 10th dist, nth ware

<table>
<thead>
<tr>
<th>NEW ORDER</th>
<th></th>
<th>10,000,000 unique IDs</th>
<th>= O_ID, with O_ID between 2,101 and 3,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>no_o_id int4</td>
<td>int4</td>
<td>10,000,000 unique IDs</td>
<td>= O_ID, with O_ID between 2,101 and 3,000</td>
</tr>
<tr>
<td>no_d_id int4</td>
<td>int4</td>
<td>2*W unique IDs</td>
<td>= W_ID</td>
</tr>
<tr>
<td>no_w_id int4</td>
<td>int4</td>
<td>20 unique IDs</td>
<td>= D_ID</td>
</tr>
</tbody>
</table>

NEW ORDER PRIMARY KEY (no_w_id, no_d_id, no_o_id), populated sequentially sorted by no_o_id, no_d_id, no_w_id:

All new-orders of 1st dist, 1st ware, 2nd dist, 1st ware, ….. 10th dist, 1st ware, ..10th dist, nth ware
<table>
<thead>
<tr>
<th>ORDER-LINE</th>
<th>Type</th>
<th>Description</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>ol_o_id int4</td>
<td>10,000,000 unique IDs</td>
<td>= O_ID</td>
<td>principal key (ol_w_id, ol_d_id, ol_o_id, ol_number), populated sequentially sorted by</td>
</tr>
<tr>
<td><strong>STOCK</strong></td>
<td>s_i_id int4</td>
<td>200,000 unique IDs - 100,000 populated per warehouse</td>
<td>unique within [100,000]</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>--------------------------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>s_w_id int4</td>
<td>2^W unique IDs = W_ID</td>
<td>random within [10 .. 100]</td>
<td>s_quantity int2</td>
</tr>
<tr>
<td>s_dist_01 char(24)</td>
<td>random a-string of 24 letters</td>
<td>s_dist_02 char(24)</td>
<td>random a-string of 24 letters</td>
</tr>
<tr>
<td>s_dist_03 char(24)</td>
<td>random a-string of 24 letters</td>
<td>s_dist_04 char(24)</td>
<td>random a-string of 24 letters</td>
</tr>
<tr>
<td>s_dist_05 char(24)</td>
<td>random a-string of 24 letters</td>
<td>s_dist_06 char(24)</td>
<td>random a-string of 24 letters</td>
</tr>
<tr>
<td>s_dist_07 char(24)</td>
<td>random a-string of 24 letters</td>
<td>s_dist_08 char(24)</td>
<td>random a-string of 24 letters</td>
</tr>
<tr>
<td>s_dist_09 char(24)</td>
<td>random a-string of 24 letters</td>
<td>s_dist_10 char(24)</td>
<td>random a-string of 24 letters</td>
</tr>
<tr>
<td>s_ytd numeric(8,2)</td>
<td>0</td>
<td>s_order_cnt int2</td>
<td>0</td>
</tr>
<tr>
<td>s_remote_cnt int2</td>
<td>0</td>
<td>s_data varchar(50)</td>
<td>random a-string [26 .. 50]. For 10% of the rows, selected at random, the string &quot;ORIGINAL&quot; must be held by 8 consecutive characters starting at a random position within S_DATA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>ITEM</strong></th>
<th>i_id int4</th>
<th>200,000 unique IDs - 100,000 items are populated</th>
<th>unique within [100,000]</th>
<th>item1 PRIMARY KEY (i_id) populated sequentially sorted by i_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_im_id int4</td>
<td>200,000 unique IDs</td>
<td>random within [1 .. 10,000]</td>
<td>i_name varchar(24)</td>
<td>random a-string' [14 .. 24]</td>
</tr>
<tr>
<td>i_price float8</td>
<td>random within [1.00 .. 100.00]</td>
<td>i_data varchar(50)</td>
<td>random a-string' [26 .. 50]. For 10% of the rows, selected at random, the string &quot;ORIGINAL&quot; must be held by 8 consecutive characters starting at a random position within I_DATA</td>
<td></td>
</tr>
</tbody>
</table>
Appendix C: The TPCC-UVA

Transaction SQL Source Code and Service Demand Calculation

In this Appendix, the SQL source code of the TPCC-UVA system is detailed. Due to space considerations, we have removed all other code in the transactions. The calculation of the service demands for the SQL statements is based on Section 4.4.2 and the TPCC-UVA table designs in Appendix B.

C.1 Calculation of TPCC-UVA Index I/O Cost

To calculate tree index fan-out, we assume index pages are fully loaded and ignoring header size. The index fan-out is:

\[
\left\lfloor \frac{\text{PageSize}}{\text{IndexEntrySize}} \right\rfloor
\]

where_PAGESize is the DB page size and _IndexEntrySize is the size of the index key + index pointer. The PostgreSQL index pointer size is 6 bytes long [31]. PostgreSQL page size is 8192 bytes [102]. Table C.1 shows the fan-out values for the indexes of the TPCC-UVA database design.
Table C.2 shows a partial calculation of the I/O cost for the TPCC-UVA database design based on the cost model in Section. These values are used in the following sections.
Table C.1 Calculation of the TPCC-UVA index fan-out.

<table>
<thead>
<tr>
<th>key</th>
<th>WAREHOUSE</th>
<th>DISTRICT</th>
<th>CUSTOMER</th>
<th>HISTORY</th>
<th>ORDER</th>
<th>NEW-ORDER</th>
<th>ORDER-LINE</th>
<th>STOCK</th>
<th>ITEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>w_id</td>
<td>1</td>
<td>10</td>
<td>30,000</td>
<td>30,000</td>
<td>30,000</td>
<td>9,000</td>
<td>300,000</td>
<td>100,000</td>
<td>100,000</td>
</tr>
<tr>
<td>d_w_id,d_id</td>
<td>100</td>
<td>1,000</td>
<td>3,000,000</td>
<td>3,000,000</td>
<td>3,000,000</td>
<td>900,000</td>
<td>30,000,000</td>
<td>10,000,000</td>
<td>100,000</td>
</tr>
<tr>
<td>c_w_id, c_d_id, c_id</td>
<td>89</td>
<td>95</td>
<td>655</td>
<td>46</td>
<td>24</td>
<td>8</td>
<td>54</td>
<td>306</td>
<td>82</td>
</tr>
<tr>
<td>o_w_id, o_d_id, o_id</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no_w_id, no_d_id, no_o_id</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ol_w_id, ol_d_id, ol_o_id, ol_number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s_w_id, s_i_id</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i_id</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>key size in bytes</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td>10</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>index entry size in bytes</td>
<td>8</td>
<td>10</td>
<td>14</td>
<td>0</td>
<td>14</td>
<td>8</td>
<td>16</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>fan-out (F)</td>
<td>1024</td>
<td>820</td>
<td>586</td>
<td>0</td>
<td>586</td>
<td>1024</td>
<td>512</td>
<td>683</td>
<td>820</td>
</tr>
</tbody>
</table>
Table C.2 Partial calculation of the TPCC-UVA index I/O cost.

<table>
<thead>
<tr>
<th>WAREHOUSE</th>
<th>DISTRICT</th>
<th>CUSTOMER</th>
<th>HISTORY</th>
<th>ORDER</th>
<th>NEW-ORDER</th>
<th>ORDER-LINE</th>
<th>STOCK</th>
<th>ITEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>30,000</td>
<td>30,000</td>
<td>30,000</td>
<td>9,000</td>
<td>300,000</td>
<td>100,000</td>
<td>100,000</td>
</tr>
<tr>
<td># of rows</td>
<td>100</td>
<td>1,000</td>
<td>3,000,000</td>
<td>3,000,000</td>
<td>900,000</td>
<td>30,000,000</td>
<td>10,000,000</td>
<td>100,000</td>
</tr>
<tr>
<td>row length in bytes</td>
<td>89</td>
<td>95</td>
<td>655</td>
<td>46</td>
<td>24</td>
<td>8</td>
<td>54</td>
<td>82</td>
</tr>
<tr>
<td>table size in bytes</td>
<td>8,900</td>
<td>95,000</td>
<td>1,965,000,000</td>
<td>138,000,000</td>
<td>72,000,000</td>
<td>7,200,000</td>
<td>1,620,000,000</td>
<td>3,060,000,000</td>
</tr>
<tr>
<td>rows per page (PostgreSQL page size / row length)</td>
<td>93</td>
<td>87</td>
<td>13</td>
<td>179</td>
<td>342</td>
<td>1,024</td>
<td>152</td>
<td>27</td>
</tr>
<tr>
<td>total # of pages (B)</td>
<td>2</td>
<td>12</td>
<td>239,869</td>
<td>16,846</td>
<td>8,790</td>
<td>879</td>
<td>197,754</td>
<td>373,536</td>
</tr>
<tr>
<td>index fan-out (F)</td>
<td>1024</td>
<td>820</td>
<td>586</td>
<td>0</td>
<td>586</td>
<td>1024</td>
<td>512</td>
<td>683</td>
</tr>
<tr>
<td>log₃B</td>
<td>0.1</td>
<td>0.37</td>
<td>1.94</td>
<td>0</td>
<td>1.42</td>
<td>0.98</td>
<td>1.95</td>
<td>1.97</td>
</tr>
<tr>
<td>(index:row) ratio (R)</td>
<td>0.09</td>
<td>0.11</td>
<td>0.02</td>
<td>0</td>
<td>0.58</td>
<td>1</td>
<td>0.3</td>
<td>0.04</td>
</tr>
<tr>
<td>log₈(index:row)*B</td>
<td>-0.25</td>
<td>0.04</td>
<td>1.33</td>
<td>0</td>
<td>1.34</td>
<td>0.98</td>
<td>1.76</td>
<td>1.47</td>
</tr>
</tbody>
</table>
### C.2 The New-Order Transaction

<table>
<thead>
<tr>
<th>Source SQL Statements</th>
<th>Formulas for Service Demands</th>
</tr>
</thead>
</table>
| **SELECT** w_tax, c_discount, c_last, c_credit INTO :w_tax, :c_discount, :c_last, :c_credit **FROM warehouse, customer** WHERE w_id=:w_id AND c_w_id=:w_id AND c_d_id=:d_id AND c_id=:c_id; | The warehouse table is the smaller of the two tables, therefore the query optimizer will choose it first. **WAREHOUSE**: b-tree unclustered tree index: equality on index key cost = \( D(1 + \log_{RB}) = D(1 + (-0.25)) = 0.75D \)  
**CUSTOMER**: b-tree unclustered tree index: equality on index key cost = \( D(1 + \log_{RB}) = D(1 + 1.33) = 2.33D \) |
| **SELECT** d_next_o_id, d_tax INTO :d_next_o_id, :d_tax **FROM district** WHERE d_id=:d_id AND d_w_id=:w_id; | b-tree unclustered tree index: equality on index key cost = \( D(1 + \log_{RB}) = D(1 + (0.04)) = 1.04D \) |
| **UPDATE district** SET d_next_o_id=:d_next_o_id+1 **WHERE d_id=:d_id AND d_w_id=:w_id;** | b-tree unclustered tree index: equality on index key search plus Update: cost = Search + 2D  
the initial DB pages are in the buffer,  
\( \text{cost} = 0 + 2D = 2D \) |
| **INSERT** INTO new_order (no_o_id, no_d_id, no_w_id) VALUES (:o_id, :d_id, :w_id); | Even though all the information in the rows are available in the index, PostgreSQL does a table look-up [102], therefore: Insert unclustered tree index cost = \( D(3 + \log_{RB}) = D(3 + 0.98) = 3.98D \) |
| while((i<15) && (new_order->item[i].flag==1)) here we assume an average of 10 items to an order |  |
| **SELECT** i_price, i_name, i_data INTO :i_price, :i_name, :i_data **FROM item** WHERE i_id=:ol_i_id; | b-tree unclustered tree index: equality on index key cost = \( 10xD(1 + \log_{RB}) = 10xD(1+0.71) = 17.1D \) |
| **SELECT** s_quantity, s_data, s_dist_01, s_dist_02, s_dist_03, s_dist_04, s_dist_05, s_dist_06, s_dist_07, s_dist_08, s_dist_09, s_dist_10 INTO :s_quantity, :s_data, :s_dist_01, :s_dist_02, :s_dist_03, :s_dist_04, | b-tree unclustered tree index: equality on index key cost = \( 10xD(1 + \log_{RB}) = 10xD(1+1.47) = 24.7D \) |
```sql
:s_dist_05, :s_dist_06,
:s_dist_07, :s_dist_08,
:s_dist_09, :s_dist_10
FROM stock
WHERE s_i_id = :ol_i_id
AND s_w_id = :ol_supply_w_id;

if s_quantity>=ol_quantity+10) {
    s_quantity=s_quantity-
    ol_quantity;
    EXEC SQL
    UPDATE stock SET
s_quantity=s_quantity
WHERE s_i_id = :ol_i_id AND
s_w_id = :ol_supply_w_id;
}
else{
    s_quantity=(s_quantity-
    ol_quantity)+91;
    EXEC SQL
    UPDATE stock SET
s_quantity=s_quantity
WHERE s_i_id = :ol_i_id AND
s_w_id = :ol_supply_w_id;
} /*end if*/

UPDATE stock SET
s_ytd=s_ytd+cast(:ol_quantity
as real),
s_order_cnt=s_order_cnt+1
WHERE s_i_id = :ol_i_id AND
s_w_id = :ol_supply_w_id;

if(ol_supply_w_id!=w_id) {
EXEC SQL
UPDATE stock SET
s_remote_cnt=s_remote_cnt+1
WHERE s_i_id = :ol_i_id AND
s_w_id = :ol_supply_w_id;

    o_all_local=0;
} /*end if*/

EXEC SQL
INSERT INTO order_line
(ol_o_id, ol_d_id, ol_w_id, ol_number, ol_i_id, ol_supply_w_id, ol_quantity, ol_amount, ol_dist_info) VALUES (:o_id, :d_id, :w_id, :ol_number, :ol_i_id, :ol_supply_w_id,
```

The DB pages for the rows affected by these SQL statements will be in the buffer from the previous statement. Therefore, all the UPDATEs will be on the buffered pages and will only be written back once at the end of the transaction.

\[
\text{cost} = 10 \times \text{cost of writing an update} = 10 \times (2D) = 20D
\]

Insert unclustered tree index
\[
\text{cost} = 10 \times D(3 + \log_{RB} F) = 10 \times D(3 + 1.76) = 47.6D
\]

172
:ol_quantity, :ol_amount,
:ol_dist_info);
i++; /*increments the
number of items*/
} /*end while*/

**INSERT** INTO **orderr** (o_id,
o_d_id, o_w_id, o_c_id,
o_entry_d, o_carrier_id,
o_all_local)
VALUES (:o_id, :d_id, :w_id,
:c_id, :o_entry_d, 0,
:o_all_local);

**if** (o_all_local==0){
**EXEC SQL UPDATE orderr** SET
o_all_local=:o_all_local
WHERE o_id=:o_id AND
o_d_id=:d_id AND o_w_id=:w_id;
} /*end if*/

The previous **INSERT** statement brings the DB page into
the buffer. It will be UPDATED and written only ONE
time to disk. This was already accounted for by the
**INSERT** cost.

### C.3 The Payment Transaction

<table>
<thead>
<tr>
<th>Source SQL Statements</th>
<th>Formulas for Service Demands</th>
</tr>
</thead>
</table>
| **EXEC SQL**
**SELECT** w_name, w_street_1,
w_street_2, w_city, w_state,
w_zip
**INTO** :w_name, :w_street_1,
:w_street_2, :w_city, :w_state,
:w_zip
**FROM** warehouse
WHERE w_id = :w_id;
| b-tree unclustered tree index: equality on index key
cost = D(1+ log\_RB) = D(1+ (-0.25)) = 0.75D |
| **EXEC SQL**
**UPDATE** warehouse
**SET** w_ytd = w_ytd + :h_amount
**WHERE** w_id = :w_id;
| b-tree unclustered tree index: equality on index key
search plus Update:
cost = Search + 2D
the initial DB pages are in the buffer,
cost = 0 + 2D = 2D |
| **EXEC SQL**
**SELECT** d_name, d_street_1,
d_street_2, d_city, d_state,
d_zip
**INTO** :d_name, :d_street_1,
:d_street_2, :d_city, :d_state,
:d_zip
**FROM** district
WHERE d_w_id = :w_id AND
d_id = :d_id;
| b-tree unclustered tree index: equality on index key
cost = D(1+ log\_RB) = D(1+ 0.04) = 1.04D |
| **EXEC SQL**
**UPDATE** district
| b-tree unclustered tree index: equality on index key
search plus Update: |
\[
\text{cost} = \text{Search} + 2D
\]

the initial DB pages are in the buffer,
\[
\text{cost} = 0 + 2D = 2D
\]

\[
\text{if} \ (c\_id == 0) \ {/} */\text{Customer selects BY C\_LAST} */
\]

\[
\text{EXEC SQL}
\]
\[
\text{SELECT} \ \text{count(c\_id)}
\]
\[
\text{INTO} :\text{cont}
\]
\[
\text{FROM} \ \text{customer}
\]
\[
\text{WHERE} c\_last = :c\_last \ \text{AND}
\]
\[
c\_d\_id = :c\_d\_id \ \text{AND} \ c\_w\_id = :c\_w\_id;
\]

\[
\text{EXEC SQL}
\]
\[
\text{DECLARE} \ c\_porlast \text{ CURSOR}
\]
\[
\text{FOR}
\]
\[
\text{SELECT} \ c\_id, c\_first,
\]
\[
c\_middle, c\_street\_l,
\]
\[
c\_street\_2, c\_city, c\_state,
\]
\[
c\_zip, c\_phone, c\_credit,
\]
\[
c\_credit\_lim, c\_discount, c\_balance, c\_since
\]
\[
\text{FROM} \ \text{customer}
\]
\[
\text{WHERE} c\_w\_id = :c\_w\_id \ \text{AND}
\]
\[
c\_d\_id = :c\_d\_id \ \text{AND} \ c\_last = :c\_last
\]
\[
\text{ORDER} \ \text{BY} c\_first;
\]
\[
\text{EXEC SQL OPEN} \ c\_porlast; /*\text{It initializes the cursor} */
\]

\[
\text{for} \ (i = 0; i < \text{cont}/2; i++)\{
\]
\[
\text{EXEC SQL FETCH FROM} \ c\_porlast
\]
\[
\text{INTO} :c\_id, :c\_first,
\]
\[
:c\_middle, :c\_street\_l,
\]
\[
:c\_street\_2, :c\_city, :c\_state,
\]
\[
:c\_zip, :c\_phone, :c\_credit,
\]
\[
:c\_credit\_lim, :c\_discount, :c\_balance,
\]
\[
:c\_since;
\]
\[
\text{EXEC SQL CLOSE} \ c\_porlast;
\]
\[
\} \ \text{else} \ {/} */\text{Customer selects BY C\_ID} */
\]

\[
\text{EXEC SQL}
\]
\[
\text{SELECT} \ c\_first, c\_middle,
\]
\[
:c\_last, c\_street\_l,
\]
\[
c\_street\_2, c\_city, c\_state,
\]
\[
:c\_zip, c\_phone, c\_credit,
\]
\[
c\_discount, c\_balance, c\_since
\]
\[
\text{INTO} :c\_first, :c\_middle,
\]
\[
:c\_last, :c\_street\_l,
\]

By customer last name
b-tree unclustered index, partial match range search
Given that the file is sorted on the key, it will be a range search, however the DB pages depend on the qualifying number of pages and not the number of records.

\[
\text{Cost} = D(\ \log_{\text{RB}} \ # \ of \ matching \ pages)
\]
\[
= D(1.33 + 250) = 251.33D
\]

b-tree unclustered tree index: equality on index key search plus Update:
\[
\text{cost} = \text{Search} + 2D
\]
the initial DB pages are in the buffer,
\[
\text{cost} = 0 + 2D = 2D
\]

\[
\text{total} = 253.33D
\]

By customer id
b-tree unclustered tree index: equality on index key
\[
\text{cost} = D(1 + \log_{\text{RB}}) = D(1 + 1.33) = 2.33D
\]

Average: 60\% by customer last name, 40\% by customer id
\[
\text{total} = (0.6)(253.33) + (0.4)(2.33)D = 152.93D
\]

The rest of the SQL statements, the rows affected by these SQL statements will be in the buffer from the previous statements.
:c_street_2, :c_city,
:c_state, :c_zip, :c_phone,
:c_credit, :c_discount,
:c_balance, :c_since
FROM customer
WHERE c_w_id = :w_id AND
  c_d_id = :d_id AND
  c_id = :c_id;
} /* if (c_id == 0) */

EXEC SQL
UPDATE customer
SET c_balance = c_balance -
  :h_amount, c_ytd_payment =
  c_ytd_payment + :h_amount,
  c_payment_cnt = c_payment_cnt +
1
WHERE c_w_id = :c_w_id AND
  c_d_id = :c_d_id AND
  c_id = :c_id;
if (c_credit[0]=='B'){

EXEC SQL
SELECT c_data
INTO :c_data
FROM customer
WHERE c_id = :c_id AND
  c_w_id = :c_w_id
AND c_d_id = :c_d_id;

EXEC SQL
UPDATE customer SET
  c_data = :c_new_data
WHERE c_w_id = :c_w_id AND
  c_d_id = :c_d_id AND
  c_id = :c_id;
} /* end if */

EXEC SQL
INSERT INTO history (h_c_d_id,
  h_c_w_id, h_c_id, h_d_id,
  h_w_id, h_date, h_amount,
  h_data)
VALUES (:c_d_id, :c_w_id,
  :c_id, :d_id, :w_id, :h_date,
  :h_amount, :h_data);

INSERT into heap file
Cost = 2D
C.4 The Order-Status Transaction

<table>
<thead>
<tr>
<th>Source SQL Statements</th>
<th>Formulas for Service Demands</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>if</strong> (c_id != 0){ /<em>Customer selects BY C_ID</em>* EXEC SQL</td>
<td>By customer last name b-tree unclustered index, partial match range search</td>
</tr>
<tr>
<td>SELECT c_balance, c_first, c_middle, c_last INTO :c_balance, :c_first, :c_middle, :c_last FROM customer WHERE c_w_id = :w_id AND c_d_id = :d_id AND c_id = :c_id; } else { /<em>Customer selects POR C_LAST</em>/ EXEC SQL SELECT count(c_id) INTO :cont FROM customer WHERE c_last = :c_last AND c_w_id = :w_id AND c_d_id = :d_id; EXEC SQL DECLARE c_porlast2 CURSOR FOR SELECT c_id, c_first, c_middle, c_balance FROM customer WHERE c_w_id = :w_id AND c_d_id = :d_id AND c_last = :c_last ORDER BY c_first; EXEC SQL OPEN c_porlast2; for (i = 0; i &lt; cont/2; i++){ EXEC SQL FETCH c_porlast2 INTO :c_id, :c_first, :c_middle, :c_balance; } /<em>end for</em>/ } /* end if*/</td>
<td>Given that the file is sorted on the key, it will be a range search, however the DB pages depend on the qualifying number of pages and not the number of records. Cost = D( log_RB + # of matching pages) = D(1.33 + 250) = 251.33D</td>
</tr>
<tr>
<td>EXEC SQL DECLARE cur_ordenes CURSOR FOR SELECT o_id, o_entry_d, o_carrier_id FROM orderr WHERE o_w_id = :w_id AND o_d_id = :d_id AND o_c_id = :c_id ORDER BY o_id DESC; /*in</td>
<td>By customer id b-tree unclustered tree index: equality on index key cost = D(1+ log_RB) =D(1+ 1.33) = 2.33D</td>
</tr>
<tr>
<td></td>
<td>Average: 60% by customer last name, 40% by customer id total cost = [(.6)(251.33) + (.4)(2.33)]D = 151.93D</td>
</tr>
<tr>
<td></td>
<td>b-tree unclustered index, partial match range search Given that the file is sorted on the key, it will be a range search, however the DB pages depend on the qualifying number of pages and not the number of records. Cost = D( log_RB + # of matching pages) = D(1.34 + 9) = 10.34D</td>
</tr>
</tbody>
</table>
descending order of the serial number*/

EXEC SQL OPEN cur_ordenes;

EXEC SQL DECLARE cur_ord_lines
CURSOR FOR
SELECT ol_i_id, ol_supply_w_id,
ol_quantity, ol_amount,
ol_delivery_d
FROM order_line
WHERE ol_w_id = :w_id AND
ol_d_id = :d_id AND ol_o_id =
:o_id;

EXEC SQL OPEN cur_ord_lines;

b-tree unclustered index, partial match range search
Given that one transaction is running at a time, we
assume that the order-lines of one order will be on one
DB page. Hence, the cost will be by number of pages not
number of matching records.

Cost = D( log_{RB} + # of matching pages)
= D(1.76 + 1) = 2.76D

C.5 The Delivery Transaction

<table>
<thead>
<tr>
<th>Source SQL Statements</th>
<th>Formulas for Service Demands</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (no_d_id = 1; no_d_id &lt;= 10; no_d_id++){</td>
<td></td>
</tr>
<tr>
<td>EXEC SQL SELECT min(no_o_id)</td>
<td>b-tree unclustered tree index: equality on index key</td>
</tr>
<tr>
<td>INTO :no_o_id</td>
<td>cost = 10xD(1+ log_{RB}) = 10xD(1+ 0.98) = 19.8D</td>
</tr>
<tr>
<td>FROM new_order</td>
<td></td>
</tr>
<tr>
<td>WHERE no_w_id = :w_id AND no_d_id = :no_d_id;</td>
<td></td>
</tr>
<tr>
<td>EXEC SQL DELETE</td>
<td>b-tree unclustered tree index: equality on index key</td>
</tr>
<tr>
<td>FROM new_order</td>
<td>search plus DELETE:</td>
</tr>
<tr>
<td>WHERE no_o_id = :no_o_id AND no_w_id = :w_id AND no_d_id = :no_d_id;</td>
<td>cost =10x[ Search + 2D]</td>
</tr>
<tr>
<td></td>
<td>the initial DB pages are in the buffer,</td>
</tr>
<tr>
<td></td>
<td>cost =10x[ 0 + 2D] = 20D</td>
</tr>
<tr>
<td>EXEC SQL SELECT o_c_id</td>
<td>b-tree unclustered tree index: equality on index key</td>
</tr>
<tr>
<td>INTO :o_c_id</td>
<td>search plus UPDATE:</td>
</tr>
<tr>
<td>FROM orderr</td>
<td>cost =10x[ Search + 2D]</td>
</tr>
<tr>
<td>WHERE o_w_id = :w_id AND o_d_id = :no_d_id AND o_id = :no_o_id;</td>
<td>the initial DB pages are in the buffer,</td>
</tr>
<tr>
<td></td>
<td>cost =10x[ 0 + 2D] = 20D</td>
</tr>
<tr>
<td>EXEC SQL UPDATE orderr</td>
<td></td>
</tr>
<tr>
<td>SET o_carrier_id = :o_carrier_id</td>
<td></td>
</tr>
<tr>
<td>WHERE o_w_id = :w_id AND o_d_id = :no_d_id AND o_id = :no_o_id;</td>
<td></td>
</tr>
<tr>
<td>EXEC SQL UPDATE order_line</td>
<td>b-tree unclustered index, partial match range search</td>
</tr>
<tr>
<td>SET ol_delivery_d = :ol_delivery_d</td>
<td>Given that one transaction is running at a time, we</td>
</tr>
<tr>
<td></td>
<td>assume that the order-lines of one order will be on one</td>
</tr>
<tr>
<td></td>
<td>DB page. Hence, the cost will be by number of pages not</td>
</tr>
</tbody>
</table>
WHERE ol_o_id = :no_o_id AND
ol_w_id = :w_id AND
ol_d_id = :no_d_id;

EXEC SQL
SELECT sum(ol_amount)
INTO :c_balance
FROM order_line
WHERE ol_o_id = :no_o_id AND
ol_w_id = :w_id AND
ol_d_id = :no_d_id;

EXEC SQL
UPDATE customer
SET c_balance = c_balance +
:balance,
c_delivery_cnt =
c_delivery_cnt + 1
WHERE c_w_id = :w_id AND
c_d_id = :no_d_id AND
c_id = :o_c_id;

WHERE ol_o_id = :no_o_id AND
ol_w_id = :w_id AND
ol_d_id = :no_d_id;

WHERE d_id = :d_id AND
d_w_id = :w_id;

EXEC SQL
SELECT COUNT(DISTINCT (s_i_id))
INTO :lowstock
FROM stock, order_line
WHERE ol_w_id = :w_id AND
ol_d_id = :d_id AND
ol_o_id < :d_next_o_id AND
ol_o_id >= :d_next_o_id -20
AND s_w_id = :w_id AND
s_i_id = ol_i_id AND
s_quantity < :threshold;

EXEC SQL
SELECT d_next_o_id
INTO :d_next_o_id
FROM district
WHERE d_id = :d_id AND
d_w_id = :w_id;

C.6 The Stock-Level Transaction

<table>
<thead>
<tr>
<th>Source SQL Statements</th>
<th>Formulas for Service Demands</th>
</tr>
</thead>
</table>
| EXEC SQL SELECT d_next_o_id INTO :d_next_o_id FROM district WHERE d_id = :d_id AND d_w_id = :w_id; | b-tree unclustered tree index: equality on index key search plus UPDATE: 
\[
\text{cost} = 10x[\text{Search} + 2D] = 10x[D(1+ \log_2 \text{RB}) + 2D] = 10x[D(1 + 1.33 ) + 2D ] = 43.3D
\] |
| EXEC SQL SELECT COUNT(DISTINCT (s_i_id)) INTO :lowstock FROM stock, order_line WHERE ol_w_id = :w_id AND ol_d_id = :d_id AND ol_o_id < :d_next_o_id AND ol_o_id >= :d_next_o_id -20 AND s_w_id = :w_id AND s_i_id = ol_i_id AND s_quantity < :threshold; | This will be an nested-index JOIN, with the ORDER-LINE rows in the outer-loop and the inner loop for 20 orders. Given that one transaction is running at a time, we assume that the order-lines of one order will be on one DB page. 
\[
\text{ORDER-LINE: b-tree unclustered tree index: range search for 20 orders. Given that one transaction is running at a time, we assume that the order-lines of one order will be on one DB page. cost} = D(\log_2 \text{RB} + \# \text{ of matching pages}) = D(1.76 + 20) = 21.76D
\] |
| | Assuming 10 items per order, this gives 200 items b-tree unclustered tree index: range search cost = D(\log_2 \text{RB} + \# \text{ of matching records}) = D(1.47 + 200) = 201.47D |
Appendix D: QNAP2 Model

QNAP2 is a software tool for describing and solving queueing networks. It provides a collection of solution methods for queueing network models, including exact and approximate methods and discrete event simulation. In addition, the tool has a Pascal-like language for model description, analysis control and result representation. The model parameters are specified for the tool, i.e. number of customer classes, arrival rates, service demand for each server and routing probabilities. Models are solved by invoking QNAP2 on the command line with the model description as input to the tool. The tool solves the model based on the method specified in the description and produces the results. In this Appendix, an example of a QNAP2 model description for the TPCC-UVA queueing network models is presented.

D.1 Queueing Network Model Description

The TPCC-UVA clients are described in QNAP2 in Figure D.1. Each client, up to the maximum number of clients, will choose a transaction using a weighted random function. The client waits a constant transaction keying time, then sends the transaction customer to the transaction monitor and waits until it receives a signal that the customer has completed (left the queueing network). After transaction completion, the client will wait an exponentially distributed think time and the process starts again.
/station/ name = clients(1 step 1 until maxcus);
type = source;
service = begin

if time=0 then
  set(cust_out); &initialize the flag
wait(cust_out); &wait for a customer to leave the network
reset(cust_out); &reset flag to prevent other customers from entering

if time<>0 then &think time for previous customer class leaving the network
begin
  cl:=c(curr_cl);
  exp(cl.lamda);
end
else cst(0.0000001); &entered when time=0, so memory does not overflow

wran := rint(1,100); &random number between 1 and 100

if wran <=43 then
  cl:=c(2) &Payment
else if wran <=47 then
  cl:=c(3) &Order-Status
else if wran <=51 then
  cl:=c(4) &Delivery
else if wran <=55 then
  cl:=c(5) &Stock_Level
else cl:=c(1); &new order

customer.cl_id:=cl.idcl;
customer.sender:=idq; &let current customer take the client id
curr_cl:=cl.idcl; &assign this client, the class of current customer
cst(cl.key_time); &min constant keying time of user, for chosen transaction

if server(cl.entrytab).nb >= server(cl.entrytab).N then
begin
  ndrop:=ndrop+1;
  transit(out);
  set(cust_out);
end
else
begin
  cl:=c(customer.cl_id);
customer.b_time:=time;

  if cl.idcl=4 then
  begin
    set(clients(customer.sender).cust_out);
    &set flag, only for delivery transaction when queued
  end;

  transit(server(cl.entrytab),c(cl.idcl));
end
end; &client description

Figure D.1 QNAP2 description of the TPCC-UVA clients.
Figure D.2 details the description of the queueing servers. Each server will service the current customer/transaction based on its specified service demand. The service demands for the Order-Status transaction are calculated based on the number of New-Order transactions already executed. After the customer completes service, based on its routing probabilities it will move to the next server or leave the network. When a transaction leaves the network the transaction monitor and client are signalled.

```
/station/ name = server(1 step 1 until maxq);
sched = fifo;
service = begin
L2:
   if idq=5 then &count number of new-order transactions in order table
      begin
         if customer.cl_id=1 then
            c_order:=c_order+1;
         end;
      end;
   if cl.entrytab=idq then &if server is the transaction monitor
      begin
         q:=server(idq);
         if q.nbin=1 then set(tm_out);
         &for server 10 which represents the transaction monitor
         wait(tm_out); &wait for a customer to leave the network
         reset(tm_out); &reset flag to prevent other customers from entering
      end
   else begin
      if idq=5 then &if server is the order table
         begin
            if customer.cl_id = 3 then &order-status transaction
               begin
                  prob:=c_order/maxcus;
                  num_pg:=c_order/374;
                  if prob>=num_pg then
                    begin
                      exp((miou(idq)+num_pg)*0.001199);
                    end
                  else
                    begin
                      exp((miou(idq)+prob)*0.001199);
                    end;
               end
            else
               &other transactions in the order table
               begin
                  exp(miou(idq)*0.001199);
               end;
         end
      end
      else
         &other transactions
      begin
         exp(miou(idq)*0.001199);
      end;
   end
```
else &other tables
  begin
    exp(miou(idq)*0.001199);
    end;
  end;
  norm:=1.0;

if trans0(idq)>0.0 then
  if draw(trans0(idq)) then
    begin
      if customer.cl_id<>4 then
        &delivery already set the flag on first entry table
        begin
          set(clients(customer.sender).cust_out);
          &read current customer's sender id to set its flag
        end;
      set(tm_out);
      &the tm waits for the network to become empty before allowing anyone in
      customer.e_time:=time;
      transit(out);
      goto L1;
    end
  else norm:=norm-trans0(idq);
  end

for m:=1 step 1 until M do
  begin
    for j:=1 step 1 until R do
      begin
        if trans(idq,m,j)>0 then
          if draw(trans(idq,m,j)/norm) then
            begin
              q:=server(m);
              if q.nb=q.N then
                begin
                  goto L2;
                end
              else transit(q,c(j));
            end
          else norm:=norm-trans(idq,m,j);
      end
  end
L1: end;  & service ends

Figure D.2 QNAP2 description of the TPCC-UVA queueing network servers.
References


[31] Doxygen, "PostgreSQL Source Code: PGROOT\src\include\storage\temptr.h," 2010.


