Imperial College of Science, Technology and Medicine
Department of Computing

Evaluating the Performance of Transaction Workloads in Database Systems using Queueing Petri Nets

David Coulden

Supervisor: William Knottenbelt
Co-supervisor: Rasha Osman
Second Marker: Peter Harrison

Submitted in part fulfilment of the requirements for the
MEng Honours degree in Computing of Imperial College, June 2013

Abstract

Relational databases are used to provide data storage in countless software applications across a broad variety of industries. The performance of database systems however is difficult to predict as a database system involves many complex interactions with both logical and physical resources making it challenging to create database designs that will satisfy the performance requirements of the application. Thus there is great potential for methodologies and tools that evaluate the performance of these systems which can be used to aid in the design process.

Many methodologies for database performance analysis have been proposed however very few attempts have been made to model the locking process as it is implemented in a database management system. We present QPNPED, a methodology that maps database transaction traffic to a Queueing Petri net model that estimates database performance under that traffic. QPNPED focuses on modelling correctly the concurrency control mechanisms that are used in real database systems and produces models that require minimal parameterization. We demonstrate that models generated by QPNPED are capable of accurately estimating the performance of database systems by applying it to a case-study based upon the readers-writers problem.

We then developed AutoQPNPED, a tool that automates the QPNPED process and gives a pipeline from database traffic specification through to performance results. It provides a straightforward and easy to specify method of applying QPNPED without the user needing to understand the underlying process. We show its effectiveness by modelling some more complex scenarios such as the pgbench benchmark and comparing the results to a measured system.

Acknowledgements

I would like to thank the following people for their help with this project:

- My supervisor Dr. William Knottenbelt, for his many useful ideas and positive attitude that helped me maintain a positive outlook through the project.
- My co-supervisor Dr. Rasha Osman, for keeping me sane when results were not quite what I expected, giving me good advice and guidance throughout the project and for spending a large amount of her free time reading my extraordinarily long draft report sentences.

Contents

Chapter 1

Introduction

1.1 Motivation

Advances in the capacity of storage mediums has made it possible to store large amounts of data at affordable costs allowing organisations to store more data than ever before. Combining this with an increasing number of the people being connected to the internet, leads to a higher demand and need for organisations to store larger amounts of data. A large proportion of this data storage is managed by databases. Although NoSQL databases are becoming increasingly popular there is still a strong industry dependency on relational databases and their use is forecasted with steady growth to 2016 [1]. This high demand makes database system performance a critical concern for an organization.

Many methodologies have been proposed for database system evaluation. However through the complex nature of database system performance and the lack of strong tool support they have not been applied in industry. Therefore there is considerable space for growth in this area and great potential in a methodology that can be applied in industry. A large number of the methodologies were surveyed by Osman and Knottenbelt[23]. They found Queueing Networks were often used to evaluate database performance and one study using Petri nets was also suggested [2]. However to the best of our knowledge there were no studies employing Queueing Petri nets.

Database system performance is impacted by many interdependent factors (e.g. disk contention, concurrency and lock contention, database buffer management). Due to this it is very difficult to create effective performance models of database systems. The area of concurrency and lock contention in particular can create bottlenecks that are very difficult to predict for even experienced database designers. The mechanics of transactions possessing multiple locks at once and transactions queueing until the appropriate lock is available for acquisition is difficult to model in many formalisms. Queueing Petri nets however show potential in being able to model the locking dynamics effectively due to their capability to represent simultaneous resource possession and blocking present in locking systems.

1.2 Objectives

The aims of and objectives of this thesis are:

- To develop a methodology performance evaluation methodology for database transaction traffic workloads based on Queueing Petri nets. The methodology should meet the following aims:
  - Provide a simple mapping from transaction traffic workloads to Queueing Petri net models.
  - Have the capability of modelling database locking mechanisms as they are implemented in database management systems.
  - Provide feedback that is useful for database designers.
- To automate this methodology so that the process of taking a transaction workload to Queueing Petri net model to performance results is simple and requires minimal user interaction.
- To demonstrate the applicability and effectiveness of the methodology in various case studies.

1.3 Contributions

The contributions of this paper are as follows:

- We propose a database performance evaluation methodology using Queueing Petri nets, QPNPED, with a focus on modelling database concurrency control. The methodology is based upon the work of Osman on QuePED [24], a methodology to translate database designs to Queueing network models. It is adapted to map transaction workloads to Queueing Petri net models that provide accurate modelling of database system locking mechanisms. We show the methodology using table-level non-transactional locking.
- We evaluate the methodology by applying it to the readers-writers problem [25]. This involved applying the QPNPED methodology to a formulation of the readers-writers problem to produce a QPN model. The model was constructed using QPME [30], a QPN builder and simulator. The model was evaluated by simulation and compared against the measured performance of the system. We successfully show that the QPN model is able to predict performance, under circumstances of high contention, with error of only 6%. While also following the performance trend of the measured system under all workloads.
- We developed AutoQPNPED, a tool that takes as input transaction workload specifications and outputs a QPN model according to the QPNPED methodology. The output model is in a format that can be simulated and viewed using QPME. AutoQPNPED can then invoke SimQPN, the QPME QPN simulator, to simulate the generated model over a range of client amounts. This allows a user to go from transaction workload specification to transaction performance estimations without the need for any significant amount of interaction with AutoQPNPED. We also describe our design choices and the structure of AutoQPNPED with the aim of it being applicable for use in industry.
- We evaluate the AutoQPNPED tool by modelling pgbench [33], the benchmarking tool provided by PostgreSQL, and by modelling an adapted reader-writer formulation that uses JOIN statements. We show the functional correctness of the tool and the ease of specification it provides. We investigate the effectiveness and limitations of AutoQPNPED and the QPNPED methodology.

1.4 Dissertation outline

The rest of the thesis is as follows:

Chapter 2 describes background material in database theory and in Queueing Petri nets. In addition the QuePED [24] methodology is described that our work is based largely upon.

Chapter 3 introduces the QPNPED methodology that maps transaction workloads to Queueing Petri net models. The methodology focuses on modelling accurately the concurrency control mechanisms used in DBMSs. The chapter describes the mapping when modelling table-level non-transactional locking.

Chapter 4 applies the QPNPED methodology to a formulation of the readers and writers problem [25]. We present the resulting Queueing Petri net model and compare the prediction accuracy to a Petri net model that attempts to model the same system, using a series of transaction workloads.

Chapter 5 introduces an automation of the QPNPED methodology, AutoQPNPED. We describe its architecture and each of the stages involved in the automated process.

Chapter 6 details two case studies that were modelled using AutoQPNPED. The first is modelling of the pgbench benchmark where we compare the performance of the generated Queueing Petri net model over a series of databases of differing sizes. The second modelled a variation of the reader and writer problem in which the reader executes a JOIN statement. We investigate the prediction accuracy over a series of transaction workloads.

1.5 Publications

The following publications were produced during work on this project:

- **4th ACM/SPEC International Conference on Performance Engineering (ICPE 2013)** [34] presents a Queueing Petri net model of table-level database locking for the case of reader and writer contention on a single table. We show that the Queueing Petri net predicts measured system performance more accurately than a counterpart Petri net. The content of this WIP paper is presented in Chapter 4. This paper is included in Appendix B.1.
- **20th International Conference on Analytical and Stochastic Modelling Techniques and Applications (ASMTA 2013)** [35] presents Queueing Petri net models of a variety of different locking mechanisms employed by DBMSs. These include row-level strict two-phase locking and multi-version two-phase locking. We show that the Queueing Petri net model was capable of accurate prediction for all the examined locking mechanisms when there was high contention. The paper is included in Appendix B.2.

Chapter 2

Background and Related Work

This chapter begins by describing a series of modelling formalisms, including Queueing Networks and a number of types of Petri net, to build up towards Queueing Petri nets which is the formalism used throughout our work. We then move on to discuss database systems including design, the SQL language and concurrency control. We finish with related work where we describe the QuePED methodology that our work builds from.

2.1 Queueing Networks

A queueing network is a collection of individual queues that are connected together to form a directed graph where customers move along the arcs from one queue to the next when they have been serviced. They are used commonly in modelling shared resource usage as the resource is represented by a server and customers request access the server and must wait in the queue until they are scheduled. If there are multiple paths from a server to other queues then there will be a routing probability associated with each path and the customer will follow the path based upon those probabilities once it has completed service. Customers can be scheduled in a variety of different ways with some common queueing disciplines being [3]:

- **First-come-first-served (FCFS)** - Customers are serviced upon order of arrival.
- **Last-come-last-served (LCFS)** - The last customer to arrive will be serviced first.
- **Random** - Customers are serviced in a random order.
- **Processor sharing (PS)** - Customers are serviced in parallel with each an equal share of the servers capacity.

Queueing Networks come in two main types:

- **Open** queueing networks have customers arriving from outside the network and customers can also leave the network once service is complete.
- **Closed** queueing networks have all the customers existing within queues in the network and no customers can leave the network, therefore the total number of customers inside the network is constant.

Customers can be separated into classes and each class can then have different routes taken through the network and require different amounts of service time at each server [3]. This allows for a single queueing network to model more complex systems that involve customers with different requirements.

A single queue has multiple properties associated with it, such as arrival distribution, these are often represented using Kendall's notation [4] in the form A/S/c/m/N/D, where:

- **A** is the customer inter-arrival time distribution. It can take M for Markovian/exponential, G for general or D for deterministic.
- **S** is the customer service time distribution. It can take the same values as A.
- **c** is the number of servers in the queue. When this is infinite then the queue is said to have infinite server semantics meaning all customers are serviced in parallel at a rate equal to the rate a single server in the queue services at.
- **m** is maximum number of customers that can be in the queue at once. By default it is infinite.
- **N** is the maximum number of customers in the system with the default being infinite.
- **D** is the queueing discipline the queue uses such as FCFS and LCFS.

Queueing Networks can be solved both analytically or via simulation and the results give details of the performance of the network. Statistics such as mean service time, mean queue length and queue throughput can be found and these in turn should give performance estimations for the underlying system being modelled. Queueing Networks can be used to model systems that depend of resource sharing however they are less appropriate for modelling concurrent systems that require synchronization.

2.2 Petri nets

Petri nets are a modelling formalism used to describe concurrent systems and have applications both as a graphical and mathematical tool. As a graphical tool they can aid in conveying the concurrent behaviours of a system similar to a flow chart [5]. As a mathematical tool Petri nets can be analysed to provide quantitative information about the underlying modelled system. Petri nets were originally proposed by C.A. Petri in 1966 [6] and have since then been developed and extended to model a wider variety of systems.

2.2.1 Regular Petri nets

A standard Petri net is formed from four components [7]:

- Places - Represented by circles. Model conditions or objects.
- Tokens - Represented by black dots. Tokens represent the state of a condition or object and are contained within places.
- Transitions - Represented by rectangles. Model actions that can change the values of conditions or objects.

- Arcs - Represented by an arrow. Specifies connections between places and transitions to indicate which objects are changed by each activity.

A Petri net forms a bipartite directed graph with places in one partition and transitions in the other. Therefore only places and transitions may be connected via an arc and two places or two transitions may not be connected. The fundamental behaviour of a Petri net is defined by two rules [7]:

Enabling a transition A transition is enabled if all the input places of the transition have at least one token.

Firing an enabled transition Any enabled transition can fire. When a transition fires, one token from each of the input places is destroyed and a token is deposited in all of the output places for that transition.

A formal definition of a Petri net is as follows [7]:

Definition 2.2.1. *A regular Petri Net (PN) is a 5-tuple* $PN = (P, T, I^-, I^+, M_0)$ *where*

- $P = \{p_1, ..., p_n\}$ *is a finite and non-empty set of places,*
- $T = \{t_1, ..., t_m\}$ *is a finite and non-empty set of transitions,*
- $P \cap T = \emptyset$*,*
- $I^-, I^+ : P \times T \to \mathbb{N}_0$ *are the backward and forward incidence functions, respectively*
- $M_0 : P \to \mathbb{N}_0$ *is the initial marking.*

The functions I^- and I^+ denote the connections between places and transitions where I^- is from place to transition and I^+ is the opposite. The functions map to the natural numbers where the numbers represent the number of tokens required in input places for the transition to be enabled in the I^- case and the number of tokens deposited in output places for the transition in the I^+ case. M_0 is the function that maps places to the initial number of tokens in that place.

2.2.2 Coloured Petri nets

Coloured Petri nets are an extension of regular Petri nets that do not mathematically increase the modelling capabilities of the Petri net. Coloured Petri nets address the problem in Petri nets of requiring many identical subnets, one for each different type of process, as merging them into a single subnet would make different processes indistinguishable [8]. They provide the capability to define complex Petri nets in a far more manageable way by merging these identical subnets into a single subnet. A coloured Petri net associates a piece of information to each token known as the *token-colour*, allowing tokens to be categorised. Each transition has *occurrence-colours* associated with it that describe different firing modes [8]. An occurrence-colour defines the colour and number of tokens it needs from each input place as well as the number and colour of tokens that it will deposit in the output places when the transition fires.

To formally define a coloured Petri net a multi-set needs to be defined.

Definition 2.2.2 (Multi-set [7]). *A multiset m, over a non-empty set S, is a function* $m \in [S \mapsto \mathbb{N}_0]$. *The non-negative integer* $m(s) \in \mathbb{N}_0$ *is the number of appearances of the element s in the multi-set m.*

Definition 2.2.3. *Let* S_{MS} *be the set of all finite multi-sets over a set S*

Now the formal definition for a coloured Petri net [7]:

Definition 2.2.4. *A Coloured Petri net (CPN) is a 6-tuple* $CPN = (P, T, C, I^-, I^+, M_0)$, *where*

- P *is a finite and non-empty set of places,*
- T *is a finite and non-empty set of transitions,*
- $P \cap T = \emptyset$
- C *is a colour function defined from* $P \cup T$ *into finite and non-empty sets,*
- I^- *and* I^+ *are the backward and forwards incidence functions defined on* $P \times T$ *such that* $I^-(p,t), I^+(p,t) \in [C(t) \to C(p)_{MS}], \forall (p,t) \in P \times T$,
- M_0 *is a function defined on P describing the initial marking such that* $M_0(p) \in C(p)_{MS}, \forall p \in P$.

The definition has many similarities with that of ordinary Petri nets however it introduces C which maps places and transitions to token-colours and occurrence-colours respectively. The I^- function is again related to enabling transitions however it specifies the number of tokens needed of each token-colour needed to fire the transition for a given occurrence-colour and again the I^+ function the same idea but with the colour and number of tokens being deposited. For example if to fire a transition t for the occurrence-colour x there must be three tokens of token-colour a at place p then I^- would be defined such that $I^-(p,t)(x)(a) = 3$. M_0 again defines the number of tokens at each place however in a coloured Petri net it will define the number of tokens of each token-colour at each place.

2.2.3 Coloured Generalized Stochastic Petri nets

A Coloured Generalized Stochastic Petri net (CGSPN) combines the Coloured Petri net with the Generalized Stochastic Petri net (GSPN). GSPNs were outlined by Marson, Balbo and Conte [9] as an extension of the work of Molloy [10] on Stochastic Petri nets (SPN). A SPN is formed from an ordinary PN and then associating an exponentially distributed firing delay with each transition. A GSPN separates transitions into two categories [11]:

Timed transitions Associates a firing rate which gives rise to random exponentially firing delays for the transition as in a SPN.

Immediate transitions Fire in zero time and take priority over timed transitions when firing. A firing weight is associated such that when multiple immediate transitions are enabled the one to fire is randomly chosen with respect to the firing weights. For example given two enabled immediate transitions t_i and t_j with firing weights w_i and w_j respectively the probability of firing t_i is given by $\frac{w_i}{w_i+w_j}$.

GSPNs allow for far more complex models that can be analysed to provide detailed quantitative information such as transition throughput and mean token residence times. This allows for finding performance bottlenecks in systems as well as for predicting performance. Incorporating colours into a GSPN can simplify it by allowing different tokens to be distinguished from each other. Behaviours, such as firing rates, can then be defined on a per token type basis resulting in a more compact representation. The formal definition for a CGSPN is as follows [7]:

Definition 2.2.5. *A Coloured GSPN (CGSPN) is a 4-tuple* $CGSPN = (CPN, T_1, T_2, W)$ *where*

- $CPN = (P, T, C, I^-, I^+, M_0)$ *is the underlying Coloured Petri net.*
- $T_1 \subseteq T$ *is the set of timed transitions,* $T_1 \neq \emptyset$*,*
- $T_2 \subset T$ *is the set of immediate transitions,* $T_1 \cap T_2 = \emptyset$*,*
- $W = (w_1, ..., w_{|T|})$ *is an array whose entry* w_i *is a function of* $[C(t_i) \mapsto \mathbb{R}^+]$ *such that* $\forall c \in C(t_i) : w_i(c) \in \mathbb{R}^+$ *is a*
  - *Rate of negative exponential distribution specifying the firing delay with respect to the colour c, if* $t_i \in T_1$
  - *Firing weight with respect to colour c, if* $t_i \in T_2$

2.2.4 Queueing Petri nets

A Queueing Petri net (QPN) is the combination of a CGSPN and a Queueing Network to produce a very powerful modelling formalism that has all the synchronization capabilities of a Petri net while also being capable of modelling queueing behaviours. The QPN formalism was created by Bause [12] to address shortcomings in both SPNs and queueing networks. QPNs build upon CGSPNs by introducing a new type of place called a queueing place. These queueing places consist of two components, the queue and a place to deposit tokens (customers). Tokens enter the queueing place by the firing of input transitions, like other Petri nets, however as the entry place is a queue they are placed in the queue according to the scheduling strategy of the queue's server. Once a token has been serviced it is deposited in the deposit place where it can from there be used in further transitions. The queues in these places can have variable scheduling strategies and service distributions giving a large amount of expressivity to the model. A formal definition of a QPN follows [7]:

Definition 2.2.6. *A queueing Petri net (QPN) is a triple* $QPN = (CGSPN, P_1, P_2)$ *where*

- *CGSPN is the underlying Coloured GSPN,*
- $P_1 \subseteq P$ *is the set of queueing places,*
- $P_2 \subseteq P$ *is the set of ordinary places,*

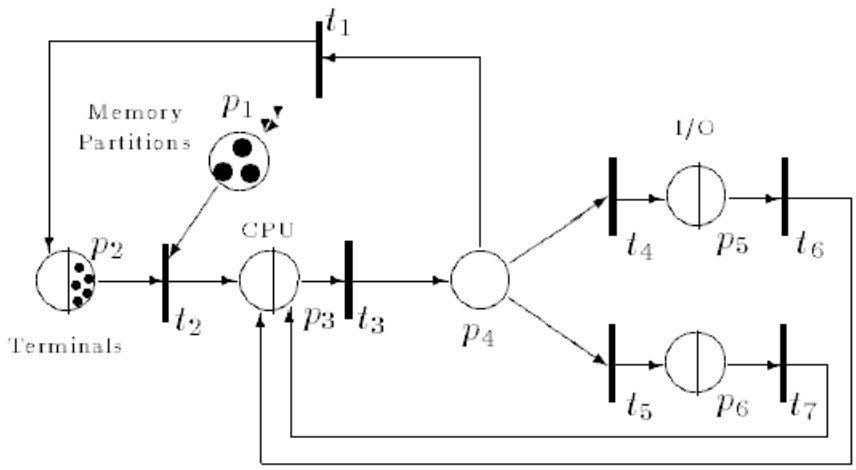


Figure 2.1: Example QPN [12]: The queueing places are graphically represented as a circle with a line through it, splitting the queue from the deposit place.

- $P_1 \cap P_2 = \emptyset,\ P = P_1 \cup P_2$.

A QPN is a powerful modelling formalism as it can model anything a CGSPN can, as it is a special case of QPN where there are no queueing places. A QPN can also represent any queueing network as all queues in the queueing network can be replaced with queueing places and immediate transitions can be added to connect the queueing places in the same configuration the queues were connected. Firing weights on the immediate transitions can then be use to imitate the branch probabilities [12]. Therefore a QPN combines the features of both a CGSPN and queueing network without losing any expressivity, however it does suffer from being less tractable than the formalisms that inspired it.

2.3 Relational databases

A database is an organized collection of data [13] and it can vary from an xml file to a complex relational database requiring management software. The relational model for databases was pioneered by Codd [14] and is based on storing data in a 2D representation in the form of tuples which are stored inside relations. Database management systems (DBMS) are software systems that provide high level access to data in the database however the DBMS also provides facilities for maintaining **data integrity**, managing **concurrency control**, **recovering** from failures and **security**. Examples of DBMSs for the relational model are PostgreSQL [16], MySQL [15] and Oracle [17]. The DBMS provides a high-level language known as a data definition language (DDL) to define table structure. It also provides a data manipulation language (DML) for accessing and modifying data. The most widely used DML is Structured Query Language (SQL) [19] which is based upon relational algebra [20]. Although it has grown in size and expressivity as more useful capabilities are added to the SQL standard. The basics of SQL will be covered in the next section.

2.3.1 Structured Query Language (SQL)

This section covers the structure and syntax of the key SQL statements for retrieving and modifying data in tables.

Data retrieval Data is retrieved from tables using a SELECT SQL statement which is composed of two mandatory clauses, SELECT and FROM, as well as a variety of optional clauses of which the most common is WHERE [18].

- **SELECT** - Declares a list of attributes (columns) to be returned by the query.
- **FROM** - Declares the relations (tables) that the data is being requested from.
- **WHERE** - Specifies a condition on the attributes of the tuples that should be returned.

An example:
SELECT firstname, surname, address **FROM** customer **WHERE** surname = 'smith';

Data modification Data is modified using INSERT, DELETE and UPDATE statements which,as their names imply, insert new tuples into relations, delete tuples from relations and update tuples respectively [18]. Examples of these statements:

- **INSERT INTO** customer **VALUES** ('John', 'Smith', '54 Highland Way');
- **DELETE FROM** customer **WHERE** firstname = 'John';
- **UPDATE** customer **SET** firstname = 'Michael' **WHERE** surname = 'Smith' ;

2.3.2 Conceptual Data Model

A database designer must transform real world data into a form that can be stored within a database. The first stage of this transformation is producing the conceptual data model. The conceptual data model describes the significant objects to store data about and the relationships between these objects. A concrete form of representing the conceptual schema is entity relationship (E-R) modelling [20]. In E-R modelling the data is represented as entities with attributes and relations between entities are described. Figure 2.2 shows a simple E-R diagram concerning two entities. The *Customer* entity has two attributes *Customer No* and *Name* which are data items stored about each customer. The attribute *Customer No* is underlined meaning it is used to uniquely identify a customer. The setup is similar for the *Product* entity. The diamond between the two entities represents a relationship between the two, with the relationship being that the *Customer* buys *Products*.

2.3.3 Logical Data Model

The logical data model is the next step from the conceptual data model and covers transforming the data into a form that is compatible with a relational database. The conceptual data model does not contain any technical details and is a high level view of the data to be stored. Transforming to the logical data model requires moving from entities to relations or tables and attributes become columns. Each column needs to have a specific data type specifying what kind of data will be stored

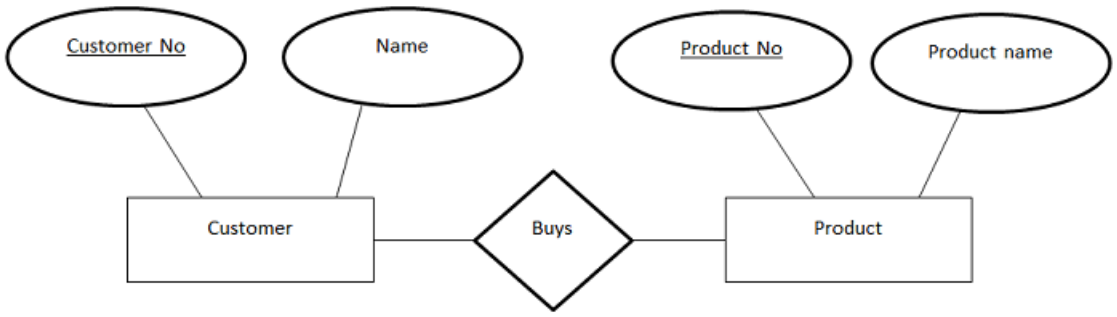


Figure 2.2: An example ER diagram of a customer and product system

in it. Constraints can be placed on columns such as uniqueness for a primary key[1], referential integrity for a foreign key[2].

2.3.4 Physical Data Model

The physical data model describes the physical storage format of the database. Of the three levels of database model it is the least abstract and most technical. Some of the consideration made in the physical data model include [21]:

- **Data files**, the storage files that will be used to store the databases data
- **Partitioning**, The splitting of a database or table into multiple data files which could be in turn in different locations.
- **Indexes**, additional data structures used to allow rapid access to data records based upon the values they contain.

2.3.5 Transaction processing

A transaction is a unit of work in a DBMS that may be made up of multiple execution steps but the DBMS ensures that either all the statements in the transaction succeed and the result is committed (and made visible to other transactions) or the transaction is rolled back. This is done with the intension of a transaction never leaving the database in an inconsistent state. As the transaction can never commit half completed. Therefore as far as other concurrent transactions are concerned the transaction has either ran completely or failed and been rolled back. One of the major tasks of a DBMS is to maintain the ACID properties to guarantee reliable execution of database transactions. These properties are as follows [22]:

- **Atomicity** - Requires that every transaction leaves the database in a state such that it has run to completion or it has failed and made no effect to the database state.

[1] A primary key is a column that uniquely identifies a row in a table [20]

[2] A foreign key is a column that is a primary key in another table. Referential integrity means the values in that column must exist is the primary key column that the foreign key refers to [20].

Lock requested	Lock held	
	Read	Write
Read		X
Write	X	X

Table 2.1: Locking conflicts regular read/write locking

- **Consistency** - Ensures that all transaction executions will take the database from one consistent state to another. This includes that data is valid according to any and all constraints specified.
- **Isolation** - All concurrent executions of a set of transactions will leave the database in the same state as some serial execution of the transactions.
- **Durability** - When a transaction commits the results must be stored permanently even if the database crashes.

Transactions are natively supported in SQL as it provides constructs to define transaction start and end. There are multiple systems that a DBMS uses to maintain these properties such as a transaction log to allow for rollbacks. However what we are most concerned with is concurrent transactions and in particular locking.

2.3.6 Concurrency control

There are multiple issues that can occur during concurrent execution of transactions that violate the ACID properties. Some of the common issues include:

- **Dirty read problem** - This occurs when a transaction reads the uncommitted write of another transaction. If the reading transaction goes on to commit and the writing transaction goes on to abort then incorrect values will have been committed to the database.
- **Lost update problem** - This occurs when two transactions simultaneously the same value from the database and then both update that value. This will mean that one of the updates will be lost and only the last transaction to write will be committed. An update has been lost.
- **Inconsistent analysis** - This occurs when a transaction reads an inconsistent view of the database caused by another transaction being midway through execution. This will result in the reading transaction seeing a view that is neither the result of the second transaction running completely or not running at all which violates the isolation property.

There are multiple ways of handling these problems however the one that is of most concern in this study is locking.

Locking

In database systems there are generally two types of locks read locks and write locks, which as their names imply are required to acquire when reading or writing respectively. A read lock conflicts with write locks but not with other read locks and write locks conflict with both other read and

write locks, shown in Figure 2.1. Each lock is associated with a resource or collection of resources. Locks have a property known as granularity which is a measure of how many resources the lock protects. If the lock protects a large resource, such as a whole database table, then it is known as a coarse grained lock. When the lock only protects a small resource, such as a single record in a table, then it has a fine granularity. Coarse grained locks have a much higher likelihood of causing contention as it will conflict with any finer grained locks that protect a subset of the resources it protects. Locks of varying granularity can be found in database systems with common lock granularities being table-level and record-level which protect the resources that their names suggest.

Database systems that employ locks require transactions to acquire locks of the appropriate granularity based upon the operation performed and the DBMS policies. When a transaction is unable to acquire the locks it requires to proceed (due to another transaction holding conflicting locks) it must wait for those locks to be released for it to proceed. This does however introduce the problem of deadlock however it is not relevant to this study. Once a transaction has completed access the locked resource it can then release the lock and proceed with execution.

The locking form described above where a transaction acquires a lock prior to access and releases post access is known as non-transactional locking. It has the advantage of being a simple locking scheme that provides a large amount of concurrency. But consistency issues can still exist as problems such as *inconsistent analysis* can still happen.

A locking protocol that addresses this problem is two-phase locking (2PL). As the name implies there are two phases in 2PL:

1. Expanding phase: locks are acquired and no locks are released.
2. Shrinking phase: locks are released and no locks are acquired.

A transaction schedule is considered serializable if the result of the executing the schedule is the same as some serial execution of the transactions involved. If a transaction schedule obeys 2PL then it will be serializable[13]. There are two special cases of 2PL of note, strict two-phase locking and strong strict two-phase locking. Strict two-phase locking dictates that all write locks held by a transaction can only be released when a transaction ends. Strong strict two-phase locking dictates that both read and write locks held by a transaction can only be released at the end of the transaction.

When there is contention for a lock the DBMS queues transactions waiting on the lock, so that the transactions will acquire the lock in the order that they requested it in. This ensures fairness for both reading transactions and writing transactions. Without this queueing writing transactions could be starved as a sufficient number of reading transactions could lock out a writing transaction since the read lock would be continuously held.

Locking presents an issue to stochastic modelling techniques. As it involves simultaneous resource possession since transactions can acquire and hold any number of locks at one time. It also requires modelling blocking as a transaction can wait on the availability of a lock. Locking as done in a DBMS can not be modelled accurately using Stochastic Petri nets as they are not able to model the queueing of waiting transactions that occurs. Regular Queueing Networks can not model locking

accurately either as it is not possible to that read transactions block write transactions but not other reads. Queueing Petri nets however can model locking well as they can provide the queueing capabilities that stochastic PNs lack and the blocking capabilities that regular QNs lack.

2.4 Related work

In this section I will cover briefly other work that has been done in the area of database performance evaluation. With a larger focus on QuePED [24] a methodology that we base a large amount of our work on.

2.4.1 Categorisation of Database Performance Modelling Methodologies

Many methodologies for modelling database systems have been proposed and they vary greatly in their focus and approach. Osman and Knottenbelt [23] reviewed a large number of these methodologies that modelled interactions between transactions and a database. They classified them in four main categories based upon the level of detail with which the database transaction's internal design is represented. These categories were:

- **Black box model**: The whole database is represented by as a single queueing service centre. Each transaction class T_i that access the database has an arrival rate λ_i and a service demand μ_i. This methodology was applied distributed databases by giving the single queueing service centre multiple servers, with each server representing a distributed database site [23].

- **Transaction processing model**: The database is represented by the hardware architecture it runs on, using the central server model. Service centres in the network represent hardware components. Each transaction class accessing the database is defined by its service demands on these hardware components and is routed through the system probabilistically [23].

- **Transaction size model**: Each transaction class accesses a number of data objects in the database. These object could be rows, data pages or locks depending on the phenomenon being studied [23]. Studies which fall in this area were found to have a focus in concurrency control methods making it a category of particular relevance to our work. However the findings indicated that most studies only considered exclusive locks. A study by Thomasian and Ryu [36] considered both shared and exclusive lock types They obtained analytical expressions for the probability of lock conflict and waiting time per conflict to produce service demands for transactions. This approach does not model actual lock requests as performed by the DBMS.

- **Transaction phase model**: Each transaction class is represented by a series of phases. Phases are general stages that all transactions classes go through, such as accessing data objects. Each phase is represented by a service centre in the Queueing Network model and routing between phases is probabilistic. QuePED [24] falls into this category and so does our methodology QPNPED (described in Chapter 3).

2.4.2 QuePED: a queueing network methodology for database designs

The QuePED is a methodology for modelling database performance for large databases, targeted especially at databases where I/O performance is a considerable part of database performance.

The methodology does not use a queueing network to model the hardware components, instead the table structure of the database are represented by the queueing network [23]. As input the methodology takes a database design consisting of the following properties [24]:

- For each table:
  - The data types of the attributes and selectivity,
  - The expected number of rows and row length,
  - The index types and structure.
- For each transaction:
  - The rate of occurrence or its percentage of the total transactions,
  - The SQL makeup of the transaction,
  - The transaction structure meaning the procedural statements enclosing the SQL

Using this information tables are mapped onto servers with a infinite capacity FCFS queue and transactions are mapped onto customer classes. Service times for the queues are calculated on a per transaction per table basis using the page usage statistics of the SQL statement in the transaction combined with the time it takes for the underlying storage system to read/write a single page. The queues are connected to form a queueing network such that a transactions customer class will be serviced at tables in the same order as the transaction accesses tables in it's execution.

Table 2.2: Mapping between database designs and queueing network models in QuePED [24]

Database design	**Queueing Network model**
Table	Server
Transaction type	Customer class
Transaction rate of occurrence or percentage of total transactions	Arrival rate or number in system
Cost of I/O DB pages needed to execute the SQL statements of the transaction on a table	Customer class service demand on a server
Order of SQL statements in the transaction	Traversal path of the customer class

Service demands

Service demands are estimated from the expected DB I/O time for the transaction execute SQL statements on the table in question. The cost model used to calculate the DB I/O execution times is specified in [21] and it is based on the file organization of the DB file. The categorizations of file organizations used are:

- Heap file with no index,
- Sorted file,
- Clustered B+ tree file,
- Clustered hash index file,

- Heap file with an unclustered B+ index,
- Heap file with an unclustered hash index.

The considered operations that the SQL statement can perform are [24]:

- **Sequential scan**: Fetch all the rows of the table from disk.
- **Search with equality selection**: Fetch all rows from disk that satisfy an equality condition placed upon an index key field.
- **Search with range selection**: Fetch all rows from disk that satisfy a range condition placed upon an index key field.
- **Insert a new row**: Locate the DB page where the row will be inserted, retrieve that page from disk, modify it to include the new row and write that page back to disk.
- **Update/delete an existing row**: Locate the DB page that contains the row to be updated/deleted, fetch the page from disk, modify it to make the appropriate update or deletion and write the DB page back to disk.

The cost calculations for the various operations on each of the file organizations is summarised in Table 2.3

Table 2.3: I/O DB page cost model for SQL operations [24]

Table Type	**Scan**	**Equality Search**	**Range Search**	**Insert**	**Update/Delete**
Heap	BD	$0.5BD$	BD	$2D$	$Search + D$
Sorted	BD	$Dlog_2B$	$D(log_2B + \#\ of\ matching\ pages)$	$Search + BD$	$Search + BD$
Clustered tree index	BD	$Dlog_FB$	$D(log_FB + \#\ of\ matching\ pages)$	$Search + D$	$Search + D$
Clustered hash index	BD	$1.2D$	$1.2D(\#\ of\ hash\ keys\ in\ range)$	$Search + D$	$Search + D$
Unclustered tree index	$BD(\#\ of\ records\ per\ page + R)$	$D(1 + log_FRB)$	$D(log_FRB + \#\ of\ matching\ records)$	$D(3 + log_FRB)$	$Search + 2D$
Unclustered hash index	$BD(\#\ of\ records\ per\ page + R)$	$2D$	BD	$4D$	$Search + 2D$

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

Applying the methodology

Once the tables have been mapped to servers, transactions to customer classes and estimations of the service times of each transaction table visit have been calculated the queueing network is almost complete. The QuePED methodology outlines an algorithm [24] for calculating the routes for each transaction along with appropriate routing probabilities. This completes the queueing network and performance metrics can be found from running simulations on the network or via analysis algorithms. From the results of this analysis you can get service times for transactions under various transaction loads and a good estimation of performance for the given database design.

Limitations

Currently QuePED transactions are serviced in a FIFO manner with only a single server. This effectively means that all transactions have exclusive access to the database table they are accessing whether they are writing or reading. In reality a DBMSs provide variable degrees of support for concurrent operation. This means the QuePED methodology will underestimate performance of systems in which transactions are allowed to access a table concurrently and as such it is limited in evaluating such concurrent systems.

Chapter 3

QPNPED, a methodology for evaluating database performance using Queueing Petri Nets

In this chapter a methodology is described for mapping transaction traffic compositions for a database system to a Queueing Petri net. The methodology is inspired by QuePED (see Section 2.4.2) as each table in the database is treated as a service station. However unlike QuePED it is designed to be applied to already existing database systems meaning it can not be used to predict performance of potential database designs. But it requires less detailed specification than QuePED as details on the structure of tables, indexes and physical properties such as fill factor are not required to be specified. The core idea of the methodology is to create a Queueing Petri net where the database tables are mapped to timed queueing places and the transactions are mapped to tokens colours that require service at these queueing places. By using QPNs we are able to model concurrency control mechanisms, namely locking, in the models produced by QPNPED which is not provided by QuePED. We are able to model locking since QPNs are capable of modelling blocking and the simultaneous resource possession needed to model the acquisition and release of locks, which was discussed in Section 2.3.6.

3.1 Assumptions

- Each transaction accesses a table at most once at any point in the transaction. This requirement is to ensure that a transaction only requires a single service at any table and as such only requires one service rate for each table.

- Execution times of SQL statements are assumed to be exponentially distributed. This is assumed so that the translated QPNs are amenable to steady state analysis[1].

- The performance of a transaction accessing a table is independent of the number of transactions currently accessing that table.

[1]Currently QPME (QPN construction and simulation tool) does not allow solving QPNs analytically, however it is a feature planned for the tool.

3.2 Transaction traffic specification

The input for QPNPED is a transaction traffic specification. The traffic specification describes the databases expected transaction workload in terms of the definitions of transactions being executed and the proportion each transaction forms of the total traffic. For each transaction definition the SQL statements that are contained within the transaction as well as the procedural structure of the transaction must be known. QPNPED currently supports SQL statements of the forms SELECT, UPDATE, INSERT or DELETE. In terms of procedural statements, IF statements are supported. An IF statement will have a probability of entering the IF statement associated with it and contains a series of further SQL statement or potentially nested IF statements.

3.3 Supported concurrency control mechanisms

In our work we showed that QPNPED can support modelling table-level non-transactional locking. However it has been extended to table-level Strict 2PL, row-level Strict 2PL and multi-version Strict 2PL in a paper for ASMTA 2013 [35] (see Appendix B.2).

3.4 Preparing the specification

For a transaction traffic specification to be mapped to a Queueing Petri net the QPNPED methodology mapping method requires that each SQL statement involved in the specification only accesses a single table. So the first stage of the methodology is to take a transaction traffic specification that includes statements that access multiple tables i.e. JOIN statements, and translate them to a single access specification. The approach taken for this is to represent transaction execution of the JOIN statement as sequential access to the tables involved. The order of the sequential access is given by the order that the tables are accessed in the optimized query tree for the statement. Assuming that DBMS query optimizers decide an execution plan for a transaction using left-deep query trees [21], the order of table access is then given by the left-deep traversal of the JOIN statements query tree.

This gives the order of the statements and the tables they access but to proceed with the mapping we need SQL content corresponding to these accesses. We generate the SQL content by translating the table scans in the query plan into SQL statements. Each table accessed in the JOIN statement will have a scan in the query plan where the table is accessed. The scan will define the fields being retrieved and the conditions on records returned. Each scan is translated into an SQL statement by creating the appropriate SELECT, FROM and WHERE clauses. This is done as follows:

- **SELECT clause**: The SELECT clause will contain all the fields that are specified as being read by the scan.
- **FROM clause**: The FROM clause will simply be the table that is being scanned.
- **WHERE clause**: The WHERE clause will include all the scans filter conditions. If a condition only involves the table being scanned then the condition is added to the WHERE clause without modification. If the condition involves a result from a table scanned previously in the query then the condition will need to be modified before it is added. The modification

is done by replacing the reference to the field in a previously scanned table by the median value of that field in the table. This replacement makes the assumption that the median value will yield the SQL query with the average selectivity of the potential values to be used.

A scan of a table in a query tree may be performed multiple times e.g. for loop nested joins the inner scan is repeated for each record returned from the outer scan, so the number of times each scan will be performed is recorded with the generated statements. This will be used when calculating the service demands of transactions on the table queues. The SQL statements that result from the translation process are ordered according to the table access order derived from the query tree.

3.5 Specifying service demands

At this stage in the methodology the transaction workload specification only involves SQL statements that access single tables. The SQL statements themselves are now used to calculate the service demands that each transaction will require at the table service centres. The calculation of service demands for a transaction token for a given table service centre is the isolated mean execution time of the SQL statement in the transaction that accesses the given table. The isolated mean execution time for an SQL statement is the mean length of time it takes for the DBMS to execute the query when the SQL statement is the only query being executed at that time. This service demand includes all processing done by the query as it measures the query from start to end. For the decomposed JOIN statements the service demand does ignore processing done to perform the JOIN. As the statements that were generated only cover the service demand for scanning the tables in the JOIN. As described previously some of the approximating statements from a JOIN will need to be executed more than once. This is reflected in the service demand by multiplying the service demand by the number of times the statement will need to be performed.

3.6 Mapping to a Queueing Petri net

3.6.1 Building the structure of the Queueing Petri Net

The first stage in mapping a transaction traffic specification involves reproducing the transaction's procedural paths in the QPN model. The result of this is an intermediary form that bears similarities to a Queueing Network and a Queueing Petri net but it's purpose is to describe how transaction tokens move through the service centres. Details about the actual transitions involved will be specified later.

- Each table accessed by any transaction in the traffic specification is represented by an infinite server queueing place. The queueing places are infinite server so that simultaneous access to the table by multiple transactions can be modelled. Service times are assumed to be exponentially distributed with the mean being the service demands calculated in the previous section.

- Each transaction is mapped to a token colour. This token colour will represent the transaction type throughout the mapped QPN. An instance of a transaction running in the database is modelled by a token of the colour corresponding to the transaction.

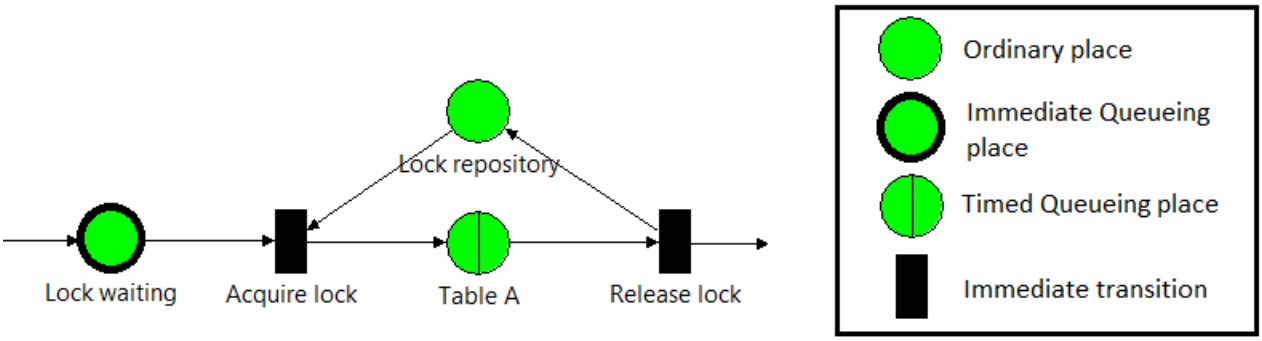


Figure 3.1: The table-level non-transactional locking subnet for Table A

- The service demand of each transaction token at a table service centre are defined in terms of isolated mean execution time of the SQL statement in the transaction that accesses the table.
- The routing between service centres for each transaction token is defined by the order in which corresponding transaction accesses their tables. An IF statement within a transaction causes a fork in the routing where the associated transaction token is routed to the first table accessed within the IF or the first table accessed after the IF. The probability of the transaction token being routed is based upon the probability that the IF statement is entered.
- A network is created from connecting the different table subnets using immediate transitions. The network is based upon the control flow of transactions so that each transaction token accesses tables in the order that they are defined in the transaction.

This structure now defines the path that transaction tokens will take through the service stations and the next stage in the mapping is to translate this to a QPN model that adds modelling of concurrency control mechanisms.

3.6.2 Concurrency control modelling

This process involves mapping each of the table service centres in the previously constructed structure to a QPN subnet. This subnet will include both the service centre as a queueing place and additional places and transitions that define the locking mechanism that is being applied the corresponding table. We will describe the QPNPED mapping that is used for table-level non-transactional locking.

The table-level non-transactional locking mechanism involves two additional places and two additional transitions to the table queueing place to form the locking subnets. The additional components are as follows:

- **Lock waiting place**: This is the entry point to the table subnet, meaning all transaction tokens that need to access the table must enter this place. This is an immediate queueing place, therefore the transaction tokens that enter do not have a service demand here but they must queue to exit the place. This models the DBMS queueing transactions that are blocked at lock acquisition as the only way to leave the place is by firing the *Lock acquire transition*.

This means that as required the transaction tokens in the model will acquire the lock in the same order that they request the lock. Lock request is in effect modelled by the entering of the *Lock waiting place*.

- **Lock repository place**: This is an ordinary place that contains a special type of token colour known as *lock tokens*. These tokens control the acquisition of locks and interact with the *Acquire lock* and *Release lock* transitions that are described below. This place is initialised with a number of lock tokens equal to the number of clients or the maximum number of transactions requiring shared access to the table.
- **Acquire lock transition**: This is an immediate transition that models the acquisition of a lock by a transaction. It has a different firing mode (*occurrence colour*) for each transaction type that needs to request the lock. Each mode has a number of *lock tokens* that required from the *Lock repository place* for the mode to fire. This amount is based upon whether the corresponding transaction type requires exclusive or shared access to the table. If shared access is required only one token is required, however for exclusive access the maximum number of lock tokens is required. This means that when the number of lock tokens is initialised as described above then all the transactions that require shared access can accessed the table concurrently. But only a single exclusive transaction can access the table at any one time and only if there are no shared transactions accessing the table. When one of the transitions firing modes fires the transaction token is removed from the *Lock wait place* and an appropriate number of lock tokens are removed from the *Lock repository place*. The transaction token is then deposited in the entry place for the table's queueing place.
- **Release lock transition**: This is an immediate transition that models the release of a lock by a transaction. Similarly to the *Acquire lock transition* it has a firing mode per transaction type that is accessing the table. The firing mode will return the lock tokens used by the transaction back to the *Lock repository place*. This transition also forms the exit point of the table subnet. Therefore transaction tokens will be removed from the table's queueing place depository and placed at their next table destination once the transition is fired. If there is another table the that the transaction must visit the destination will be the *Lock wait place* for that tables subnet. If it is the end of the transaction then the destination will be the client subnet that we describe in Section 3.6.3.

The mapping described above models table-level locking as there is only a single locking resource per table. It is non-transactional as the lock is released once the transaction has finished it's access to the table. The locking subnet in full is shown in Figure 3.1. With the locking subnet defined we need to specify whether a transaction needs exclusive or shared access to a table. If the accessing SQL statement in a transaction is a read i.e. a SELECT statement, then the transaction will require shared access. Whereas it the statement is a write i.e. UPDATE, INSERT or DELETE statements, then the transaction will require exclusive access.

3.6.3 Client modelling

At this point the whole QPN central QPN structure has been constructed and the last piece will be the client subnet. This subnet will output transaction tokens to the beginning of the QPN structure built so far and receive transaction tokens from the end of structure. We have explored two different methods of modelling clients for QPNPED, using client tokens where each token represents a client

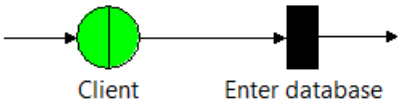


Figure 3.2: The client subnet for the transaction tokens method of modelling clients

or alternatively using transaction tokens to represent clients. Both of them represent closed systems with a fixed number of clients and the potential for defining client think times. In either case the tokens are placed in the entry place of a infinite server queueing place [2] and a client think time can be introduced by setting the service demand of these tokens at the queuing place accordingly.

Client tokens In this method the client queueing place contains a special type of token called client tokens. Each client token will enter the depository of the queueing place once it has completed the think time for the client. From there an immediate transition is used that has multiple modes of execution. Each mode corresponds to a transaction in the traffic specification and the mode will be selected with proportion equal to the proportion of total traffic that transaction represents. When the transition mode fires a client token is removed from the queue depository and the appropriate colour token is deposited in a place that represents the start of transactions. When a transaction token reaches the end of its execution path an end immediate transition will fire depositing the token in the entry place on the client queue place so the cycle continues. This can be conceptually seen as each client executing a transaction with a certain probability and once it has completed the transaction it will choose another to execute probabilistically. This is the more accurate representation of the behaviour of a client in a database system. This method is used in the automated tool in Chapter 5.

Transaction tokens In this method the transaction tokens are stored in the entry place of the client place and they represent a client that will repeatedly execute the transaction corresponding to the token colour with think times between each execution. This results in a simpler Queueing Petri Net as it does not have client tokens and there is no non-deterministic choice between transition modes. However it requires more parameterization as the number of each type of transaction token needs to be specified instead of just a single value for number of clients. It is less realistic than the client token representation as a client that executes the same transaction repeatedly is unlikely to occur in a real system. This method is used in the reader and writer case study in Chapter 4. The client subnet for this method is shown in Figure 3.6.3

This completes the QPNPED mapping from transaction traffic specification to QPN model for table-level non-transactional locking.

Transaction A - 60%	`SELECT * FROM branches;` `SELECT count(*) FROM accounts WHERE id > value;`
Transaction B - 40%	`UPDATE accounts SET balance = balance + 50 WHERE id = 10;` `SELECT name FROM customers;`

Figure 3.3: An example QPNPED transaction workload specification involving two transactions with a 60:40 split of traffic.

3.7 An Example

The example specification in Figure 3.3 depicts a transaction workload formed from two transactions where transaction A forms 60% of the traffic and transaction B 40%. Each of the statements only accesses a single table so no JOIN decomposition needs to be performed. Three tables are accessed by the transactions, *accounts*, *branches* and *customers*. Therefore the results QPN will have three timed queueing places that represent service centres for these three tables.

Next for each of the four SQL statements the isolated mean execution time needs to be calculated as that will form service demands used at the table queueing places. The actual values of the isolated mean execution times are not relevant for the example. But as an example suppose it was found that the mean execution time of "SELECT * FROM branches;" was 2ms. This would mean that the Transaction A tokens would have a service demand of 2ms at the timed queueing place that represents branches.

The QPN that is generated from this example is shown in Figure 3.4. The *Enter database* transition is connected to *Lock waiting branches* place as *Transaction A* tokens will be deposited in there as it is the first table accessed. Similarly for *Lock waiting accounts* and *Transaction B* tokens. Once *Transaction A* has progressed through the branches table it will move on to the accounts table and then back into the *Client* place. *Transaction B* will progress through accounts table and then the customers table. The only statement that requires exclusive access to a table in this case is when *Transaction B* updates the accounts table, all the others only require shared access. The QPN model can then be simulated using a tool such as QPME 2.0 [30] to give performance details such as mean response time for each transaction type.

[2]Ideally this would be represented with a timed transition with rate appropriate for the think time however QPME, the QPN construction and simulation software used, does not currently support them.

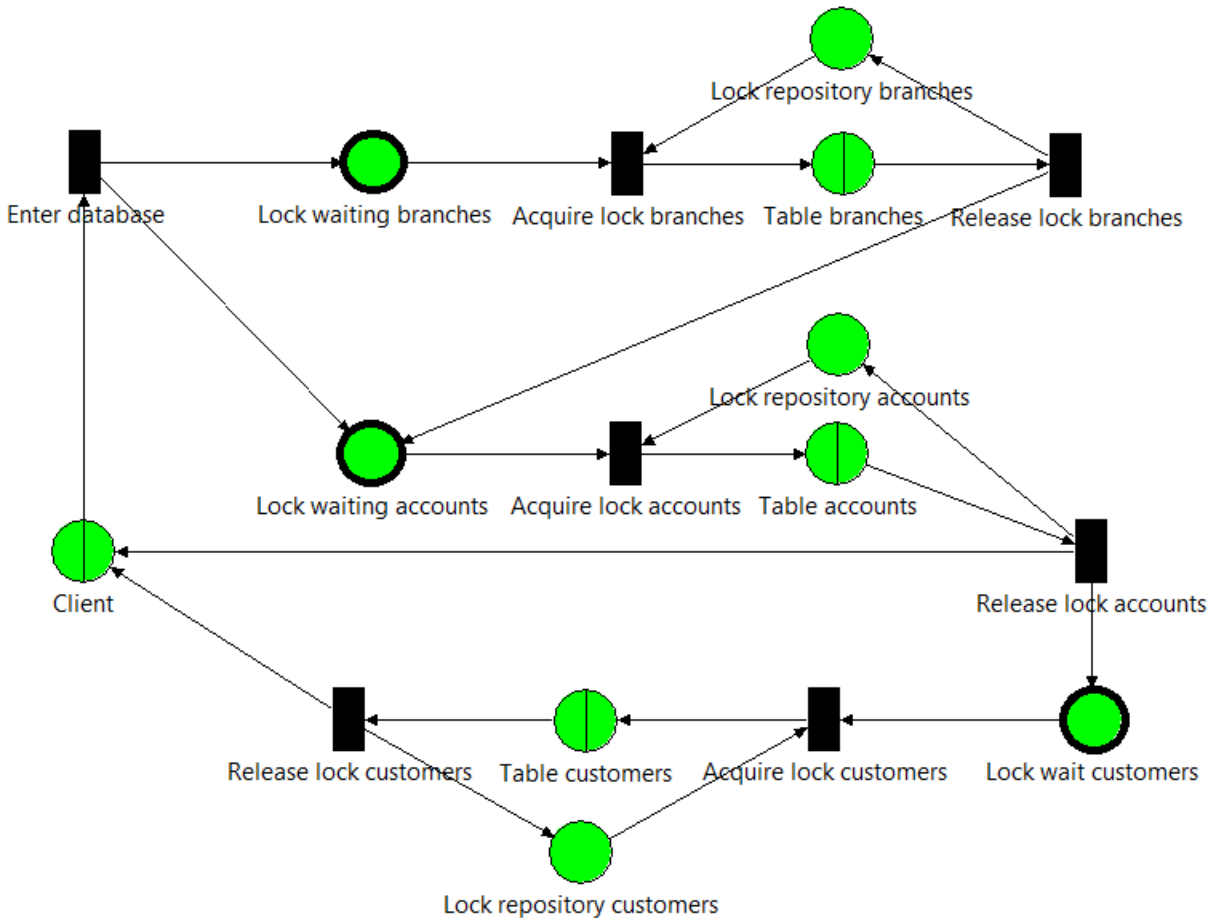


Figure 3.4: The generated QPN model for the example

Chapter 4

Modelling the readers-writers problem

To test the effectiveness of the previously described mapping from transaction traffic to a QPN model, we modelled the readers-writers problem[25] in the context of concurrent access to a single database table. For this study table-level non-transactional locking was used and clients were modelled to repeatedly execute the same transaction. The work described in this Chapter was published in ICPE 2013 [34], the paper can be found in Appendix B.1.

4.1 Measured system

In the experiments PostgreSQL 9.1[27] is used as the measured DBMS and table-level locking is enforced by including explicit table locking statements in the transactions. The measured system has two types of transaction: shared and exclusive that compete for access to a single table (*Table A*) with shared transactions attempting to read while exclusive transactions attempt to modify the table.

4.1.1 Transaction structure

Table A was configured to contain 100,000 rows containing random data that was generated using a simple C program. The contents of the rows is unimportant to the performance of the measured system in this case. Shared transactions perform a sequential full table scan on table A, as no index is utilized by the transaction, to read approximately 1% of the table rows. The exclusive transaction also performs a sequential scan to update the same rows as the shared transaction reads. The exact statements that make up the transactions are described in Figure 4.1. The exclusive transaction includes a PostgreSQL sleep statement to artificially increase the length of the transaction by 40ms. This is done to simulate a TPC-W like workload where exclusive transactions are longer than read transactions. As PostgreSQL does not have table locking by default, each transaction has the a locking statement of the following form:

`LOCK TABLE` *`name`* `IN` *`lock-mode`* `MODE`

The statement will lock the specified table for the duration of the transaction in the specified mode. Conflicting table-level locking modes were chosen for shared and exclusive transactions. These were

Shared transaction

```
BEGIN;
    LOCK TABLE Table A in ACCESS SHARE MODE;
    SELECT count(*) FROM Table A WHERE id > value;
END;
```

Exclusive transaction

```
BEGIN;
    LOCK TABLE Table A in ACCESS EXCLUSIVE MODE;
    UPDATE Table A SET other-id = other-value WHERE id > value;
    SELECT pg_sleep(0.04);
END;
```

Figure 4.1: Structure of shared and exclusive transactions

ACCESS SHARED for shared transactions and *ACCESS EXCLUSIVE* for exclusive transactions. The *ACCESS EXCLUSIVE* also conflicts with itself meaning two exclusive transactions will not be able to concurrently access the table. However *ACCESS SHARED* lock does not conflict with itself meaning that shared transactions can execute simultaneously.

4.1.2 C++ benchmark

A C++ program was developed to measure transaction performance by generating multiple clients that concurrently submit transactions to the database. It takes as input the number of shared and number of exclusive transactions as well as the corresponding SQL definitions for those transactions. Threads are spawned using the POSIX thread library and each thread emulates a client that is either performing shared or exclusive transactions with numbers of each client type based upon the benchmark input.

Each client executes a loop involving a exponentially distributed think time with mean 500ms and then submitting the specified query to the database, see Figure 4.2. The loop continues until the benchmark has run for a certain period of time, in the case of the reader-writer problem the benchmark was run for 5 minutes. The exponential distribution is sampled using the inverse method which is realised by evaluating Equation 4.1 where U is a uniformly distributed random variable between 0 and 1 and λ is the rate parameter.

$$E = \frac{-lnU}{\lambda} \tag{4.1}$$

The libpq library[29], a PostgreSQL library for C and C++, was used to connect to the database and execute queries within the benchmark. Each client thread maintains a connection to the database and measures the response time of the queries it submits as well as the number of queries that it executes and records it in memory. Once the benchmark duration is up the clients exit the loop and clean up the connection and exit. Once all the client threads have exited the main thread wakes up and calculates the response time and transactions per second for each transaction type.

To ensure that each run is fair at the start of each benchmark execution the PostgreSQL state-

```
double sample = sample_exponential_distribution(2);
usleep( sample * 1000000.0 );
if (time(NULL)-start > RUN_DURATION)
  break;
struct timespec startTime, end;
clock_gettime(CLOCK_REALTIME, &startTime);
PGresult* res = PQexec(conn, exclusiveTransaction);
clock_gettime(CLOCK_REALTIME, &end);
```

Figure 4.2: Code snippet showing the core section of the client loop

ments *VACUUM FULL Table A;* and *CHECKPOINT;*. The former removes all the dead records that accumulate as a by-product of the way PostgreSQL performs table modifications which can impact performance of queries if not removed. The latter forces a checkpoint to be performed[1] which could negatively impact performance if it occurred in the middle of running the benchmark.

The measured system was run on an Intel(R) Core(TM) i7-2600 CPU@3.40GHz box running Ubuntu 12.10 64-bit and PostgreSQL 9.1.

4.2 Queueing Petri Net model

The QPN model for the measured system was created using QPME 2.0[30] and is pictured in Figure 4.3. The model was developed by applying the methodology described in Chapter 3. The *Client* place uses transaction tokens to define numbers of each client type and after an exponentially distributed think time with mean 500ms the clients submit their corresponding query to the database (either shared or exclusive). The query being sent by the client is represented by the *Enter database* immediate transition. The *Lock waiting* immediate queueing place has FIFO departure discipline meaning that transactions must leave the place in the order that they entered. This has the effect of transaction tokens acquiring the table lock in the order that the tokens entered the *Lock waiting* place.

A transaction at the front of the *Lock waiting* place queue will proceed using the *Acquire lock* transition when there are appropriate number of lock tokens in the *Lock repository* place for it to execute. For shared transactions they require only one lock token and exclusives require all the lock tokens. The number of lock tokens in the *Lock repository* is equal to the number of shared transactions in the model. This is necessary so that all the shared transactions can acquire the table lock simultaneously. The *Table A* infinite server queueing place represents the execution of the query and once the transaction has been serviced the corresponding transaction token is placed in the depository of the *Table A* place. Transactions that have completed execution in Table A will then fire the *Complete transaction* immediate transition causing the lock tokens to be returned to the repository (representing the release of the lock) and the transaction token is added to the client think time queueing place for the process to repeat.

[1] A checkpoint causes all the changes by committed transactions to be written to the logs and can be an expensive operation.

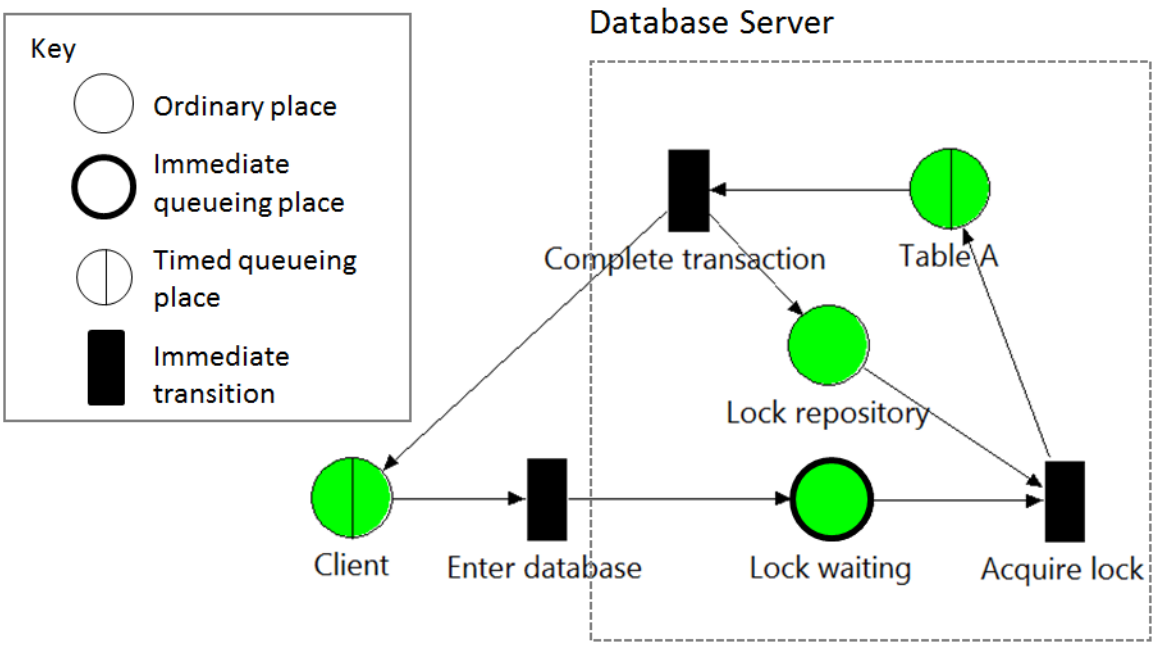


Figure 4.3: Queueing Petri net model of table-level locking.

The service times for each transaction in the *Table A* queueing place are found by isolated execution of each of the shared and exclusive transactions to find their average execution time. This average execution time is used as the mean service demand of the corresponding transaction tokens.

4.3 Petri Net model

The Petri net model is very similar to the QPN model constructed however the *lock waiting* place is no longer a queueing place. It is instead a regular place meaning transactions are not necessarily serviced in order of arrival. The timed queueing places in the QPN model (*Table A* and *Client* places) are ideally represented using infinite service timed transitions. However due to lack of support for timed transitions in QPME they were modelled as equivalent infinite server queueing places with immediate incoming and outgoing transitions. Therefore by removing FIFO departures from the *Lock waiting* place, the model becomes equivalent to a Coloured Generalized Stochastic Petri net.

4.4 Experimental results

4.4.1 Workloads

The experiment used workloads from the TPC-W benchmark, which is an e-commerce benchmark that implements an online bookstore. It defines three workload mixes:[28]:

Transaction type	Mean response time (ms)
Shared	18.8
Exclusive	60.4

Table 4.1: Mean isolated execution time for shared and exclusive transactions

- **Browsing** - 95% reads and 5% writes
- **Shopping** - 80% reads and 20% writes
- **Ordering** - 50% reads and 50% writes

The workloads will show different contention behaviours with browsing expected to show the least contention as it contains the least amount of exclusive transactions and ordering expected to show the most contention as it contains the largest proportion of exclusive transactions.

4.4.2 Parameterizing the models

As described in 4.2 the two transactions are run in isolation to find mean execution times which are be used as service demands. The mean response times calculated are shown in Table 4.1. These mean response times were translated to service rates, by calculating their reciprocal.

4.4.3 Result analysis

Measurement method

The benchmark was run for 180 seconds per trial and each trial was repeated five times to improve the reliability of results. The QPN model was simulated using the method of non-overlapping batch means method (with default QPME settings) to estimate steady state mean token residence times with 95% confidence intervals. For all the simulations the confidence intervals were sufficiently small for the results to be reliable.

Measured system performance

Under browsing workload (Figure 4.4) the shared transactions dominate the traffic and as such experience little change from the increase in exclusive transactions. This indicates that there is not a large amount of contention taking place, even at high numbers of clients. The step-like trend for both of the transactions is due to the number of exclusive transactions being a constant value over a series of client values. The shopping workload (Figure 4.5) shows a large impact on the performance of both transaction types as there is more contention occurring, therefore lock waiting time is increased when compared to transaction execution time. This is particularly noticeable for the shared transactions as they have undergone considerable increase in mean response time compared to the browsing workload. The ordering workload (Figure 4.6) displays approximately equal performance of shared and exclusive transactions indicating that the lock waiting time is dominating the execution time of the transactions to such an extent the difference between the base execution time of each transaction type is irrelevant.

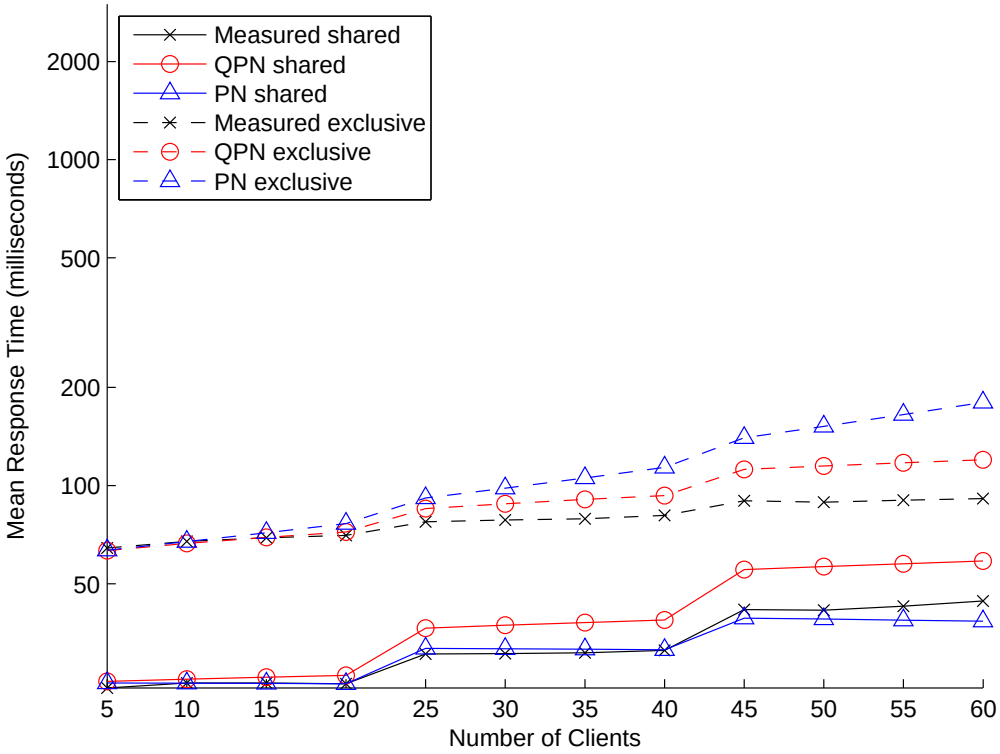


Figure 4.4: browsing (95% shares and 5% exclusives)

Petri net model performance

For the browsing workload the Petri net model underestimates the exclusive transaction performance greatly with a 97% error at 60 clients. Although for shared transactions it overestimates performance with an error of 13% at 60 clients. This is caused by exclusive transaction starvation in the Petri net model as transactions do not queue for lock acquisition, resulting in poor exclusive transaction performance in exchange for better shared transaction performance. The PN model continues to overestimate the performance of shared transactions for shopping and ordering workloads with errors of 83% and 94% at 60 clients respectively. This increase in error is due to the shared transaction response time increasing dramatically for these workloads in the measured system. While the shared transactions in the PN model degrade in performance to a lesser extent as they still can skip over exclusive transactions whenever another shared transaction is running. The opposite effect is seen with exclusive transactions for which the error decrease to 43% for shopping and 17% for ordering at 60 clients. This is due to the larger number of exclusive transactions in the system meaning it is more difficult for them to be starved by the shared transactions.

Queueing Petri net model performance

The QPN model underestimates both transaction types performance for the browsing workload with an error of 32% at 60 clients. Generally the performance prediction of the QPN improves as there is a larger proportion of exclusive transactions in the mix. This is caused by an increase in contention resulting in longer lock wait times which the QPN can model accurately. This is shown

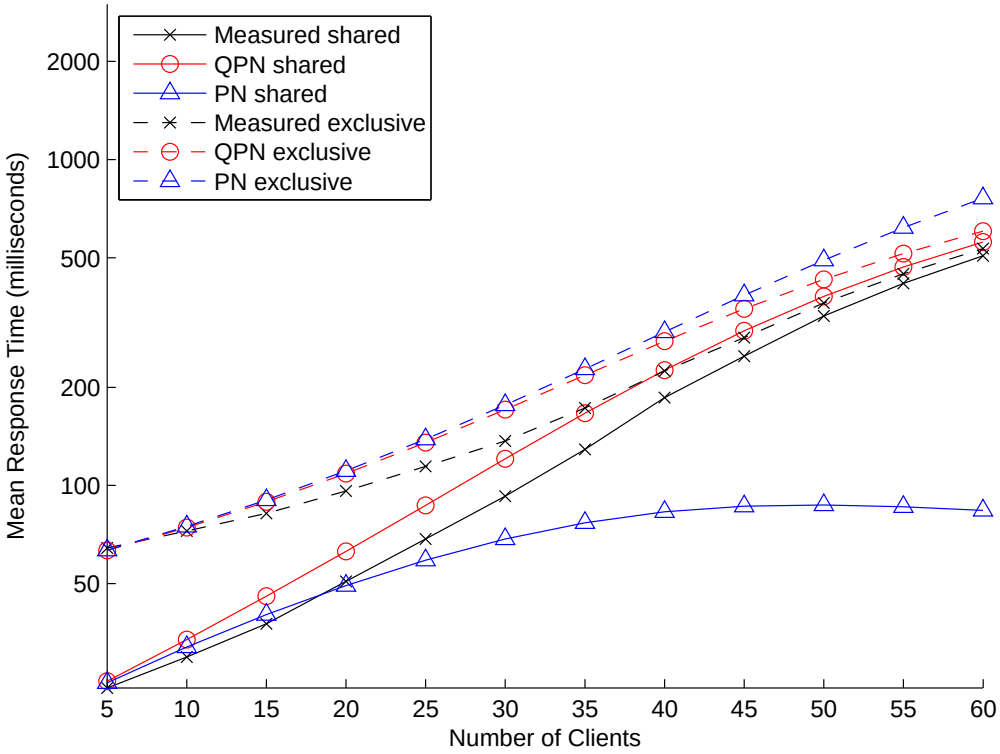


Figure 4.5: shopping (80% shares and 20% exclusive)

in the accuracy of the QPN as the underestimate under shopping traffic improves to only 10% error for shared transaction and 13% for exclusive transactions at 60 clients. The fact that it underestimates the performance could be due to unaccounted multi-core processing or possibly due to the exponential approximation of transaction execution. For the ordering workload the QPN correctly predicts approximately equal response times for shared and exclusive transactions and overestimates performance with an average error of 8% at 60 clients. This overestimate could be caused by a large amounts of updates that result in more disk access which is not modelled by QPN model.

Overall the QPN is capable of following the performance trend for both transaction types at each of the workloads, in particular the performance of the QPN model for shared transactions is far better than the PN model. These results show the potential of QPNs in modelling database performance and indicate that the QPNPED methodology can be used to predict database performance. However the results also show that the effectiveness of the QPNPED methodology is dependent upon whether contention for database locks is a performance bottleneck. Since the QPN model was most accurate under high contention scenarios. The methodology does not directly model physical hardware so it would be expected that bottlenecks derived from physical resources such as CPU or disk may not be predicted as well by the QPN model. We investigate this in Chapter 6.

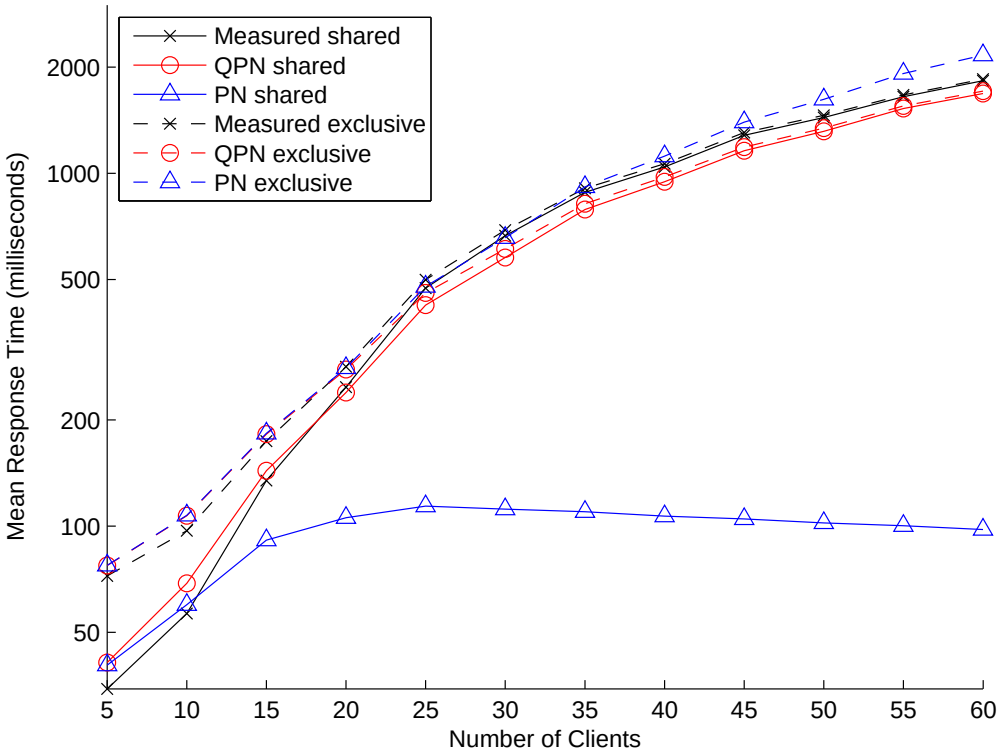


Figure 4.6: ordering (50% shares and 50% exclusive)

Chapter 5

AutoQPNPED, A tool that automates the mapping from traffic specification to Queueing Petri net

5.1 Requirements

The tool is aimed to take as input a transaction traffic specification, consisting of the transactions and their proportions of the total transaction traffic, and generating a Queueing Petri net model representing the traffic based upon applying the QPNPED methodology. The generated QPN needs to be output in a format suitable to be simulated and viewed in QPME 2.0[30]. The aim is for the tool to be flexible and easy to use making it more amenable to be used in an industrial setting. This is in part achieved by using the QPNPED methodology as it does not require the low level details of the database server hardware to be specified.

5.2 System overview

AutoQPNPED was designed in a modular fashion to aid extensibility and reusability in its components. The key modules are as follows:

- **Transaction traffic specification reader** - This module handles input of transaction workload specifications in the form of a simple domain specific language. The grammar for this language is in Appendix A. The input specification is parsed to produce an internal representation of the transaction traffic specification.
- **Query atomization and annotation** - This module processes the internal transaction workload representation to prepare it for the further stages of the tool. This involves splitting queries that access multiple tables into multiple approximating single access queries (atomization). Once the specification has been atomized it undergoes annotation where details such as isolated query execution time are calculated for each query. This leaves a fully specified transaction workload specification suitable to undergo the QPN mapping process.
- **Queueing Network construction** - This module constructs a Queueing Network that represents the procedural structure of the transactions in the transaction workload. The

construction method is based upon that of QuePED described in Section 2.4.2.

- **Queueing Petri net construction** - The Queueing Network created in the previous stage is mapped to a Queueing Petri net by replacing each table queue with a subnet that models the table and it's locking mechanism.
- **QPME XML translator** - Translates the internal representation of the Queueing Petri net model into an XML format compatible with QPME 2.0.
- **QPME runner** - This module automates the running of the QPME QPN simulator over a user specified range of client amounts. A data file is then created that contains the mean response times, from the simulation results, for each transaction type across the various client amounts.

5.3 Language choice

The tool was written in Java for the primary reason of portability. As Java is compiled into an intermediary representation, Java bytecode, it can be run on any platform that is running the Java Virtual machine. As database servers are run on a variety of different systems in industrial settings, it is important for the tool to be capable of running on any many systems as possible. Another important advantage was that the main output targeted was QPME 2.0 which is also written is Java meaning invoking the QPN simulator can be done from within Java making the interaction simpler. The tool also makes use of a variety of libraries to simplify tasks such as parsing input and writing XML, therefore the Java library support was key in developing a simpler system.

5.4 Program structure

5.4.1 Transaction traffic specification reader

Transaction traffic is specified in a simple domain specific language that involves defining transactions and the proportion of total traffic that makes them up. This primary purpose for this component is to read the specification and translate it to an internal representation of the traffic specification. The language grammar can be found in Appendix A and it is defined using ANTLR 4.0[31] for Java. ANTLR generated a parser that is able to parse the input specification from a parse tree and from that a visitor was written to walk the parse tree to generate the internal representation of the traffic specification. This process is then encapsulated to have simple entry points that adhere to a specification reader interface. This has the effect of making input techniques interchangeable allowing the tool to be extended to accept input from different sources in the future.

A traffic specification can come in two different forms, regular and detailed. A regular specification defines the transactions and the traffic proportions and ignores specifying any details about execution times of SQL statements and leaves it to the tool to infer the required properties to create the QPN. A detailed specification will have for each SQL statement the table accessed by the statement, whether it is an exclusive statement and the mean execution time of the statement when executed in isolation. These three pieces of data can be found in the atomization stage of the tool but the user can choose to specify the details themselves to override the default inferences.

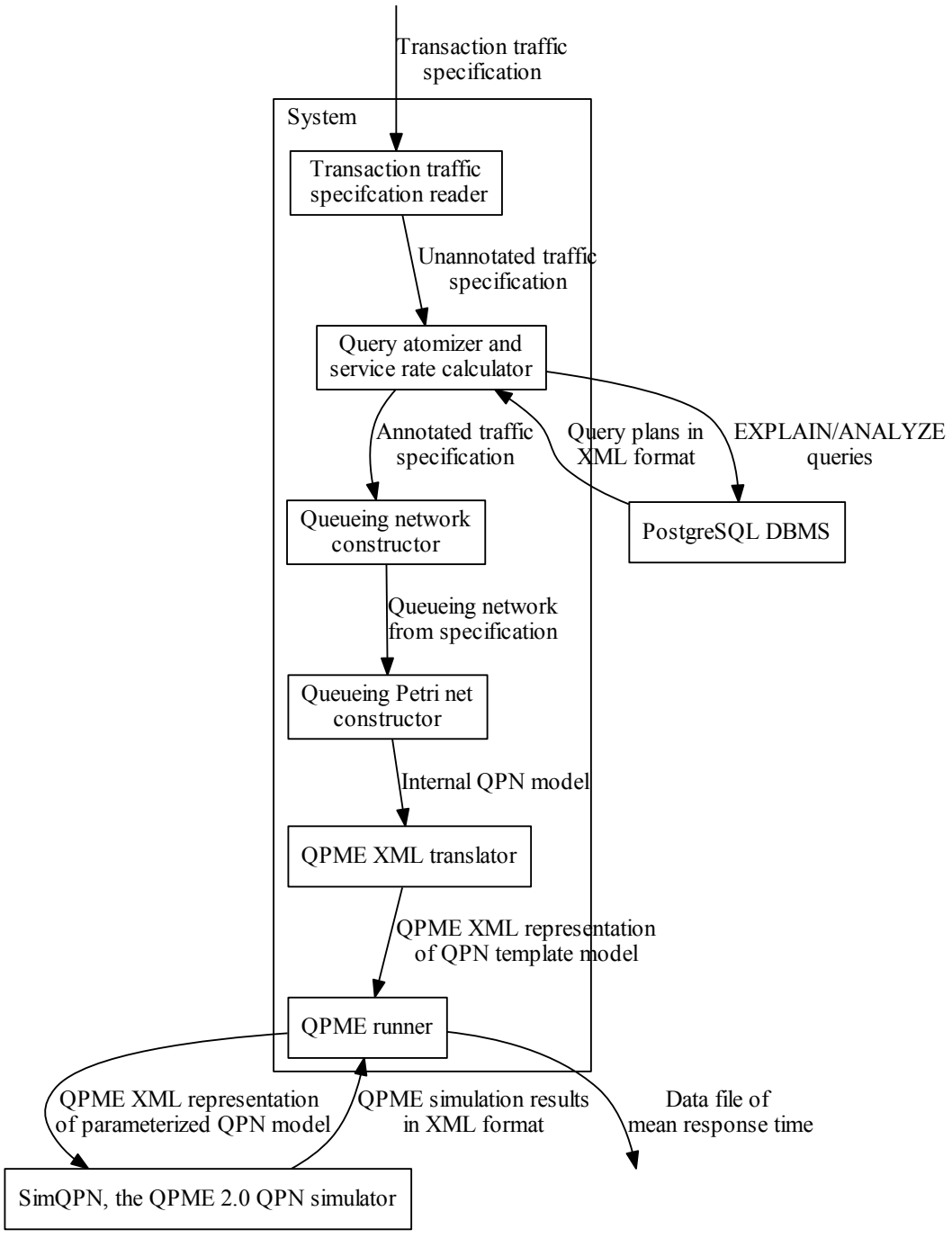


Figure 5.1: Structure diagram for the QPNPED tool showing control flow

```
transaction Trans1 0.3 {
    -SELECT * FROM table_a;
    if 0.2 {
        -UPDATE table_b SET day = 'Monday' WHERE amount > 4;
    }
}
transaction Trans2 0.7 {
    -SELECT day FROM table_b;
}
```

Figure 5.2: A regular transaction traffic specification file with two transaction types

```
transaction Trans1 0.3 {
    statement exclusive=false runtime=0.1 table=table_a {
        -SELECT * FROM table_a;
    }
    if 0.2 {
        statement exclusive=true runtime=0.4 table=table_b {
            -UPDATE table_b SET day = 'Monday' WHERE amount > 4;
        }
    }
}
transaction Trans2 0.7 {
    statement exclusive=false runtime=0.25 table=table_a {
        -SELECT day FROM table_b;
    }
}
```

Figure 5.3: A detailed transaction traffic specification file with two transaction types

This gives the user flexibility as well as allowing them to use specification files that have already been annotated and as such avoiding the computational cost of reannotating them.

The specification in Figure 5.2 shows two transactions defined Trans1 and Trans2 which occur 30% and 70% of the time respectively. Trans1 contains a SELECT statement on table_a and an if statement that is entered 20% of the time. Inside the if statement there is an UPDATE statement to table_b. Trans2 contains a SELECT statement for table_b. Figure 5.3 shows the same transactions however it has been augmented with additional details on the statements to describe the effects of the statements. These details can either be provided by the user or they can be inferred in the atomization stage of the tool.

5.4.2 Query atomization and annotation

This module in the tool takes as input an unannotated traffic specification and loops through each transaction inferring the details of each SQL statement. If the statement accesses multiple tables

```
SELECT * FROM table_a JOIN table_b ON table_a.id>table_b.population
```

Figure 5.4: A simple SQL statement requiring a loop nested join where id is indexed but age is not

it breaks the statement into multiple single access approximating statements. This module of the tool is dependent upon the DBMS being used so the current implementation is for PostgreSQL. It is the only section that is dependent upon the DBMS so implementations for other DBMSs can be written and integrated in with relatively little effort. The start of the atomization and annotation process for any single SQL statement is to perform an EXPLAIN ANALYZE[32] query using the statement to retrieve a query plan from PostgreSQL. EXPLAIN ANALYZE is a PostgreSQL specific statement that returns both the query plan that PostgreSQL used to execute the query as well as details about the execution. A query plan consists of a hierarchy of subplans in which each plan tends to be either a join operation or a scan operation. Query plans are requested in XML format and the rest of the process involves parsing those plans.

Query atomization

Once the query plan has been retrieved it needs to be checked as to whether it accesses multiple tables, since if that is the case it must undergo a splitting process to produce multiple one table accessing queries. The checking process involves traversing the query plan through all the subplans recording the tables accessed, if any, by each subplan. Once the whole query plan has been traversed if multiple tables are accessed then the atomization process begins, if not the statement goes on to be annotated.

The atomization process involves another traversal of the query plan, searching for the table scans involved in the query. The key types of scan being sequential and index. A sequential scan of a table involves reading the whole table to find the data needed and is most often used when a large proportion of the table needs to be read. An index scan involves reading the index to then locate and read matching records in the table and it is used when only a few records match the query and there is an appropriate index available to use.

For each scan a query is constructed attempting to replicate the effects of the scan i.e. a query with a query plan consisting of just the scan. This involves taking from the scan plan the data items that are selected as well as the filter condition and using them to construct the query. However this does not work in general for join cases as the filter condition may involve values from other tables such as in Figure 5.4. This is because this is executed using a nested loop join which means that it will perform an outer scan which is a sequential scan on table_b, then the inner scan, in this situation it is an index scan, is executed once for each record returned by the outer scan. This is shown in Figure 5.5 as the index scan has loops value 239 which means it is executed 239 times. Therefore the query that approximates the inner loop will need to have its execution time multiplied by 239 during the annotation stage. But to separate the join into two independent queries the table_b.population value required by the inner scan needs to be approximated.

This approximation is done by generating a query to select the approximated field and to return it in ascending order. The middle value in the resulting dataset is used as the approximating

```
Nested Loop  (cost=0.00..9191.08 rows=324960 width=145) (actual time=0.097..16.663 rows=49930 loops=1)
  ->  Seq Scan on table_b  (cost=0.00..7.39 rows=239 width=114) (actual time=0.008..0.078 rows=239 loops=1)
  ->  Index Scan using table_a_pkey on table_a  (cost=0.00..24.83 rows=1360 width=31)
                                     (actual time=0.001..0.036 rows=209 loops=239)
        Index Cond: (id > table_b.population)
Total runtime: 17.686 ms
```

Figure 5.5: The readable output from EXPLAIN ANALYZE on the query in Figure 5.4

```
SELECT * FROM table_b;
SELECT * FROM table_a WHERE table_a.id> 10000;
```

Figure 5.6: Result from atomizing Figure 5.4 where 10000 is the middle value for table_b.population

value. This is chosen since the outer query can be assumed to have returned a uniform distribution of values for that field and as such the average value of the inner query can be realised by using the middle value from the outer query. This approximation works best for fields that do have uniformly distributed values however that is unlikely to occur in many cases, making the approximation rough. This method of splitting joins does not take into account additional processing in query execution as it focuses entirely on the scanning of tables. The higher the proportion of time the table scans make up of overall query execution time the better the approximation is. An example of this approximation is shown in Figure 5.6 where the second query will have its isolated execution time multiplied by 239 as that query would have been executed approximately 239 times in the original query. Once the approximating queries are found they are then passed into the annotation stage.

Query annotation

By this stage each query has been atomized such that it only accesses a single table. For each query the mean isolated execution time, whether the query requires exclusive table access and the table the query accesses is inferred and recorded. The table the query has accessed is already known if the query went through the atomization process and it is taken straight from the query plan so it is straight forward to record this in the query object. For exclusivity the default approach is that if a SQL statement is modifying a table then it will require exclusive access to that table, therefore again the query plan is used to find out if the query is modifying the table or not and the result is recorded.

Isolated execution time calculation takes more time and effort as the query must be submitted to the PostgreSQL database repeatedly and for sufficient number of trials to give a good estimate of execution time. The approach taken is that the query is executed and the time taken is measured and recorded, this repeats until the required number of executions is reached and the mean is calculated. A possible alternative to give isolated execution times is to use EXPLAIN ANALYZE statements, as they give the length of time that each part of the query takes to run. However they are not suitable for use as EXPLAIN ANALYZE has significant profiling overhead[32] that obscures the real isolated query execution time. The speed at which the mean execution time can be calculated is based upon both the length of the query and the number of trials required, there-

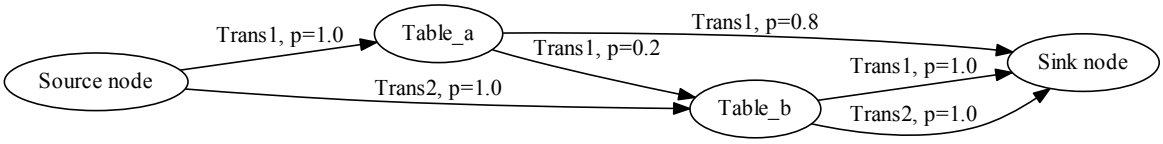


Figure 5.7: Example network formed from the specification in Figure 5.2

fore the number of trials required can be configured by the user allowing them to balance the trade between speed and accuracy. When the mean isolated execution times for each SQL statement has been calculated the specification is now considered detailed and if the user wishes can be output so that any future runs of that specification will not need to go through statement annotation again.

5.4.3 Queueing Network construction

The first stage in translating an annotated transaction traffic specification to a QPN model is to translate it to a queueing network that represents the procedural structure of each of the transactions. Each transaction is translated to a token colour and each table accessed by any transaction is translated to a infinite server queue. For each SQL statement that accesses a table the isolated execution time is used to define a service demand for the corresponding token colour at the tables queue. A connection between any two queues is represented using the two queues as well as the token colour that uses the connection. The order a transaction takes in accessing tables is then encoded into connections between the queues corresponding to the tables. This results in there existing a path from the first table the transaction accesses to the last with each connection marked as belonging to that transaction token colour.

The first table a transaction accesses has a connection added from a source node to the first table queue and in a similar fashion there is a sink node added for all paths ends the execution path of each transaction. IF statements are initially handled by inserting virtual nodes that then fork to both the first table in the IF and the first table after the IF, with the connections having probability p and $1 - p$ respectively where p is the probability of entering the if condition. In a further pass these virtual nodes are then removed by merging matching incoming and outgoing connections for each virtual node resulting in a queueing network representing the structure of the transaction traffic. Figure 5.7 depicts the translation to a QN for the specification in Figure 5.2, the connections are labelled with the probability of being chosen. At this point in the translation the traffic proportions are not yet incorporated into the model.

5.4.4 Queueing Petri net construction

This stage of the translation takes in the QN model from the previous stage as well as the traffic proportions to create an internal representation of a QPN that models the traffic specification. This section of the code defines the concurrency control mechanism that is being modelled. To extend the tool to allow modelling with another form of concurrency control would only require an implementation for this stage that creates the correct QPN structure. The algorithm that translates the

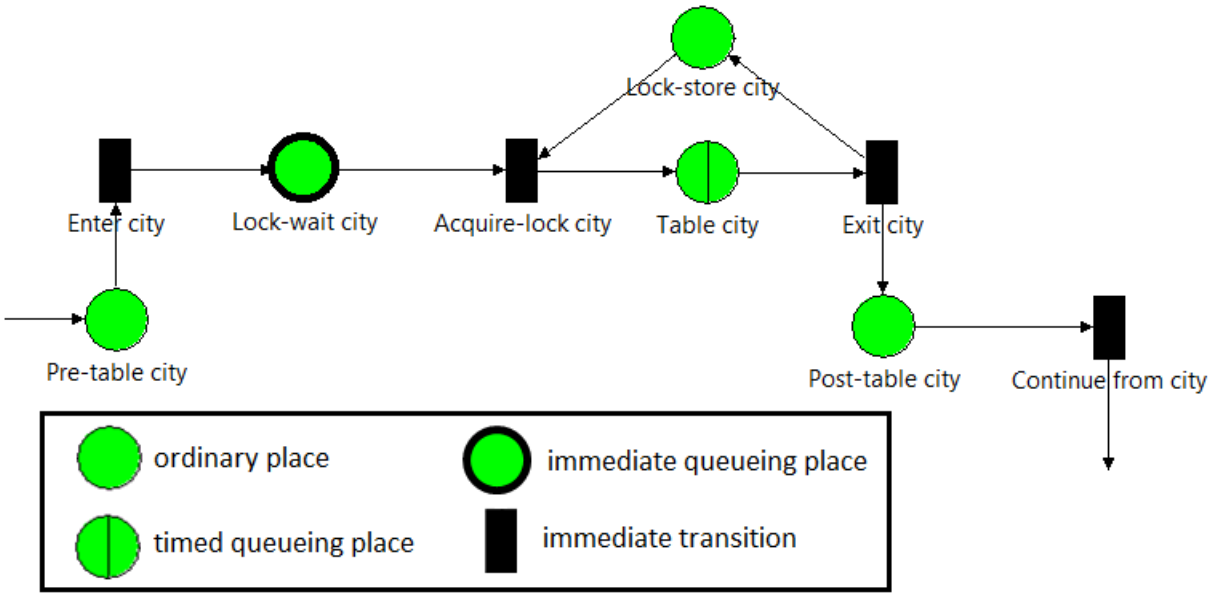


Figure 5.8: The subnet generated for table city

QN into a QPN functions by translating each queue (representing a database table) into a subnet of a QPN such that the subnet represents the table and the locking mechanism for the table. Once the subnets are created it is a case of connecting the subnets together in the same configuration as the QN and replacing the source and sink nodes with a client subnet that controls client think time as well as generating transactions in the right proportions. The current implementation generates models of table-level non-transactional locking.

Each table is translated to contain the following components:

- *Pre-table place* - This is the entry point for the subnet all transaction tokens that need to access the table will come to this point.
- *Enter table transition* - This is an immediate transition that moves transaction tokens from the pre-table place to waiting to acquire the lock in the lock-wait place.
- *Lock-wait place* - This is an immediate queueing place with FIFO departures that enforces transaction tokens to acquire the lock in the order that they arrive.
- *Lock-store place* - This is an ordinary place that holds the lock tokens currently available for this table. The number of lock tokens originally in the place will be equal to the total number of clients in the closed system.
- *Table place* - This is a timed queueing place that contains the corresponding table queue from the QN stage, therefore it contains an infinite server queue that services tokens for transactions that need access to the table. Once a transaction token has completed its service demand here it will be placed in the depository so that it can leave the table.

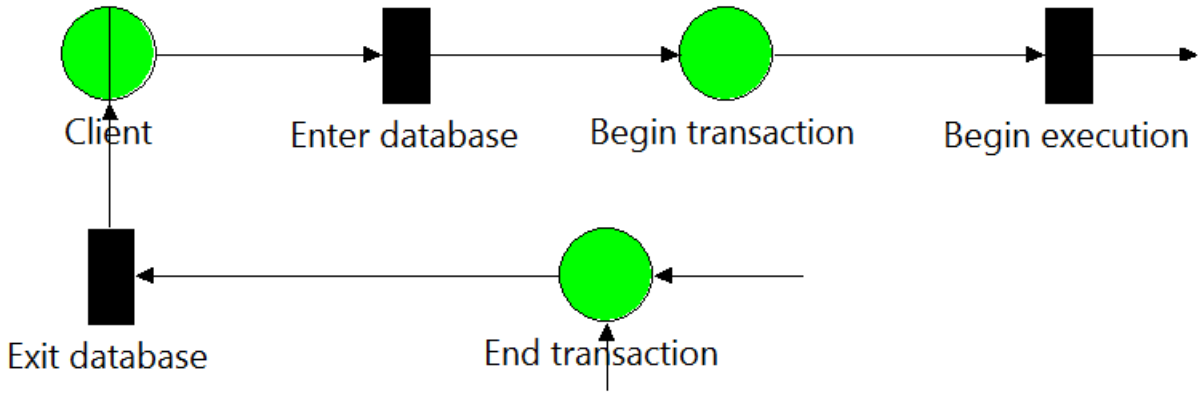


Figure 5.9: The client subnet that is generated for every QPN model

- *Acquire-lock transition* - This is an immediate transition that requires lock tokens from the lock store to fire and will deposit transaction tokens into the queue section of the table place. If the transaction requires exclusive access to the table then it will require the maximum number of lock tokens from the lock-store place whereas if it requires shared access then it will only require one token.

- *Exit table transition* - This is an immediate transition that will take transaction tokens from the table place to the post-table place as well as returning any lock tokens that the transaction used to the lock-store. That is one for a shared access and the maximum number for an exclusive access.

- *Post-table place* - This is an ordinary place that all transaction tokens that access the table will pass into to leave.

- *Continue from transition* - This is an immediate transition that will transfer the token to the next subnet that the token needs to visit. This transition will have multiple modes of execution if the transaction splits due to an if statement.

The source and sink nodes are translated to a single client subnet containing the following components:

- *Client place* - This is a infinite server queueing place that holds and services client tokens. Clients tokens can require either an exponentially distributed think-time or a deterministic think-time.

- *Enter database transition* - This is an immediate transition that has multiple modes of firing. For each transaction type in the system it has a firing mode with a firing weight equal to the proportion the transaction makes up of the total traffic. This connects the client place and begin transaction place such that client tokens are removed from the client depository and transaction tokens are added to begin transaction.

- *Begin transaction place* - This is an ordinary place that acts as a buffer between the enter database transition and begin execution transition.

- *Begin execution transition* - This is an immediate transition that removes transaction tokens from the begin transaction place and deposits them in the transactions first table access. If the first table access could be one of multiple places due to if statements then there are multiple firing modes in competition, each weighted according to the probability of the transaction reaching that table first.

- *End transaction place* - This is an ordinary place that all transaction tokens will be deposited in when the transaction is complete.

- *Exit database transition* - This is an immediate transition that removes transaction tokens from the end transaction placed and deposits client tokens back in the queue section of the client place, allowing the client to repeat.

Once the table queues have been translated to subnets and the client subnet has been created the QN connections are reproduced. This is done between two tables by connecting the continue from transition of the source table to the pre-table place of the destination table. Connecting to the old source and sink nodes is similarly done except both nodes are replaced with the client subnet and the begin execution transition is used when the client is the source of the connection while the end transaction place is used when the client is the destination of the connection. This completes the translation from specification to a QPN model, the last stage is to output the QPN in a format acceptable for QPME 2.0.

5.4.5 QPME XML translator

The output of the QPME 2.0 QPN editor is an XML definition of the QPN model as well as the simulation configuration to be used with the model in SimQPN (QPMEs QPN simulator). Therefore the internal representation of a QPN needs to be translated to appropriate XML to be compatible with QPME. Allowing QPME to be used view the QPN models and SimQPN to be used to simulate them. This involves traversing the QPN model, meaning looping through the token colours, transitions and places translating each component to an XML element following the QPME format.

To be able to calculate response times from a QPME simulation of the model, a probe needs to be added to the model. A probe is a QPME concept that measures token activity during simulation between the two points specified for the probe. Therefore to calculate response time the probe should be placed between on entry to the begin transaction place and on exit from the end transaction place. This enables it to measure the response time of each transaction type as all transaction tokens will pass between those points.

The translation encodes a default simulation configuration to allow QPME to simulate it straight from the output. In a SimQPN translation for each place or probe a stats level can be defined which dictates how much data is recorded for that component. The stats level is a QPME concept relating to what data is collected during simulations and the statistics to provide at the end of the run. A summary of the levels are as follows [37]:

- **Level 1** : At this level only token throughput data is recorded resulting in token arrival and departure rates being estimated for each colour at each place.

- **Level 2** : This level provides the data collection from level 1 with additionally token population and utilization data. Allowing statistics such as average number of tokens in a place to be provided.

- **Level 3**: Records token residence times at each place. This allows calculation of statistics such as estimates of steady state mean token residence times.

- **Level 4**: Provides all of the above and dumps observed token residence times into a file.

To reduce the simulation times for generated QPN models the default stats level for the response time probe is marked as 4 meaning as much data as possible is recorded and for all other components the stats level is set to 1 meaning minimal data is recorded. This however can be changed by the user either in the tool settings or by creating a simulation configuration themselves and running SimQPN manually.

5.4.6 QPME runner

This part of the software is written for user convenience as the QPN has already been generated at the end of the XML translator stage which can then be modified and simulated with QPME 2.0. This part of the tool allows the user to define a range of client amounts to run over and uses the QPN template file generated by the QPME XML translator and creates a new QPME QPN model for each client amount. This involves reading the template file and parameterizing it with the specific number of clients which impacts the number of lock tokens in each lock repository and the number of client tokens in the client place.

For each of the QPN models, SimQPN (the QPME QPN simulator) is invoked remotely to simulate the model. Unfortunately even though QPME is written in Java it currently provides no way of neatly interacting with the QPME simulator code but instead the script file that starts the QPME simulator has to be run from within Java. This is not ideal but as QPME is developed we are hopeful that they will add an API which would make interacting with it far easier. To speed up the process of simulation, AutoQPNPED allows the user to specify a number of threads to use to execute the simulation job. As QPME simulations are not innately parallel, each thread will invoke a single simulation such that there will be as many concurrent simulations as the number of user specified threads.

Once all the threads have completed the simulation invocation, AutoQPNPED reads the output files generated by the simulations and reads the mean response times of each transaction type. These mean response times are then combined to create a data file that describes the change in response time as the number of clients changes. More detailed information about the simulation runs can be found by exploring the simulation output in QPME itself.

5.5 Limitations and extensibility

AutoQPNPED currently supports only PostgreSQL and only produces QPNs that model table-level non-transactional locking. To be useable in industry the tool ideally would support all DBMSs and be capable of modelling a variety of locking setups. As we did not have time to implement these

additional features we instead wrote AutoQPNPED to be easily extensible so the features could be added in the future. We identified four key areas that extension was important in:

- **Specification input** - Currently the tool defines a simple domain specific language to enable definition of transaction traffic. To be able to interface with other systems and other potential input sources the input method must be decoupled from the rest of AutoQPNPED's functionality. We achieve this by creating an internal representation of a transaction workload that functions as an intermediary format. This allows new input sources to be used as long as the input is translated to the internal transaction workload representation.

- **DBMS supported** - Allow for more DBMSs to be supported than just PostgreSQL. We achieved this by reducing DBMS dependency to a single module, *Query atomization and annotation*. Therefore further DBMSs can be supported by writing an implementation just for the one module.

- **QPN translation** - Allow for a variety of locking mechanisms to be modelled. To allow for easier adding of locking mechanisms we separated the Queueing Network construction from the Queueing Petri net construction. This means that to implement another locking mechanism only the translation of each table to its locking mechanism needs to be defined.

- **Output format** - The tool outputs the QPN in a format that is compatible with QPME 2.0 however other tools that simulate or analyse QPNs may be developed and the tool must be able to adapt and support them as needed. We approach this is a similar method to the input, that is by using an internal representation we can decouple the output format from the rest of the working of AutoQPNPED.

Chapter 6

Evaluation of AutoQPNPED

6.1 Modelling pgbench

6.1.1 Pgbench overview

Pgbench is a built-in benchmarking tool for PostgreSQL [33] that involves generating a simple database structure and with clients that repeatedly execute a specified transaction. The aim being to measure the transactions per second executed as well as the mean response time per transaction. Pgbench generates a database that is based upon a simple banking system and it contains four tables: *pgbench_accounts*, *pgbench_branches*, *pgbench_tellers* and *pgbench_history*. The structure of the tables is displayed in Figure 6.1 however by default the foreign key relationships are not enforced.

The standard query that pgbench executes is based upon a TPC-B like scenario where random records in each of *pgbench_accounts*, *pgbench_branches* and *pgbench_tellers* have their balances updated by a random delta value. Each change then produces a history entry that is inserted into *pgbench_history*. The records are retrieved using an index scan as each record is selected via its indexed primary key. The default script used by pgbench is displayed in Figure 6.2 where scale is the scale factor used to generate the pgbench tables which dictates the number of records in each table. The number of records in *pgbench_branches* is equal to the scale factor and for each record in *pgbench_branches* there are 10 records in *pgbench_tellers* and 100,000 records in *pgbench_accounts* while *pgbench_history* has initially no records.

The scale factor allows generating databases with different performance properties. For small scale factors the database tables will be able to fit in the memory in the allocation for postgreSQL as well as in the OS and disk caches. In this situation the performance bottleneck definitely not physical disk as there should be minimal disk interaction with the tables in memory and is potentially quite likely to be contention for a logical resource such as locks. When the scale factor is increased the disk will become a much larger factor as the tables can not longer fit entirely in memory resulting in disk accesses. This dynamic makes it a useful benchmark to measure the limitations of AutoQPNPED.

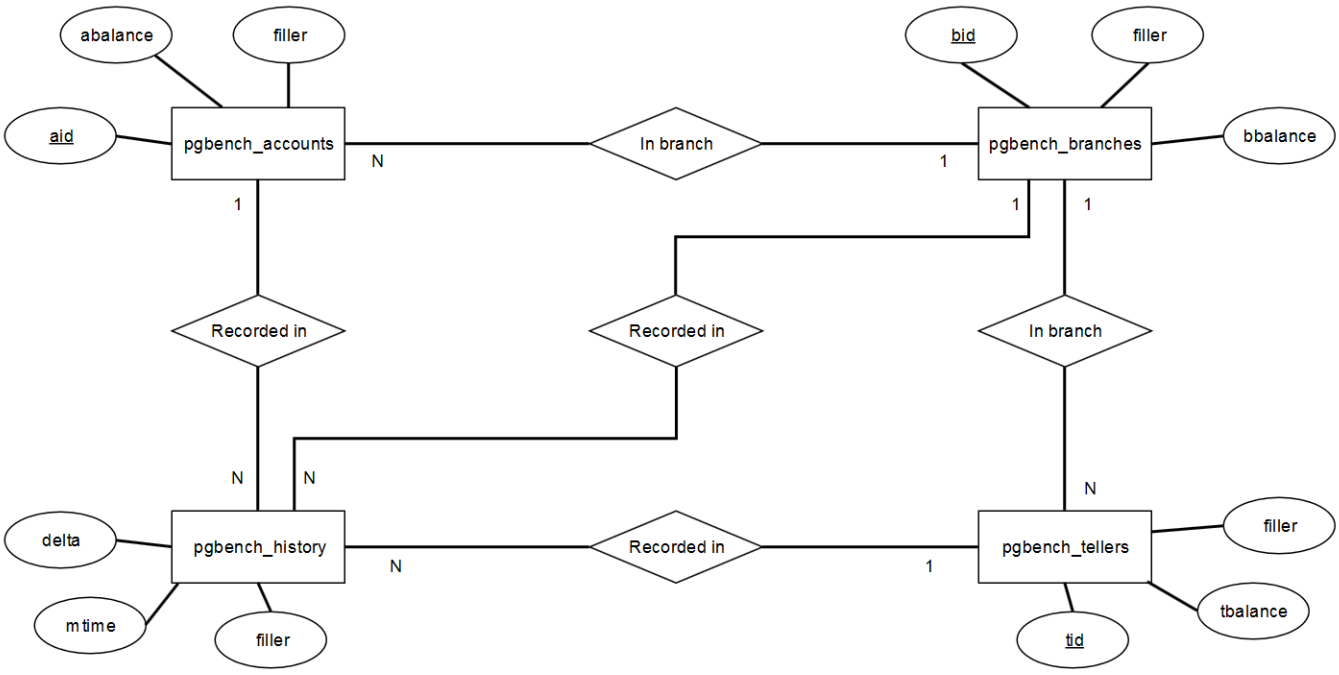


Figure 6.1: ER diagram of the pgbench tables

```
\set nbranches :scale
\set ntellers 10 * :scale
\set naccounts 100000 * :scale
\setrandom aid 1 :naccounts
\setrandom bid 1 :nbranches
\setrandom tid 1 :ntellers
\setrandom delta -5000 5000
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
    VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
END;
```

Figure 6.2: The default pgbench transaction

6.1.2 Adapting pgbench

The default pgbench transaction needs to be adapted to be modelled using AutoQPNPED in the following ways:

- By default postgreSQL will use multiversion concurrency control as its concurrency control mechanism however AutoQPNPED models non-transactional table-level locking. To change pgbench to use this it requires explicit acquisition of locks as in Chapter 4 therefore reads will acquire table locks in ACCESS SHARE mode and updates will acquire in ACCESS EXCLUSIVE mode so that the reads and writes will conflict. To model the non-transactional locking each statement will be encased in its own transaction as locks are released when a transaction ends.

- The default script for pgbench performs both a read and a update to the *pgbench_accounts* which is a multiple access to one table that is not supported by AutoQPNPED amd the QPNPED methodology. The update was removed to resolve this issue as it leaves the transaction containing a mix of shared and exclusive statements meaning more of the functionality of QPNPED is tested.

The resulting script used with our runs of pgbench is shown in Figure 6.3.

6.1.3 Pgbench traffic specification

To use AutoQPNPED a traffic specification needs to be written describing the transaction traffic in a pgbench run. Since pgbench involves repeated execution by all clients of the same query there is only one transaction type which makes up the whole transaction traffic composition. The transaction run in pgbench uses random values that are not supported in queries by AutoQPNPED so they are replaced with queries that access a single named record and the insert is changed to not insert the current timestamp into the *pgbench_history* table.

The resulting traffic specification is shown in Figure 6.4. The values chosen were semi-arbitrary as the only requirement is that the record exists no matter the scaling factor and if the record exists performance should be approximately equal for each record.

6.1.4 Experimental approach

The aim of the experiment is to show the correctness of the tool in both inferring details from a transaction specification as well as exploring the performance evaluation capabilities of the QPN models generated by AutoQPNPED in terms of both its successes and limitations. We do this by generating a variety of pgbench databases of different sizes by initialising them using different scaling factors with the aim to cover the region where the database no longer fits in available memory. For each database the AutoQPNPED tool is run on the specification in Figure 6.4 which generating a series of QPNs that vary in number of clients which are solved via simulation using QPME 2.0 to produce estimated query response times. We then compare these response times to those found from pgbench runs on the same databases while varying clients.

The experiments were performed on a Intel(R) Core(TM) i7-2600 CPU@3.40GHz box running Ubuntu 12.10 64-bit and PostgreSQL 9.1 with 8GB of RAM. The PostgreSQL setup was only

```
\set nbranches :scale
\set ntellers 10 * :scale
\set naccounts 100000 * :scale
\setrandom aid 1 :naccounts
\setrandom bid 1 :nbranches
\setrandom tid 1 :ntellers
\setrandom delta -5000 5000

BEGIN;
LOCK TABLE pgbench_accounts IN ACCESS SHARE MODE;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
END;
BEGIN;
LOCK TABLE pgbench_tellers IN ACCESS EXCLUSIVE MODE;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
END;
BEGIN;
LOCK TABLE pgbench_branches IN ACCESS EXCLUSIVE MODE;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
END;
BEGIN;
LOCK TABLE pgbench_history IN ACCESS EXCLUSIVE MODE;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
    VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
END;
```

Figure 6.3: The script used in runs of pgbench

```
transaction pgbenchtrans 1.0 {
    -SELECT abalance FROM pgbench_accounts WHERE aid = 5;
    -UPDATE pgbench_tellers SET tbalance = tbalance + 1 WHERE tid = 10;
    -UPDATE pgbench_branches SET bbalance = bbalance + 1 WHERE bid = 1;
    -INSERT INTO pgbench_history (tid, bid, aid, delta) VALUES (10, 1, 5, 1);
}
```

Figure 6.4: The traffic specification file representing the pgbench transaction traffic

```
transaction pgbenchtrans 1.0 {
  statement exclusive=false runtime=0.253441 table=pgbench_accounts {
    -SELECT abalance FROM pgbench_accounts WHERE aid = 5;
  }
  statement exclusive=true runtime=0.473862 table=pgbench_tellers {
    -UPDATE pgbench_tellers SET tbalance = tbalance + 1 WHERE tid = 10;
  }
  statement exclusive=true runtime=0.495037 table=pgbench_branches {
    -UPDATE pgbench_branches SET bbalance = bbalance + 1 WHERE bid = 1;
  }
  statement exclusive=true runtime=0.441471 table=pgbench_history {
    -INSERT INTO pgbench_history (tid, bid, aid, delta) VALUES (10, 1, 5, 1);
  }
}
```

Figure 6.5: Annotated pgbench transaction specification for the scale factor 150 database

modified by increasing the shared buffer size to 250MB to ensure that database tables of a reasonable scale can fit in the shared memory and the number of checkpoint segments was increased to 30 which reduces the number of checkpoints that occur. This results in a system that is more focused on performance than endurance and runs should not have their performance impacted by an unexpected checkpoint.

6.1.5 Running AutoQPNPED

A run of AutoQPNPED on the specification in Figure 6.4 produces an annotated version of the specification which shows the inferences that AutoQPNPED has made. The result of this process is shown in Figure 6.5. The file shows the tool has successfully inferred the table that each of the statements accesses and has determined that the SELECT statement is not exclusive whereas the rest of the statements all are exclusive. Each of the statements has also been annotated with its isolated mean execution time that will go on to be used as the service demands for the transaction at each table.

The annotated specification then goes on to be transformed into a QPN in the QPME 2.0 format. The structure of this QPN is shown in Figure 6.6 and it can be seen that the tables are accessed in the same order as in the pgbench transaction and it all creates a closed loop where the clients being in the client place. Since there is no exponentially distributed sleep time built into the pgbench benchmark the AutoQPNPED tool was configured to use a deterministic distribution of amount zero, in other words the clients do not need to be serviced in the client place.

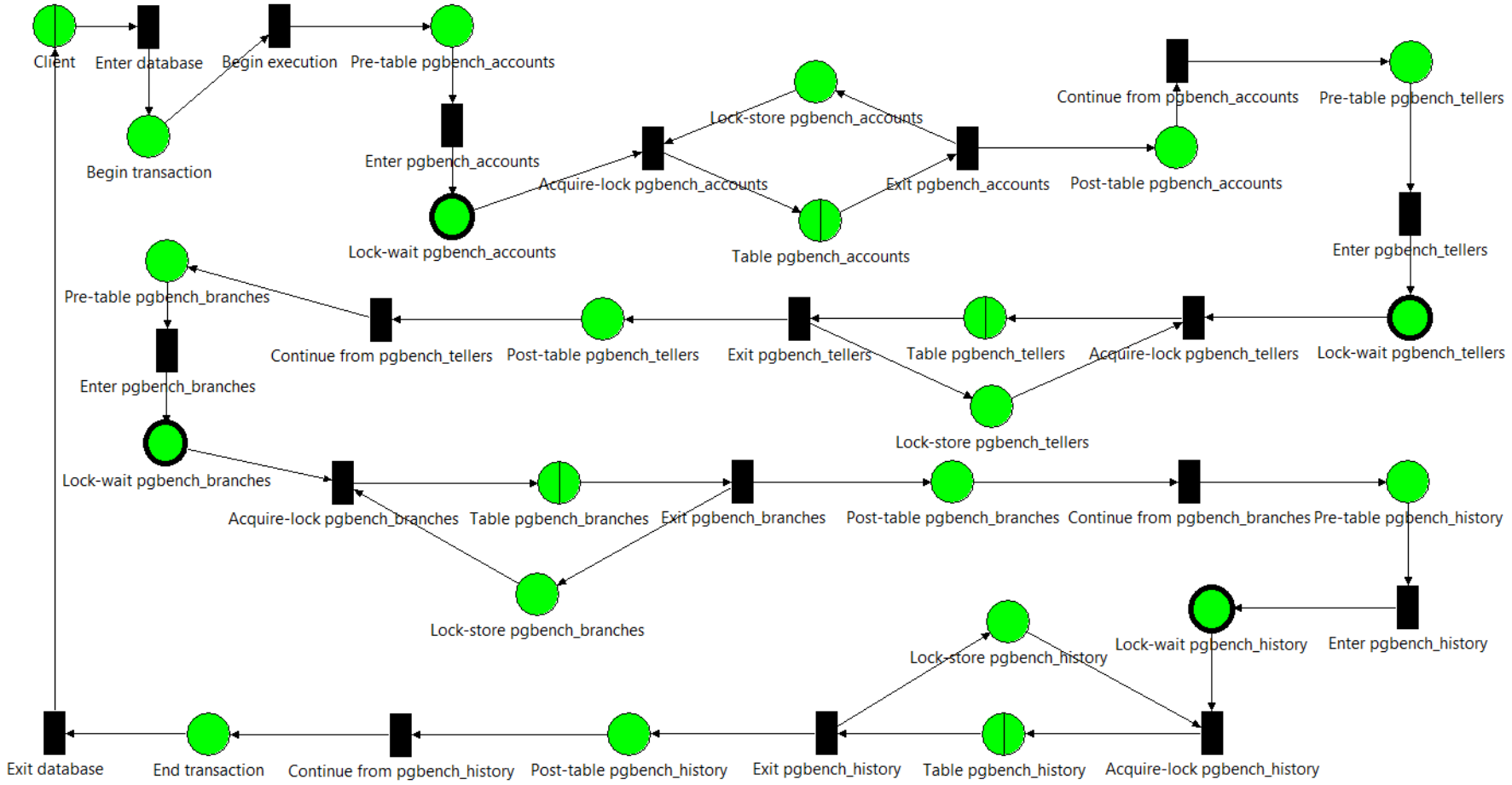


Figure 6.6: The QPN generated by AutoQPNPED for pgbench

Table 6.1: Table giving the isolated query execution times for each of the tables over a range of scale factors, with times in milliseconds.

Scale factor	pgbench_accounts	pgbench_tellers	pgbench_branches	pgbench_history
150	0.253	0.474	0.495	0.441
175	0.253	0.476	0.502	0.453
200	0.256	0.528	0.534	0.503
225	0.268	0.535	0.543	0.513
250	0.316	0.543	0.545	0.533
275	0.337	0.553	0.563	0.543
300	0.345	0.609	0.609	0.587
350	0.356	0.639	0.646	0.589

6.1.6 Experimental results

Initially we ran pgbench and generated QPNs for scale factors between 10 and 200 as we believed that since we left postgreSQL with only 250MB of shared buffers that interesting performance would be seen across this range since by 100 scaling factor the database size has reached 1456MB. However we underestimated how much caching the disk performs as well as the file caching by the OS that there was negligible difference in performance of the measured system between scale factors 10 and 150. The performance did however drop considerably when the scale factor reached 200 which indicated that this was the most interesting area to examine and as such runs covering the range from 150 to 350 were performed.

The isolated execution times for each query and each database scale factor are given in Table 6.1 and they can be seen to increase slightly as the scale factor is increased. This small increase in query execution time is likely due to larger indexes meaning the index will take longer to traverse to find required records.

Each run of pgbench was repeated three times to increase the reliability of the results. The QPN model was simulated using the method of non-overlapping batch means to estimate steady state token residence times with 95% confidence intervals. For all the simulations the confidence intervals were sufficiently small for the results to be reliable. The mean response times over the repeats are given in Table 6.2 along with the mean response times found via simulation of the QPN.

The measured system shows that the mean response time of the query increases at a slow rate between 150 to 200 scale factor and this is due to the database tables residing almost entirely in main memory meaning very few expensive disk operations are occurring. When the scale increases to 225 there is a considerable performance decrease that gets worse the more clients that are used. This is likely caused by the database tables no longer fitting completely in memory and as such there is more fetching from disk and swapping of database pages into memory and this means that queries will take longer. Longer queries means more lock contention especially so at high numbers of clients. This trend continues as the scaling factor increases such that by 350 scaling factor the average response time for a query with 60 clients is 443.86ms whereas at 150 scaling factor is was only 31.72ms.

Table 6.2: Results table giving mean response times in the measured system and estimated response times using the QPN model, times given in milliseconds

Scale factor	Number of clients											
	10		20		30		40		50		60	
	Measured	QPN	Measured	QPN	Measured	QPN	Measured	QPN	Measured	QPN	Measured	QPN
150	5.42	5.72	10.56	10.45	15.36	15.23	20.81	20.08	26.76	25.01	31.72	29.97
175	5.64	5.79	11.34	10.61	17.56	15.50	23.21	20.29	29.03	25.24	35.72	30.23
200	6.03	6.34	11.87	11.54	18.15	16.80	24.22	21.97	30.67	27.29	41.88	32.65
225	6.81	6.43	28.63	11.71	31.20	17.01	35.68	22.41	53.82	27.77	89.40	33.09
250	13.42	6.56	37.55	11.90	59.59	17.32	125.40	22.72	126.23	28.15	185.73	33.52
275	34.80	6.71	82.29	12.19	159.56	17.77	198.58	23.25	207.36	28.87	259.64	34.42
300	51.32	7.30	101.52	13.28	135.79	19.25	193.27	25.21	259.48	31.49	320.13	37.38
350	66.45	7.59	136.54	13.87	213.45	20.21	290.38	26.62	360.66	32.95	433.86	39.41

The QPN model estimates the performance of measured system most accurately at 150 scaling factor where it overestimates the performance of the measured system in general with the overestimate increasing with the number of clients. The overestimation with larger numbers of clients is likely due to increased locking overhead due to a large number of competing transactions as well as increased cost of committing the transactions caused by a large number of updates needing to be managed to give a consistent view to all transactions. As the scaling factor is increased the accuracy of the QPN decreases, slowly at first over scaling factors 175 and 200, and then dramatically from there. This is caused by a smaller percentage of the database tables being able to be kept in memory causing disk reads which are considerably slower than the isolated query execution times that are used as service demands in the QPN model. This effect gets amplified by the high amount of contention at large numbers of clients resulting in far higher lock waiting times and very bad performance that is not mirrored in the QPN model as the large increase in query execution time is not modelled. The isolated query execution time at high scaling factors becomes a bad estimation of query execution time since it is calculated from repeatedly executing the query and as the query will only select a single record this record is almost guaranteed to remain in memory so the isolated query execution time will only involve memory accesses. The degradation of the performance of the measured system and the loss of accuracy of the QPN model is shown in Figure 6.7.

At high scaling factor the disk becomes the bottleneck for performance and although lock waiting times will be high the contention is not the primary cause of the poor query execution performance. Whereas at low scaling factor the lock contention is the bottleneck in performance and as such the QPN model more accurately predicts performance. This shows that the AutoQPNPED tool and the QPNPED methodology in general performs well when there is lock contention and isolated query execution times accurately represent the query execution time in the multiclient situation, which agrees with the results in Chapter 4. However when disk becomes the bottleneck for performance the QPN model created by AutoQPNPED is very limited in its capability of estimating performance of the system as the isolated query execution times are no longer representative of the actual query executions.

6.2 Modelling a JOIN case

We now extend the reader-writer problem, from Chapter 4, such that the shared transaction performs a join between two tables with the aim of evaluating the effectiveness of the join approximation performed by AutoQPNPED. We again measure and model the performance, in the form of mean response times, of two transaction types: shared and exclusive.

6.2.1 Measured system

To measure the actual performance of the shared and exclusive queries we adapted the C++ benchmark that we used in Chapter 4. The key functionality we required from the benchmark was the same, concurrent running of transactions. However we needed to change the client system from fixed clients,clients executing the same transaction repeatedly, to choice clients where each client randomly chooses the transaction it will perform at each cycle with the probability of choosing a transaction type equal to the transactions traffic proportion. This change was required as this was the client modelling method employed by AutoQPNPED. Clients had an exponentially distributed think time with mean 500ms.

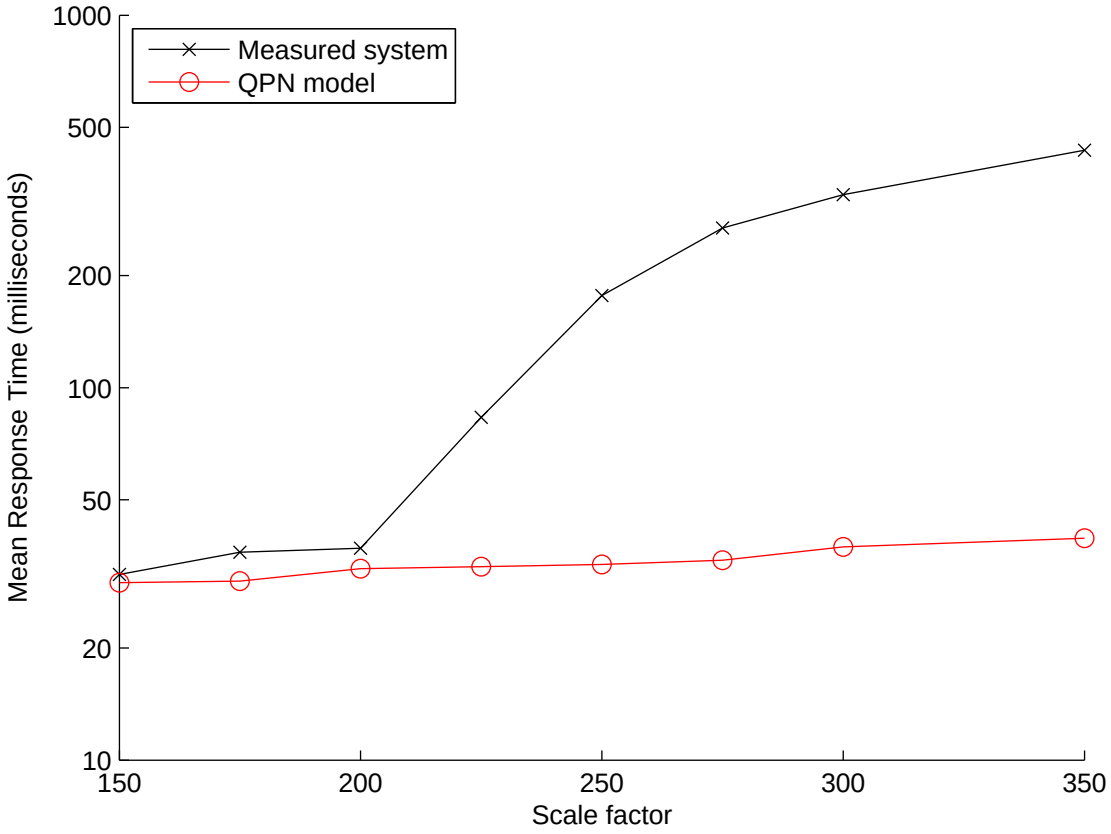


Figure 6.7: Results for 60 clients over a selection of scale factors

Shared transaction

```
BEGIN;
    SELECT table_a.fixedstring, table_b.fixedstring
    FROM table_a JOIN table_b ON table_a.id = table_b.fraction;
END;
```

Exclusive transaction

```
BEGIN;
    LOCK TABLE table_b in ACCESS EXCLUSIVE MODE;
    UPDATE table_b SET fixedstring='text' WHERE fraction > 99;
    SELECT pg_sleep(0.1);
END;
```

Figure 6.8: Structure of shared and exclusive transactions

The measured database contained two tables: table_a and table_b, with 5,000 and 10,000 records respectively. Each record is randomly generated and contains a sequential identifier named id which is the primary key as well as a fixed string field of size 20 to increase record size and an integer field called fraction that is uniformly distributed between 1 and 100. The reason the databases were kept with relatively few records was due to the lack of a dedicated postgreSQL server machine that could support many clients performing a JOIN operation on larger tables concurrently.

The transactions in Figure 6.8 show are the shared and exclusive transactions used. The share transaction performs a JOIN operation of table_a and table_b, requiring a sequential scan of each table and the exclusive transaction updates approximately 1% of the records in table_b requiring a sequential scan of table_b.The share transaction does not explicitly request locks for either table_a or table_b because there is no place the lock statement can be placed while maintaining non-transactional table-level locking. Instead the exclusive transaction acquires a ACCESS EXCLUSIVE mode lock on table_b which will conflict when the shared transaction attempts to access table_b as ACCESS EXCLUSIVE ensures that the holding transaction is the only transaction with access to the table. As there is no contention for table_a it is not necessary to acquire a lock for it. The exclusive transaction is also artificially lengthened by 100ms to better model a TPC-W like workload where update transactions are longer than read transactions [28].

The benchmark was run on Intel(R) Core(TM) i7-2600 CPU@3.40GHz box running Ubuntu 12.10 64-bit and PostgreSQL 9.1 with 8GB of RAM. The PostgreSQL server setup was modified to make checkpoints more infrequent (by increasing checkpoint buffers to 30,000) and to avoid synchronizing with disk (fsync off). This was done to avoid expensive CHECKPOINT operations and costly write backs from impacting results unevenly.

6.2.2 Running AutoQPNPED

Creating the transaction traffic specification that represents the measured system was done in two stages. This was to accommodate the sleep statement in the exclusive transaction as this statement is not natively supported by AutoQPNPED. The first stage involved creating the specification of

```
transaction share_trans 0.95 {
    -SELECT table_a.fixedstring, table_b.fixedstring
     FROM table_a JOIN table_b ON table_a.id = table_b.fraction;
}
transaction exclusive_trans 0.05 {
    -UPDATE table_b SET fixedString = 'text' WHERE fraction > 99;
}
```

Figure 6.9: The initial specification for the join readers-writers with browsing traffic (95% share, 5% exclusive)

```
                              QUERY PLAN
----------------------------------------------------------------------
 Hash Join  (cost=144.50..1297.40 rows=31892 width=27)
   Hash Cond: (table_b.fraction = table_a.id)
   ->  Seq Scan on table_b  (cost=0.00..554.92 rows=31892 width=25)
   ->  Hash  (cost=82.00..82.00 rows=5000 width=10)
         Buckets: 1024  Batches: 1  Memory Usage: 215kB
         ->  Seq Scan on table_a  (cost=0.00..82.00 rows=5000 width=10)
```

Figure 6.10: Query plan for the join statement in the shared transaction

the transaction traffic without the sleep statement, shown in Figure 6.9 and running AutoQPNPED up to the atomization and annotation stage, with it outputting the annotated specification at that point. The second stage was then to increase the calculated isolated execution time of the exclusive transactions update statement by 100ms. This has the desired effect of modelling the sleep in the measured system as well as showing how the user can override the base inferences of AutoQPNPED to give them more flexibility.

The share transaction is split into two approximating SQL statements in the atomization stage of AutoQPNPED. This is done by finding the query plan for the statement using PostgreSQL EXPLAIN ANALYZE. The plan for this statement is a hash join, shown in Figure 6.10, where a sequential scan is performed on table_a and the records undergo hashing to create entries in a hash table. A sequential scan is then performed on table_b and the records are checked against the hash table to generate the joined records. AutoQPNPED will approximate this as two SELECT statements that perform sequential scans, one selecting from table_a and the other from table_b. The atomized statements are then annotated using the AutoQPNPED techniques and the 100ms sleep is added to the exclusive statement to give the specification in Figure 6.11.

AutoQPNPED was configured to model clients to have an exponentially distributed think time with mean 500ms to match the measured system. It was then run using the annotated specification in Figure 6.11 to produce a QPN model for the measured system.

```
transaction share_trans 0.95 {
    statement exclusive=false runtime=3.1442 table=table_a {
        -SELECT table_a.fixedstring, table_a.id FROM table_a;
    }
    statement exclusive=false runtime=7.1948 table=table_b {
        -SELECT table_b.fixedstring, table_b.id, table_b.fraction FROM table_b;
    }
}
transaction exclusive_trans 0.05 {
    statement exclusive=true runtime=103.0075 table=table_b {
        -UPDATE table_b SET fixedstring = 'text' WHERE fraction > 99;
    }
}
```

Figure 6.11: Atomized and annotated specification including 100ms sleep on the exclusive transaction statement with browsing traffic (95% share, 5% exclusive)

6.2.3 Experimental results

The experiment was run over the same three transaction workloads from Chapter 4: browsing (95% share, 5% exclusive), shopping (80% share, 20% exclusive), ordering (50% share, 50% exclusive). These are workloads from the e-commerce benchmark TPC-W [28] and were chosen as they give different contention behaviours. With browsing expected to show the least contention and ordering the most. Each workload is translated into an AutoQPNPED specification that differ from each other only in transaction proportions.

The benchmark was run for 180 seconds per trial and each trial was repeated five times to improve the reliability of results. The QPN model was simulated using the method of non-overlapping batch means method (with default QPME settings) to estimate steady state mean token residence times with 95% confidence intervals. For all the simulations the confidence intervals were sufficiently small for the results to be reliable.

The results for shared transactions are given in Table 6.3 and for the exclusive transactions in Table 6.4. The measured system shows very similar performance patterns to the original formulation of the reader-writer problem in Chapter 4. Under browsing traffic the shared transaction performance reduces as the number of clients increase but not as much as the performance of exclusive transactions is affected. This is due to the increased contention for the lock on table_b that increases wait time for both transaction types. Shared transactions make up the majority of traffic and can simultaneously access table_b so they do not suffer as long wait times as the exclusives. As the traffic proportion involves more exclusives the performance of both transaction types degrades considerably and tends towards the same value. This is because the lock wait time becomes the dominant part of query response time.

Table 6.3: Table of mean response times in milliseconds for shared transactions across all three workloads. Comparing the measured system to the QPN model.

Number of Clients	Browsing response time (ms)		Shopping response time (ms)		Ordering response time (ms)	
	Measured	QPN model	Measured	QPN model	Measured	QPN model
10	46.66	18.62	78.40	50.01	188.70	149.90
20	60.53	29.05	212.88	123.43	741.81	548.96
30	73.33	42.13	457.17	251.87	1386.33	1087.77
40	104.65	58.15	823.49	438.91	2084.57	1610.16
50	134.97	76.95	1100.28	628.95	2757.31	2132.63
60	171.44	98.36	1458.76	848.61	3335.62	2694.57

Table 6.4: Table of mean response times in milliseconds for exclusive transactions across all three workloads. Comparing the measured system to the QPN model.

Number of Clients	Browsing response time (ms)		Shopping response time (ms)		Ordering response time (ms)	
	Measured	QPN model	Measured	QPN model	Measured	QPN model
10	118.54	112.28	147.64	143.96	252.93	244.73
20	153.83	122.74	287.16	219.49	805.25	652.98
30	192.23	139.42	531.07	352.64	1451.98	1181.08
40	222.41	157.55	890.71	546.25	2149.79	1709.94
50	265.25	178.56	1171.99	731.25	2817.77	2208.81
60	306.59	202.48	1536.63	951.75	3402.30	2794.08

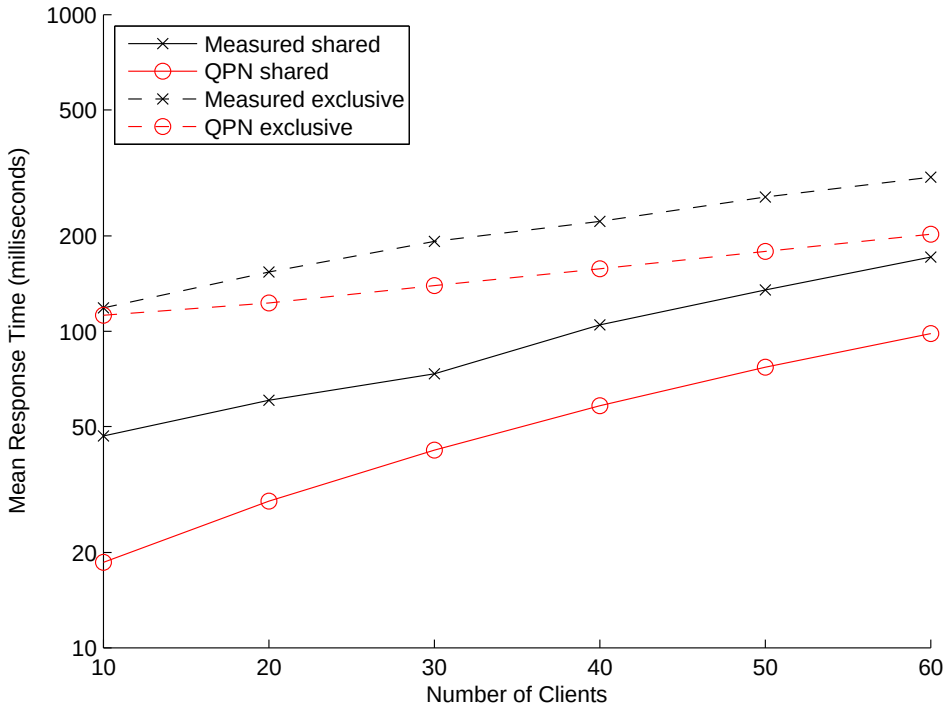


Figure 6.12: browsing (95% shared, 5% exclusive)

The QPN model universally underestimates the performance of both shared and exclusive transactions. The underestimation originates from the approximation of the JOIN in the shared transaction. The approximation does not take into account additional processing done by the JOIN, which in this case involves creating a hash table of table_a. This shows in the results as when under browsing traffic the shared transaction is underestimated even under low contention with error of 60% for 10 clients. The performance of exclusive transactions is better predicted with an underestimate of 5.2% under browsing traffic for 10 clients. As the number of clients is increased the underestimate becomes larger for both transaction types. This is due to the original under-approximation of the JOIN resulting in less contention in the QPN model than is actually happening in the measured system.

Under shopping traffic the QPN model follows a similar pattern to browsing as again the underestimate of both shared and exclusive transactions increases with number of clients. The QPN models prediction follows the same trend as the measured system. As performance prediction of the shared transactions and exclusive transactions also tends towards a similar value at higher client numbers. This indicates that the QPN model is capturing the contention dynamics exhibited in the measured system but due to the JOIN underestimation the response times are smaller.

Similar trends in underestimation can be seen with the ordering workload and again the QPN model follows the same trend as the measured system. The model underestimates the performance of shared and exclusive transactions by 19% and 17.9% respectively under 60 clients. In comparison the errors for the shopping workload were 41.8% for shared and 38% for exclusive. This shows a

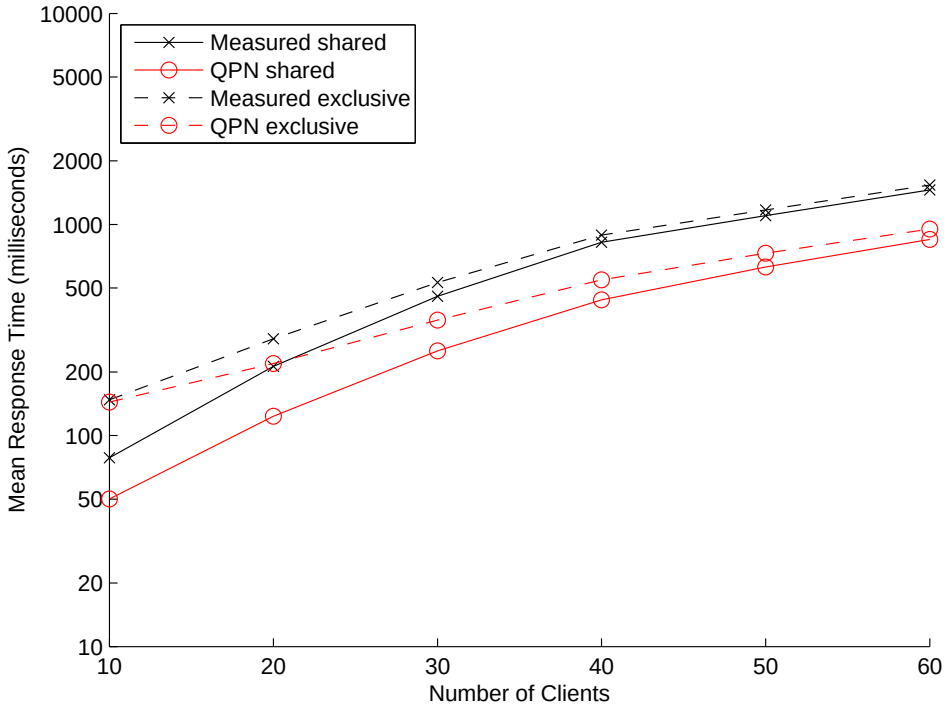


Figure 6.13: shopping (80% shared, 20% exclusive)

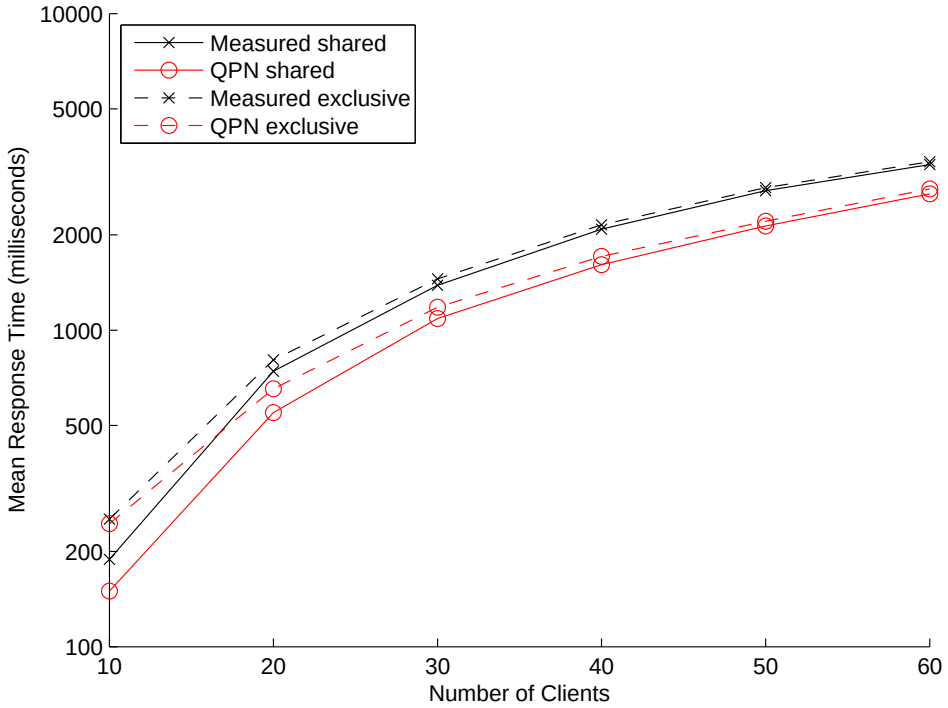


Figure 6.14: ordering (50% shared, 50% exclusive)

considerable improvement in the relative estimation of performance under ordering traffic. The is caused by the lock wait times dominating the response time of the transactions, so the JOIN underestimation makes up a far smaller proportion of query response time.

Overall the results show that the models generated by AutoQPNPED are less accurate when JOINs are involved, due to approximation of the JOIN. However the QPN model was still able to follow the trend of the measured system under each workload. This indicates that the QPN model generated is capturing the concurrency control dynamics that occur in the measured system.

Chapter 7

Conclusion

In this thesis we have presented a methodology, QPNPED, for evaluating the performance of relational database systems with a focus on modelling concurrency control contention. The methodology uses Queueing Petri nets, a modelling formalism that to the best of our knowledge has not been applied to the database performance evaluation field. The QPN models created by QPNPED correctly model the request and release locking mechanism as it is implemented in DBMSs. We have shown that the methodology creates QPN models capable of accurate prediction of performance when contention is high and under low contention the models still follow the trend of the measured database system. In addition the models generated require minimal parameterization and reflect the operational flow of the transaction traffic making them easy to understand.

We also presented a software tool, AutoQPNPED that automates the mapping from transaction traffic to QPN model defined by QPNPED. The software system was challenging to write in multiple aspects due to the considerations of flexibility and extensibility to make the tool easier to be used in an industrial environment. We showed that AutoQPNPED was functionally correct and was simple to specify when modelling the pgbench benchmark and a case study that involved JOIN statements. In modelling pgbench we investigated the limitations of the methodology when disk is the performance bottleneck of the system. We found the generated QPN was accurate in performance prediction when query executions did not interact heavily with the disk. As the amount of disk interaction increased the performance estimate became more inaccurate and eventually with very high levels of disk access the QPN model is not capable of following the trend. Our modelling of a case study that used JOIN statements showed that the QPN model exhibited the same locking behaviours as the measured system. This shows that despite the QPNPED methodology only approximating the service demand of a JOIN statement the model generated can be used to find locations of concurrency control bottlenecks.

Our work has shown that QPNPED and AutoQPNPED can be used to predict performance accurately for some real database systems. Particularly those where concurrency control is a bottleneck for database performance. Most of all our results show that Queueing Petri nets, a previously overlooked modelling formalism, can be applied successfully to database performance evaluation.

7.1 Future work

There are three key paths that could be considered for future work: extending the QPNPED methodology, extending AutoQPNPED or improving usability of AutoQPNPED.

7.1.1 Extending the QPNPED methodology

Service demand modelling Currently QPNPED uses isolated execution times as the service demands. However we have shown that the isolated execution times may not be representative of the real execution times, for example in the case where there is high disk usage. Alternative methods for calculating service demands could be investigated to improve the approximation for the problematic cases. A potential method could be calculating page reads for each query which is done by QuePED [24] however this particular example would require additional specification to be provided by the user.

Additional process modelling When executing a transaction or acquiring a lock there is always some overhead involved in the DBMS and currently this overhead is not considered by the QPNPED methodology. Incorporating these overheads into the methodology would help improve the accuracy but a method to measure these overheads would be required or a way of calculating approximations for the overheads.

Supporting further database features QPNPED supports transaction traffic that include *if* statements as well as *join* statements however there are many additional features that some DBMSs have that would be useful to support. Some of these include referential integrity, triggers and as well as more procedural constructs in transaction statements such as loops. All of these features are currently supported by QuePED and could be adapted for use in QPNPED.

7.1.2 Extending AutoQPNPED

Supporting more DBMSs The only DBMS that is supported by AutoQPNPED is PostgreSQL and for the tool to be useful to industry it must support as many of the mainstream DBMSs as possible. AutoQPNPED was designed with extensibility in mind so the task of supporting more DBMSs is relatively straightforward but necessary extension.

Supporting further locking mechanisms Currently AutoQPNPED generates QPNs that model non-transactional table-level locking and there is a large variety of locking mechanisms employed by modern DBMSs with some of them having been shown to be applicable for modelling using QPNs (see Appendix B.2). With more locking mechanisms available it could be possible to extend the tool to model different locking mechanisms for different tables giving the capability to model more complex database systems.

7.1.3 Usability of AutoQPNPED

Graphical user interface AutoQPNPED currently uses a command line interface to interact with the user, however in terms of usability a graphical user interface would be considerably better as it would give the user a better idea of the process going on and it would reduce the learning curve considerably.

Additional support for result processing Currently AutoQPNPED will generate QPNs in the QPME 2.0 format and will then send requests to QPME 2.0 to simulate the QPNs which generates result files in an XML format. This XML file format can be opened in QPME to provide statistics and graphs of the results for the user. But it does not allow comparison over multiple clients that would be useful feature for the user. AutoQPNPED by default parses the XML files and generates a data file with the overall response times for each transaction type over the range of clients the user specifies. Due to lack of time it currently does not parse any of the deeper detail in the results and does not draw any graphs. Extending AutoQPNPED with this functionality would not be a simple undertaking as there is a large amount of information to be represented in the simulation results files. This feature has the potential to improve the feedback the tool gives considerably instead of relying on the feedback from QPME.

Bibliography

[1] Olofson, C.W. Worldwide Relational Database Management Systems 2012-2016 Forecast. International Data Corporation, Doc # 236273. (2012) www.idc.com

[2] B. C. Jenq, B. C. Twichell, and T. W. Keller. Locking performance in a shared nothing parallel database machine. IEEE Transactions on knowledge and data engineering, 1989.

[3] Baskett, Forest, et al. "Open, closed, and mixed networks of queues with different classes of customers." Journal of the ACM 22.2 (1975): 248-260.

[4] Kendall, David G. "Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded Markov chain." The Annals of Mathematical Statistics (1953): 338-354.

[5] Murata, T. "Petri nets: Properties, analysis and application." Proceedings of the IEEE , vol.77, no.4, pp.541-580, Apr 1989.

[6] Petri, C.A. "Communication with Automata." New York:Griffiss Air Force Base. Tech.Rep.RADC-TR-65-377, vol. 1, Suppl. 1, 1966.

[7] Bause, Falker and Pieter S. Kritzinger. 'Stochastic Petri Nets." Vol. 1. Vieweg, 1996.

[8] Jensen, Kurt. "Coloured petri nets." Petri nets: central models and their properties (1987): 248-299.

[9] Ajmone Marsan, Marco, Gianni Conte, and Gianfranco Balbo. 'A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems." ACM Transactions on Computer Systems (TOCS) 2.2 (1984): 93-122.

[10] Molloy, Michael K. "Performance analysis using stochastic Petri nets." Computers, IEEE Transactions on 100.9 (1982): 913-917.

[11] Marsan, M. "Stochastic petri nets: an elementary introduction." Advances in Petri Nets 1989 (1990): 1-29.7.

[12] Bause, Falko. ""QN+ PN= QPN"-Combining Queueing Networks and Petri Nets." (1993).

[13] Ullman, Jeffrey D., and Jennifer Widom. First course in database systems. Prentice Hall Press, 2007.

[14] Codd, Edgar F. "A relational model of data for large shared data banks." Communications of the ACM 13.6 (1970): 377-387.

[15] MySQL. “http://www.mysql.com/.” internet - Retrieved January 3, 2013.

[16] PostgreSQL. “http://www.postgresql.org/.” internet - Retrieved January 3, 2013.

[17] Oracle. “http://www.oracle.com/index.html.” internet - Retrieved January 3, 2013.

[18] Date, Chris J., and Hugh Darwen. “A Guide to the SQL Standard.” Vol. 3.

[19] International Organization for Standardization. “International Standard ISO/IEC 9075-1:2003 (SQL:2003).” 2006.

[20] R. Elmasri and S. B. Navathe. “Fundamentals of database systems.” 5th ed: Addison-Wesley, 2007.

[21] Ramakrishnan, Raghu and Johannes Gehrke. “Database management systems.” Vol. 3. McGraw-Hill, 2003.

[22] Gray, Jim, and Andreas Reuter. “Transaction processing.” Kaufmann, 1993.

[23] Rasha Osman and William Knottenbelt. “Database system performance evaluation models: A survey.” Performance Evaluation Journal Volume 69, Issue 10, pp. 471-493, October 2012.

[24] R. Osman. “Performance Modelling of Database Designs using a QueueingNetworks Approach.” PhD thesis, University of Bradford, 2010.

[25] Courtois, Pierre-Jacques, Frans Heymans, and David Lorge Parnas. “Concurrent control with “readers” and “writers”.” Communications of the ACM 14.10 (1971): 667-668.

[26] Oracle . My SQL 5.6 Reference Manual. Internal Locking Methods, “http://dev.mysql.com/doc/refman/5.6/en/internal-locking.html”. internet - Retrieved May 30, 2013

[27] PostgreSQL. PostgreSQL 9.1.7 Documentation, “http://www.postgresql.org/docs/9.1/static/index.html”. internet - Retrieved January 5, 2013

[28] The Transaction Processing Performance Council. TPC-W Benchmark version 2. “http://www.tpc.org/tpcw/” internet - Retrieved January 9, 2013

[29] PostgreSQL. PostgreSQL 9.1 libpq Documentation, “http://www.postgresql.org/docs/9.1/static/libpq.html” internet - Retrieved November 29, 2012

[30] Simon Spinner, Samuel Kounev, and Philipp Meier. Stochastic Modeling and Analysis using QPME: Queueing Petri Net Modeling Environment v2.0. In Serge Haddad and Lucia Pomello, editors, Proceedings of the 33rd International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2012), Hamburg, Germany, volume 7347 of Lecture Notes in Computer Science (LNCS), pages 388-397, Berlin, Heidelberg. Springer-Verlag. June 27-29, 2012

[31] ANTLR.“http://www.antlr.org/” internet - Retrieved March 12, 2013

[32] PostgreSQL. PostgreSQL 9.1.7 Documentation on EXPLAIN statements. “http://www.postgresql.org/docs/9.1/static/sql-explain.html”. internet - Retrieved November 29, 2012

[33] PostgreSQL. PostgreSQL 9.2 Documentation on pgbench. “http://www.postgresql.org/docs/9.2/static/pgbench.html”. internet - Retrieved May 26, 2013

[34] David Coulden, Rasha Osman, and William J. Knottenbelt. 2013. Performance modelling of database contention using queueing petri nets. In Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE ‘13), Seetharami Seelam (Ed.). ACM, New York, NY, USA, 331-334.

[35] Rasha Osman, David Coulden and William J. Knottenbelt. 2013. Performance modelling of concurrency control schemes for relational databases. In Proceedings of the 20th International Conference on Analytical and Stochastic Modelling Techniques and Applications (ASMTA ‘13).

[36] Thomasian, A.; In Kyung Ryu, “Performance analysis of two-phase locking,” Software Engineering, IEEE Transactions on , vol.17, no.5, pp.386,402, May 1991

[37] Kounev, Samuel, and Simon Spinner. ”QPME 2.0. User’s guide” (2011).

Appendix A

Transaction workload specification language grammar

⟨*specification*⟩ ::= ⟨*transaction*⟩+

⟨*transaction*⟩ ::= '`transaction`' identifier float '`{`' ⟨*transaction-body*⟩ '`}`'

⟨*transaction-body*⟩ ::= ⟨*statement*⟩+

⟨*statement*⟩ ::= '`if`' float '`{`' ⟨*transaction-body*⟩ '`}`'
| ⟨*sql-statement*⟩
| '`statement`' ⟨*attribute*⟩* '`{`' ⟨*sql-statement*⟩ '`}`'

⟨*sql-statement*⟩ ::= sql-text

⟨*attribute*⟩ ::= '`exclusive`' '`=`' boolean
| '`runtime`' '`=`' float
| '`table`' '`=`' identifier

Appendix B

Published material

B.1 Performance modelling of database contention using queueing petri nets. Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE '13)

Performance Modelling of Database Contention using Queueing Petri Nets

David Coulden Rasha Osman William J. Knottenbelt
Department of Computing
Imperial College London
London SW7 2AZ, UK
{drc09, rosman, wjk}@imperial.ac.uk

ABSTRACT
Most performance evaluation studies of database systems are high level studies limited by the expressiveness of their modelling formalisms. In this paper, we illustrate the potential of Queueing Petri Nets as a successor of traditionally-adopted modelling formalisms in evaluating the complexities of database systems. This is demonstrated through the construction and analysis of a Queueing Petri Net model of table-level database locking. We show that this model predicts mean response times better than a corresponding Petri net model.

Categories and Subject Descriptors
C.4 [Performance of Systems]: *Modelling techniques.*

General Terms
Performance

Keywords
Queueing Petri nets, performance modelling, database locking.

1. INTRODUCTION
The data landscape has changed dramatically in size and complexity in the past decade. The Internet, ubiquitous communication, cloud services and e-Science applications have led to an explosion in data generation and storage. A large proportion of this data is stored and managed in databases. Market growth for relational database management systems is expected to double by 2016 [11], making performance of these large DBMSs a critical issue for users and vendors alike.

Database system performance is influenced by complex and interdependent functionalities (e.g. transaction usage scenarios, database cache management, disk contention, concurrency and lock contention and the implementation of logical and physical structures in DBMS). In the performance evaluation literature there have been many performance studies of different components of database systems and many methodologies developed for their performance evaluation [14]. However, the impact of these studies on industry has been limited. One of the reasons is the lack of detailed modelling needed to represent production-grade database systems. A main cause is the interaction of physical and logical resources within database systems, which is difficult to represent using traditional modelling formalisms. In this work, we return to the issue of modelling database systems by modelling table level Two Phase Locking using queueing Petri nets and illustrate the potential that this new approach has in performance modelling of database systems.

Queueing Petri Nets (QPNs) [1] extend coloured stochastic Petri nets by incorporating queues and scheduling strategies into places forming *queueing places*, thus producing a very powerful modelling formalism that has the synchronization capabilities of Petri nets (PNs) while also being capable of modelling queueing behaviours. These queueing places consist of two components: the *queue*, and the *depository* where serviced tokens (customers) are placed. Tokens enter the queueing place through the firing of input transitions, as in other Petri nets; however, as the entry place is a queue they are placed in the queue according to the scheduling strategy of the queue's server. Once a token has been serviced it is deposited in the depository where it can be used in further transitions. Queueing places can have variable scheduling strategies and service distributions; these are known as *timed queueing places*. *Immediate queueing places* impose a scheduling discipline on arriving tokens without a delay.

Queueing Petri nets have been recently applied in the performance evaluation of component-based distributed systems [4, 5] and grid environments [10]. In this paper, we apply QPN in modelling locking contention in database systems. We have chosen QPN as the modelling formalism over other variations of Petri nets, as the queueing places allow for the representation of lock scheduling in database systems, while the places and transitions naturally represent the flow of execution of a transaction in the system. Even though queueing network models (QNMs) are currently the prevailing formalism for performance modelling of database systems, QPNs are more expressive when representing simultaneous resource possession and blocking.

The rest of this paper is organized as follows. Section 2 overviews related work. Section 3 details the QPN model for a particular measured system. Section 4 analyzes the results and Section 5 concludes the paper.

2. RELATED WORK
2.1 Queueing Network Models
Osman and Knottenbelt [14] surveyed queueing network performance models of database systems. They found that the majority of studies that model concurrency control in database systems assume a uniform distribution of the locks over the total number of data items, in addition to representing update only transactions in the models. While these assumptions produce tractable models, they neglect hot-spots and do not represent the

ICPE'13, April 21–24, 2013, Prague, Czech Republic.

effect of read transactions holding shared locks on the execution of update transactions that hold conflicting locks. Moreover, these models assess the performance of the transaction at the physical hardware level; therefore they are incapable of representing the effect of lock conflicts at the table or row level, which is more beneficial to database performance tuning.

2.2 Petri Nets

There is a paucity of studies that apply Petri nets to database systems in comparison to the available research that applies queueing network models. Of the studies that do exist, Chen [2] analyzes deadlock detection scheduling in centralized databases using stochastic Petri nets. The places in the model represent transaction execution states, i.e., waiting for a lock, locking an item, CPU processing, etc. Firing of the timed transitions represents the delay in moving from one state to the next.

For parallel databases, Mikkilineni et al. [9] use a Petri net to model concurrent parallel query execution plans in a distributed database. In the PN model, the transitions represent query operations and the places represent data blocks. The firing of a transition represents data communications. Jenq et al. [3] analyze two-phase locking in a parallel database machine using a two-layered model. The higher-layer model is a Petri net representing the parallel and synchronized execution of the relational operations of a transaction. The lower-level model is a QNM that represents the hardware resources and lock waiting queues.

The modelling approach presented in this paper differs from that of previous work in that we do not model the synchronization of query execution plans. Instead, we borrow the concept of modelling the execution of the transaction at the database table level from previous work in modelling database systems using queuing networks [13]. This work is an improvement over previous models of database systems in that we are able to represent locking contention between read and write transactions in a way similar to that of actual systems. Moreover, our QPN models have a more intuitive structure that maps the transaction and database design to the QPN model. This makes the model easier to comprehend by database developers and administrators.

3. Queueing Petri Net Model

3.1 Measured System

DBMSs implement concurrency control through locking protocols. The most widely used protocol is Strict Two Phase Locking (Strict 2PL) [15]. Strict 2PL forces transactions to hold *exclusive locks* to modify data and *shared locks* to read data. For a transaction to acquire an exclusive lock on a data object, no other transaction should hold a shared or exclusive lock on that object. Transactions can acquire shared locks on data objects only if no transaction has an exclusive lock on the objects. In our case study, we model Strict 2PL at the table level. A version of table level Strict 2PL is the default locking method implemented in the MySQL default storage engine [12]. In our experiments we use the PostgreSQL 9.1 [16] DBMS. In order to mimic table level Strict 2PL we explicitly lock the table within the transactions.

The measured system has two types of transactions: shared and exclusive that compete to access *Table A* (100,000 rows). Both transactions explicitly lock the table in the appropriate lock mode (shared or exclusive) and read or modify 1% of the table rows. Access to the table is achieved through a full table scan, i.e. no index is utilized by either transaction. The structure of the transactions is shown in Figure 1. In order to simulate a TPC-W-like workload in which update transactions are longer than read transactions [17], the execution time of the exclusive transaction is artificially lengthened by 40ms. Clients submit transactions to the DB server with exponentially distributed think times with mean 500ms. The mean response time of each transaction type is measured and compared to that emerging from the QPN model for different transaction mixes. The measured system was run on an Intel(R) Core(TM) i7-2600 CPU@3.40GHz box running Ubuntu 12.10 64-bit and PostgreSQL 9.1.

```
shared transaction
BEGIN;
  LOCK TABLE Table A in
                ACCESS SHARE MODE;
  SELECT count(*) FROM Table A
                WHERE id > value;
END;

exclusive transaction
BEGIN;
  LOCK TABLE Table A in
                ACCESS EXCLUSIVE MODE;
  UPDATE Table A SET other-id = other-value
                WHERE id > value;
   SELECT pg_sleep(0.04);
END;
```

Figure 1. Structure of shared and exclusive transactions.

3.2 QPN Model of Table Level Locking

The QPN model for the measured system was developed using QPME2.0 [6, 7]; it is detailed in Figure 2. The clients are represented by a timed queueing place with an infinite-server queue. The tokens in the *client* place have two colors; each color represents a client of one transaction type. Clients submit transaction jobs to the database server after an exponentially distributed think time. Then, transactions enter the *lock waiting* place where they wait for the lock on the table to be free. The lock waiting place is an immediate queueing place with FIFO departure discipline which ensures that the transactions are serviced in order of arrival.

The table-level locking mechanism is represented using a *lock repository* place, which is an ordinary place containing *lock* tokens. A share transaction will require one lock token and an exclusive transaction will require the maximum number of tokens defined for the lock repository place. By setting the number of lock tokens within the lock repository place to be equal to the maximum number of share transactions, all share transactions will be able to run simultaneously and an exclusive transaction will be forced to wait if there is a least one share transaction accessing the table. Any transaction entering the database queues behind any waiting transaction. Once a transaction has acquired a lock it will access *Table A*. Table A is represented by a timed queuing place with an infinite server queue which models transaction execution. Each type of transaction is treated as having an exponentially distributed service time that models the entire execution of the transaction. When a transaction has been serviced it will be passed back into the client place to repeat the process.

In our QPN model, we are assuming logical resources are the bottleneck, not physical resources. Therefore, the model does not directly capture disk and CPU contention and performance. However, the effects of processing are partially reflected in the

G/M/∞-IS queueing place representing *Table A*. Infinite server scheduling models the forking of PostgreSQL processes for each database connection. To minimize the effect of DBMS automated disk access, the default PostgreSQL configuration has been modified[1]. This modified configuration will not eliminate disk access but configures the DBMS for performance instead of durability [16].

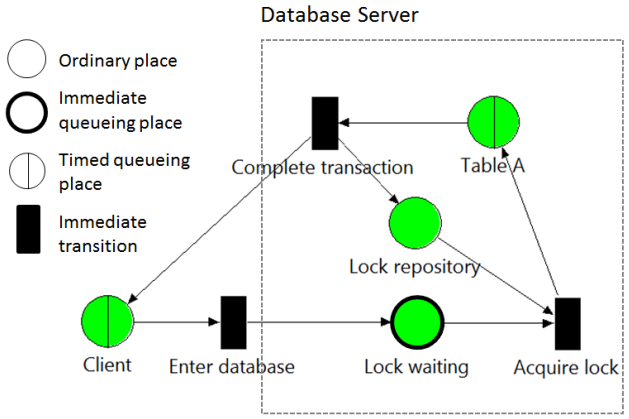


Figure 2. Queueing Petri net model of table level locking.

4. RESULTS

The experiment was based on the workloads of the TPC-W benchmark. The TPC-W benchmark is an e-commerce benchmark implementing an on-line bookstore. It has three workload mixes [8]: the *browsing mix* has 95% reads and 5% updates, the *shopping mix* has 80% reads and 20% updates, and the *ordering mix* has 50% reads and 50% updates.

Each transaction type was executed in isolation, i.e. without any locking contention, and the mean response time calculated. The measured mean response time for the shared transaction was 18.8ms and for the exclusive transaction 60.4ms.

We compare the measured system with our QPN model and an equivalent PN model. The PN model is the same as the QPN model in Figure 2 except that the *lock waiting* place is an ordinary place[2]. We have not compared to a QNM, as this type of simultaneous resource possession and blocking is difficult to express using QNMs.

The results for the browsing, shopping and ordering workloads are presented in Figure 3. First, we will discuss the performance of the actual system. For the browsing workload (Figure 3(a)) the share transactions dominate the traffic and their performance is minimally affected by the increase in exclusive transactions. The *step-like* trend for both transactions is caused by the constant number of exclusive transactions for the corresponding number of clients. For the shopping mix (Figure 3(b)) the increased number of exclusive transactions has affected the performance of both types of transactions, i.e. lock waiting time has increased in comparison to transaction execution time. This is especially evident for the shared transactions where we notice a sharp increase in the mean response time for large number of clients. For the ordering workload (Figure 3(c)) in which the number of shared and exclusive transactions are equal, lock waiting time dominants transaction execution which is evident in the performance degradation of the shared transactions, leading to approximately the same response times for both transaction types at high client numbers.

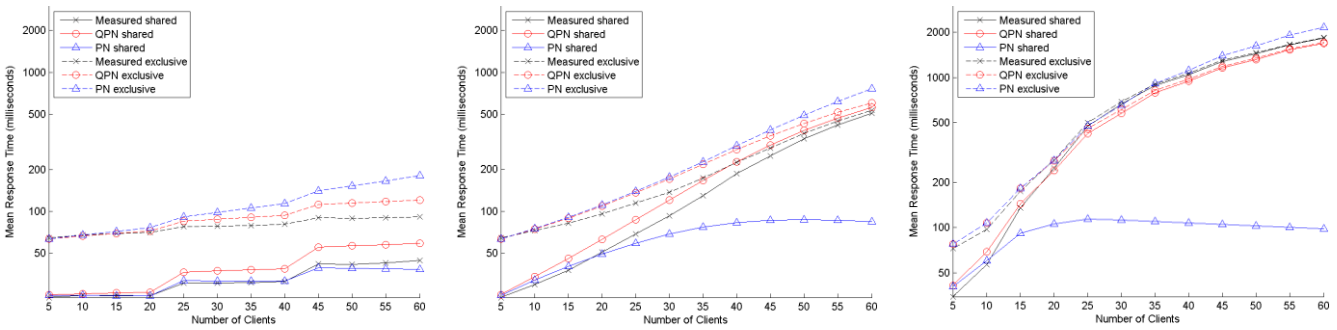
The PN model severely underestimates the performance of the exclusive transactions for the browsing workload with an error of 97% at 60 clients. However, it overestimates the performance of the shared transactions, with an error of 13% at 60 clients. This is due to the fact that in the PN model the transactions do not queue for locking, as in the real system; and therefore the exclusive transactions are starved. This trend continues as the percentage of exclusive transactions increases in the shopping and ordering workloads. From Figures 3(b) and 3(c), the PN model overestimates the performance of the shared transaction by 83% and 94%, and underestimates the performance of the exclusive transactions by 43% and 17% at 60 clients for the shopping and ordering workloads respectively. The increase in exclusive transactions in the workloads increases their probability of holding the lock token, thus the accuracy of the PN increases as the number of exclusive transactions increase. However, the opposite effect is seen on the shared transactions as without a scheduling discipline they are able to skip ahead of the exclusive transactions whenever a lock token is held by at least one share transaction.

The QPN model underestimates the performance of both transactions for the browsing workload with an error of 32% at 60 clients. The accuracy of the QPN model increases as the number of exclusive transactions increase in the system, i.e. when the lock waiting times dominate the response times. For the shopping workload the QPN underestimates the performance of both transactions with an error of 10% for the shared transaction and 13% for the exclusive transaction at 60 clients. The QPN model overestimation is possibly due to the unaccounted multi-core processing. For the ordering workload, the QPN correctly predicts that both share and exclusive transactions have approximately the same response times. Unlike the previous workloads, the QPN model underestimates the performance of both transaction types when the number of clients is less than 20, with an average error of 6%. When the number of clients is 20 or more, the QPN model overestimates the performance of both transaction types with an average error of 8% at 60 clients. The overestimate is due to the high updates that cause more disk access which, in turn, affects the response times of both transaction types. This is not accounted for in the QPN model. Nonetheless, the QPN model is able to follow the performance trend for both transactions for all workloads, especially the share transaction in comparison to the PN model.

---

[1] The modified server configuration parameters are: fsync=off, synchronous_commit=off and checkpoint_segments =600. The reader is referred to [16] for a definition of these parameters.

[2] Access to *Table A* should have been modelled as a timed transition with infinite service. However, QMPE2.0 does not support timed transitions [6], so this was approximated by a serial network consisting of an immediate transition, a timed queueing place and a second immediate transition similar to the QPN model.



(a) browsing (95% shares and 5% exclusives) **(b)** shopping (80% shares and 20% exclusive) **(c)** ordering (50% shares and 50% exclusive)

Figure 3. Mean response time for TPC-W workload.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have demonstrated the potential in modelling relational database systems using Queueing Petri Nets, thus overcoming some of the limitations of queueing network models and Petri nets which are currently the main modelling formalisms used to represent database systems.

We have presented a QPN model of two-phase table-level locking. The QPN model was able to approximate the queueing of the lock requests to the table for varying workloads, in contrast to an equivalent Petri net model.

This paper is a starting point for further investigations, which will include the extension of the QPN model to incorporate the effect of hardware contention on the system. This will lead to modelling of more representative database systems with more realistic workloads. Furthermore, for this approach to be feasible and applicable, an automated mapping tool will be developed. We will also investigate emerging paradigms, e.g. NoSQL databases, although the diversity of their data models and implementations will likely mean a generic modelling framework will be infeasible.

6. REFERENCES

[1] Bause, F. 1993. Queueing Petri Nets—A Formalism for the Combined Qualitative and Quantitative Analysis of Systems. In *Proc. 5th Int'l Workshop Petri Nets and Performance Models* (Oct 19-22, 1993), 14-23.

[2] Chen, I.-R. 1995. Stochastic Petri Net Analysis of Deadlock Detection Algorithms in Transaction Database Systems with Dynamic Locking. *The Computer Journal*, 38, 9 (1995), 717-733. DOI: 10.1093/comjnl/38.9.717

[3] Jenq, B.-C., Twichell, B.C.; Keller, T.W. 1989. Locking performance in a shared nothing parallel database machine. *IEEE Transactions on Knowledge and Data Engineering*, 1, 4 (Dec 1989), 530-543.DOI: 10.1109/69.43427

[4] Kounev, S. 2006. Performance modeling and evaluation of distributed component-based systems using queueing Petri nets. I*EEE Trans. Software Engineering*, 32, 7 (July 2006), 486-502.

[5] Kounev, S. and Buchmann, A. 2003. Performance modelling of distributed e-business applications using queuing petri nets. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software* (Mar 2003), 143-155.

[6] Kounev, S. and Spinner, S. 2011. *QPME 2.0 User's Guide*. (May 2011) Karlsruhe Institute of Technology, Germany. http://descartes.ipd.kit.edu/fileadmin/user_upload/descartes/QPME/QPME-UsersGuide.pdf

[7] Kounev, S., Spinner, S. and Meier, P. 2010. QPME 2.0-A Tool for Stochastic Modeling and Analysis Using Queueing Petri Nets. In *From Active Data Management to Event-Based Systems and More*, LNCS vol 6462, 293-311.

[8] Menasce, D. A. 2002. TPC-W: a benchmark for e-commerce. *IEEE Internet Computing*, 6, 3 (May/June 2002), 83-87. DOI: 10.1109/MIC.2002.1003136

[9] Mikkilineni, K. P., Chow, Y.-C. and Su, S. Y. W. 1988. Petri-net-based modeling and evaluation of pipelined processing of concurrent database queries. *IEEE Trans. Software Engineering*, 14, 11 (Nov 1988), 1656-1667. DOI: 10.1109/32.9053

[10] Nou, R., Kounev, S., Julia, F. and Torres, J. 2009. Autonomic QoS control in enterprise Grid environments using online simulation. *Journal of Systems and Software*, 82, 3 (March 2009), 486-502.

[11] Olofson, C. W. 2012. *Worldwide Relational Database Management Systems 2012–2016 Forecast.* International Data Corporation (August 2012**)**, Doc # 236273. www.idc.com

[12] Oracle Corporation 2013. *MySQL 5.6 Reference Manual. Internal Locking Methods*, http://dev.mysql.com/doc/refman/5.6/en/internal-locking.html

[13] Osman, R., Awan, I. and Woodward, M. E. 2011. QuePED: Revisiting Queueing Networks for the Performance Evaluation of Database Designs. *Simulation Modelling Practice and Theory*, 19, 1 (Jan 2011), 251-270.

[14] Osman, R. and Knottenbelt, W. J. 2012. Database System Performance Evaluation Models: A Survey. *Performance Evaluation*, 69, 10 (Oct 2012), 471–493. DOI=10.1016/j.peva.2012.05.006

[15] Ramakrishnan, R. and Gehrke, J. 2003. *Database management systems*. McGraw-Hill, Boston, Mass.

[16] The PostgreSQL Global Development Group 2012. *PostgreSQL 9.1.7 Documentation*. http://www.postgresql.org/docs/9.1/static/index.html.

[17] The Transaction Processing Performance Council 2003. *TPC-W Benchmark version 2.* http://www.tpc.org/tpcw/

B.2 Performance Modelling of Concurrency Control Schemes for Relational Databases. 20th International Conference, ASMTA 2013

Performance Modelling of Concurrency Control Schemes for Relational Databases

Rasha Osman, David Coulden, and William J. Knottenbelt

Department of Computing, Imperial College London
London SW7 2AZ, UK
{rosman,drc09,wjk}@doc.ic.ac.uk

Abstract. The performance of relational database systems is influenced by complex interdependent factors, which makes developing accurate models to evaluate their performance a challenging task. This paper presents a novel case study in which we develop a simple queueing Petri net model of a relational database system. The performance of the database system is evaluated for three different concurrency control schemes and compared to the results predicted by a queueing Petri net model. The results demonstrate the potential of our modelling approach in modelling database systems using relatively simple models that require minimal parameterization. Our models gave accurate approximations of the mean response times for shared and exclusive transactions with average prediction errors of 10% for high contention scenarios.

1 Introduction

It is now commonplace for organizations to each manage tens of petabytes of data. A large proportion of this data is stored in databases and is managed by database management systems (DBMSs). Despite the increasing use of NoSQL databases, relational databases are still the industry's main data model with forecasted steady growth to 2016 [6]. As users' expectations of performance and availability increases, the performance of these large DBMSs becomes a critical issue for organizations and vendors alike.

The performance engineering community has contributed many performance studies of database system components and several methodologies have been proposed for database system performance evaluation [9]. However, the impact of these studies on industry has been limited. One of the reasons may be that database system performance is affected by complex and interdependent interactions of physical and logical resources, which are difficult to represent using traditional modelling formalisms.

In previous work [2], we have demonstrated the suitability of queueing Petri nets (QPNs) as a modelling formalism for relational database contention. Queueing Petri nets [1] extend coloured stochastic Petri nets by incorporating queues and scheduling strategies into places forming *queueing places*, thus producing a powerful modelling formalism that has the synchronization capabilities of Petri nets while also being capable of modelling queueing behaviours. The queueing

places in QPNs allow for the accurate representation of lock scheduling as implemented in DBMSs, while the places and transitions naturally represent the flow of execution of a transaction in the system. Moreover, unlike previous studies of database concurrency control (see [9] for a survey), our models are able to reflect lock conflicts between read and update transactions.

DBMSs implement concurrency control through locking protocols. The most widely used protocol is Strict Two Phase Locking (Strict 2PL) [10]. Strict 2PL forces transactions to hold *exclusive locks* to modify data and *shared locks* to read data. For a transaction to acquire an exclusive lock on a data object, no other transaction should hold a shared or exclusive lock on that object. Transactions can acquire shared locks on data objects only if no transaction has an exclusive lock on the objects.

In this paper, we expand on our previous work and use QPNs to model a database system under three modes of Strict 2PL, specifically table-level, row-level and multi-version 2PL, each of which is implemented in commercial DBMSs. Our results demonstrate the capability of our modelling approach to reflect the dynamic operation of relational database systems using simple models that require minimal parameterization.

The rest of this paper is organized as follows. Section 2 describes the QPN model of the measured database system. In Sections 3, 4 and 5 we present the QPN models and results for table-level, row-level and multi-version 2PL, respectively. Section 6 concludes the paper and provides directions for future work.

2 General QPN Model of a Database System

Queueing places in QPN models consist of two components: the *queue*, and the *depository* where serviced tokens (customers) are placed. Tokens enter the queueing place through the firing of input transitions, as in other Petri nets; however, as the entry place is a queue they are placed in the queue according to the scheduling strategy of the queue's server. Once a token has been serviced it is deposited in the depository where it can be used in further transitions. Queueing places can have variable scheduling strategies and service distributions; these are known as *timed queueing* places. *Immediate queueing* places impose a scheduling discipline on arriving tokens without a delay. Due to space limitations, we refer the reader to [1] for a more detailed description of QPNs.

Figure 1 shows the general components of the QPN model for the modelled database system used in this work. The database system is composed of three tables, A, B and C. The system has two types of transactions: shared (read) and exclusive (update), with each client submitting one transaction type to the database server. The details of tables A and B are presented in their respective sections, as their components depend on the locking mechanism. There is no contention between transactions for table C and therefore no locking is modelled for table C. The *sleep* place is a timed queueing place representing the artificial delay for the exclusive transactions only.

The clients are represented by a timed queueing place with an infinite-server queue with an exponentially distributed think time with mean 200ms. The tokens in the *client* place have two colours; of which represent a client of one transaction type. The transactions enter the database through the *initial-processing* timed queueing place, which represents the delay for setting up the transaction when executing the BEGIN statement. A transaction will leave the database through the *final-processing* timed queueing place, which represents the time to commit the transaction and release its locks.

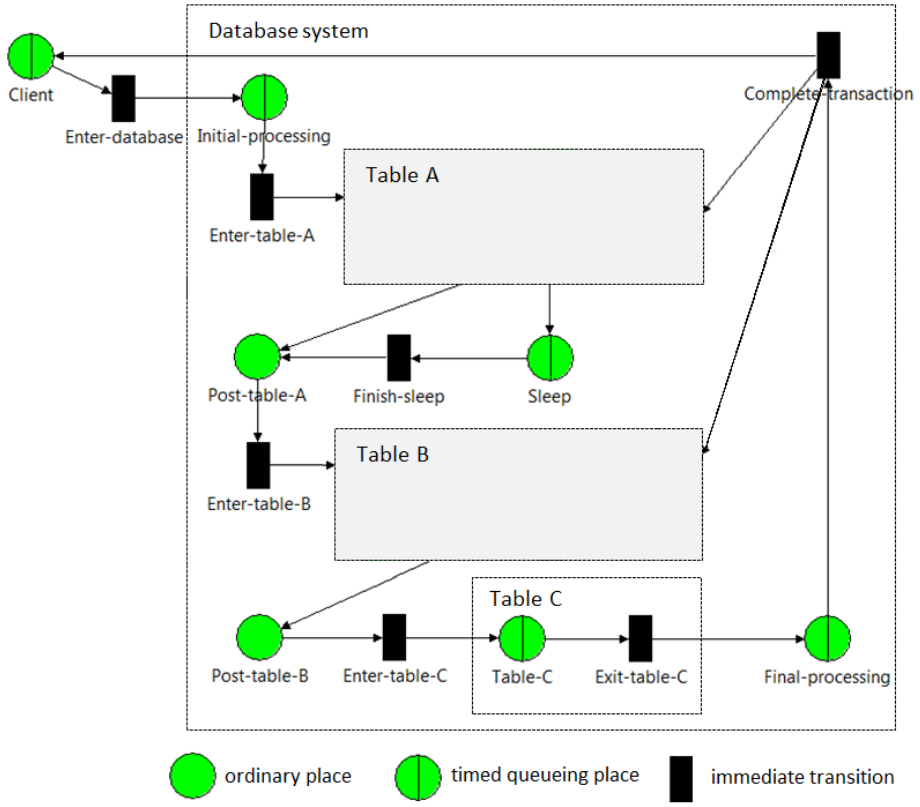


Fig. 1. General QPN model of the measured database system.

A table is represented by a timed queuing place with an infinite server queue (*table-C* place in Fig. 1) that models transaction execution. Each type of transaction has its individual exponentially distributed service time that represents the execution time of the transaction when accessing the specific table. Here, we are assuming that the database table represents the main service centre for the transactions. This concept is inspired by previous work in modelling database systems using queuing networks [8] and is similar to other work that abstracts

transaction CPU and disk execution by one service centre with an exponential service distribution [9]. When a transaction has been serviced at a table, it will continue to the next table or if it has finished executing it will continue to the *final-processing* place and then be passed back to the *client* place to repeat the process.

The QPN model reflects the logical execution of transactions on the database system. Therefore, the model does not directly model disk and CPU performance. However, the effects of processing are partially reflected in the *initial-processing* and *final-processing* places and in the $G/M/\infty - IS$ queueing places representing the tables. Infinite server scheduling models the forking of PostgreSQL processes for each database connection. To minimize the effect of DBMS automated disk access, the default PostgreSQL configuration has been modified[1]. This modified configuration will not eliminate disk access but configures the DBMS for performance instead of durability [12].

The QPN models in this paper were developed and solved using QPME2.0 [3]. QPME2.0 (Queueing Petri net Modeling Environment) is an open source performance modelling tool based on the QPN modelling formalism. QPME2.0 is composed of two components, a QPN editor (QPE) and a simulator for QPNs (SimQPN). All our simulation runs used the *method of non-overlapping batch means* (with the default settings) to estimate the steady state mean token residence times with 95% confidence intervals.

Experimental Setup. The measured database system is based on the workloads of the TPC-W benchmark. The TPC-W benchmark is an e-commerce benchmark implementing an on-line bookstore. It has three workload mixes [4]: the *browsing mix* has 95% reads and 5% updates, the *shopping mix* has 80% reads and 20% updates, and the *ordering mix* has 50% reads and 50% updates. For the experiments in the following sections, the total number of clients represents the sum of shared and exclusive transactions for the measured workload mix.

The shared and exclusive transactions access three tables, A, B and C. Both transaction types access the tables in the same order, adhering to the specifications of the TPC-W benchmark. For tables A and B, transactions randomly access one row out of a maximum of five rows. A row is chosen randomly by primary key for each transaction instance. For table C, transactions SELECT a random number of rows. Both transactions explicitly or implicitly lock the tables/rows in the appropriate lock mode (shared or exclusive) and read or modify one row for tables A and B and read a set of rows from table C. All tables have a primary key index, which is utilized by the transactions in the measured systems.

In order to simulate a TPC-W like workload in which update transactions are longer than read transactions [13], the execution time of the exclusive transac-

[1] The modified server configuration parameters are: fsync=off and synchronous_commit=off. The reader is referred to [12] for a definition of these parameters.

tion is artificially lengthened by 100ms, this is represented by the *sleep* place. To parameterize the QPN models, each transaction type is executed on the measured system in isolation (i.e. without any locking contention) and the mean service time for each QPN place was extracted from the DBMS logs. These are shown in Table 1.

The measured system was run on a virtual machine with four virtual processors@2.6GHz running Ubuntu Linux 10.04 64-bit and PostgreSQL 9.0 DBMS [12]. Each table is an average of 5MB in total size and has approximately 25 000 randomly generated rows. The mean response time of each transaction type is measured and compared to that emerging from the QPN model for the different transaction mixes.

Table 1. Mean service times (in milliseconds) for QPN model transaction types (colours). For table-level 2PL, the service times for the *final-processing* place include commit and lock statement mean execution times. All places have exponentially distributed service times, except for the row-level 2PL model, in which the *initial-processing* and *final-processing* places have a deterministic distribution with zero service time for the shared transaction. This is because the shared transaction in the row-level scenario is not contained within a BEGIN/END block.

	table-level		row-level		multi-version	
QPN places	shared	exclusive	shared	exclusive	shared	exclusive
initial-processing	0.06	0.03	0	0.04	0.03	0.03
table A	0.18	0.60	0.26	0.28	0.19	0.23
sleep	-	100.24	-	100.27	-	100.24
table B	0.12	0.18	0.13	0.23	0.11	0.19
table C	6.52	7.24	13.85	14.18	6.48	6.86
final-processing	0.14	0.74	0	0.05	0.03	0.04

3 QPN Model of Table-Level 2PL

Measured System. For this scenario, we model table-level Strict 2PL, which resembles the default locking method implemented in the MySQL default storage engine [7]. In order to implement table-level 2PL in PostgreSQL, each transaction type explicitly locks each table in the appropriate lock mode (shared or exclusive). The structure of the transactions is shown in Fig. 2. The shared transaction is composed of a set of smaller transactions; each represents an access to a table. This allows the shared transaction to release the shared lock on the table immediately after accessing the table. Therefore, exclusive transactions only wait for the shared transaction to leave a table, while shared transactions must wait for an exclusive transaction to commit before gaining access to any of the tables.

shared transaction

```
BEGIN;
  LOCK TABLE  Table A in
              ACCESS SHARE MODE;
  SELECT count(a-id) FROM  Table A
             WHERE id =  value-a;
END;
BEGIN;
  LOCK TABLE  Table B in
              ACCESS SHARE MODE;
  SELECT count(b-id) FROM  Table B
              WHERE id =  value-b;
END;
BEGIN;
  LOCK TABLE  Table C in
               ACCESS SHARE MODE;
  SELECT count(c-id) FROM   Table C
       WHERE id =  10000 +  value-c;
END;
```

exclusive transaction

```
BEGIN;
  LOCK TABLE  Table A  in
            ACCESS EXCLUSIVE MODE;
  UPDATE  Table A
            SET a-id =  other-value
            WHERE id =  value-a;
  SELECT pg_sleep(0.1);
  LOCK TABLE  Table B  in
             ACCESS EXCLUSIVE MODE;
  UPDATE  Table B
           SET b-id =  other-value
           WHERE id =  value-b;
  LOCK TABLE  Table C  in
            ACCESS SHARE MODE;
  SELECT count(c-id) FROM  Table C
      WHERE id <  10000 +  value-c;
END;
```

Fig. 2. Structure of shared and exclusive transactions for table-level 2PL.

QPN Model. The general components of the QPN model of the measured system were discussed in Section 2. Here, we present the aspects of the QPN model that are related to the modelling of table-level 2PL. Figure 3 depicts the QPN model of table A, the model for table B (not shown) is identical. Transactions wishing to enter the *table-A* (or *table-B*) place must acquire a suitable lock on the table by entering the *lock-wait-A* place, in which they will wait for the lock on the table to be free. The *lock-wait-A* place is an immediate queueing place (shown with a bold outline in Fig. 3) with FIFO departure discipline. The table-level locking mechanism is represented using the *lock-store-A* place, which is an ordinary place containing *lock* tokens. A shared transaction will require one lock token and an exclusive transaction will require the maximum number of tokens defined for the *lock-store-A* place. By setting the number of lock tokens within the *lock-store-A* place to be equal to the maximum number of shared transactions, all shared transactions will be able to run simultaneously. In contrast, an exclusive transaction will be forced to wait if there is a least one shared transaction accessing the table. This process is modelled within the *acquire/release-lock-A* transition.

The shared transactions release table locks on table A (and B) immediately after leaving the *table-A* place by depositing a token in the *prepare-lock-release-A* place, enabling the *acquire/release-lock-A* immediate transition, which returns one lock token to the *lock-store-A* place. Exclusive transactions release locks after leaving the *final-processing* place (Fig. 1) by depositing a token in the *prepare-lock-release-A/B* place of table A and B, enabling the *acquire/release-lock-A/B* transition, which returns the maximum number of tokens back to the *lock-store-A/B* place.

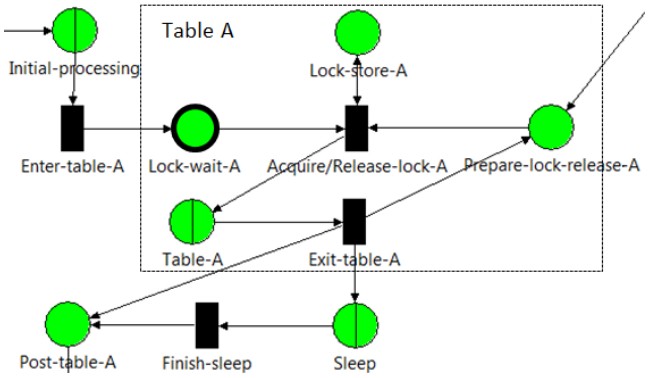


Fig. 3. QPN model of table A for table-level 2PL.

Results. The results for the browsing, shopping and ordering workloads are presented in Fig. 4. The QPN prediction error percentages are in Table 2. For the browsing workload, the model underestimates the performance of the shared transaction for 30 clients and less, with an average underestimation of 27%, while accurately predicting the mean response times for exclusive transactions. This can be attributed to the model not representing the processing which may favour the concurrently executing shared transactions. For client numbers of 50 and above, we see the opposite effect; the model overestimates performance for both transactions, with average error of 19% for both transaction types. Here, the system is empty (four exclusive transactions at 80 clients) therefore no contention is present, allowing other variables, such as processing and disk access to dominate transaction execution time. These factors are not explicitly represented in the model. Nonetheless, the model follows the trend of degradation of performance with increasing number of clients for both transaction types.

For the shopping and ordering workloads, when locking contention starts to dominate transaction execution (i.e. the number of exclusive transactions is greater than four) the model gives an excellent prediction of mean response times for both transaction types. The mean error for both transaction types is less than 6% for the shopping workload, and less than 2% for the ordering workload. We note that the modelling error for 10 clients for the shopping workload is similar to that of 10 and 20 clients for the browsing workload, as both scenarios have the same level of contention.

4 QPN Model of Row-Level 2PL

Measured System. Row-level Strict 2PL is the default locking mechanism implemented in Microsoft SQL Server [5]. In row-level 2PL, a shared transaction acquires a shared row-lock for the duration of the read statement only. Exclusive transactions hold exclusive row-locks on rows until transaction commit, thus

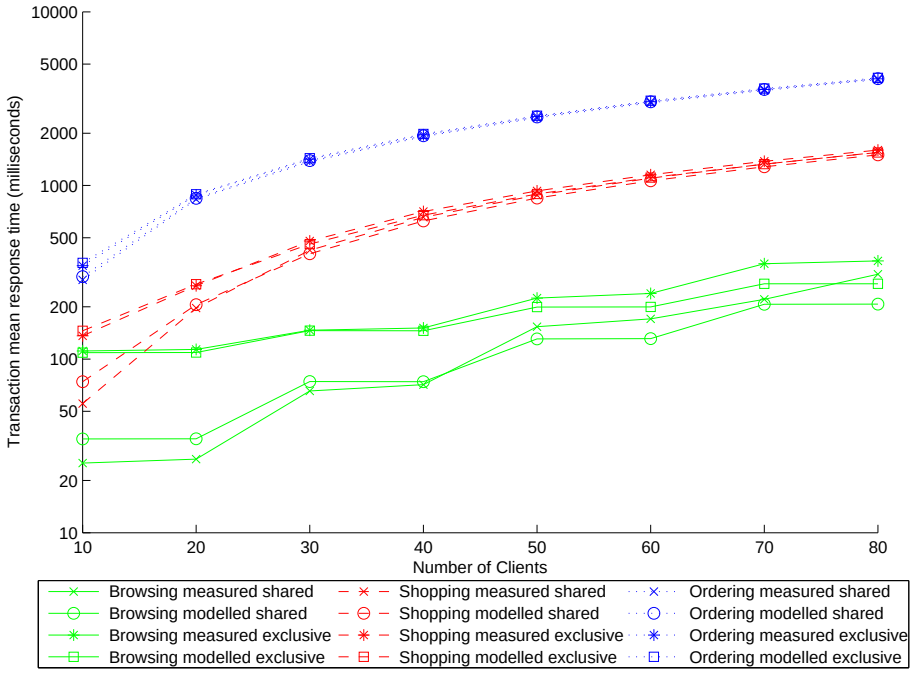


Fig. 4. Mean response time for TPC-W workload for table-level 2PL.

Table 2. Error percentages for table-level 2PL.

	browsing error (%)		shopping error (%)		ordering error (%)	
# of clients	shared	exclusive	shared	exclusive	shared	exclusive
10	37.79	-2.24	33.52	6.94	4.43	3.72
20	31.15	-3.93	4.73	2.21	1.29	0.88
30	13.37	-0.61	-4.57	-4.02	1.51	1.34
40	4.08	-3.52	-4.72	-4.67	1.29	1.15
50	-14.87	-11.34	-3.84	-3.99	1.23	1.16
60	-23.09	-16.41	-3.52	-3.68	1.25	1.17
70	-6.23	-23.35	-3.54	-3.69	1.38	1.35
80	-32.79	-26.08	-3.17	-3.23	1.35	1.34

preventing other shared and exclusive transactions from accessing these rows. The structure of the transactions is shown in Fig. 5.

To implement row-level Strict 2PL in PostgreSQL we utilize the default locking behavior of PostgreSQL statements. The shared transaction's SELECT statement includes the FOR SHARE clause, which implicitly acquires a shared lock on the retrieved rows thus preventing exclusive transactions from locking these rows [12]. A side effect of this is the increased processing needed to mark the row as locked. In addition, the shared transaction is not enclosed in a BEGIN/END block so that the shared lock is released upon statement completion. The exclusive transaction is similar to the exclusive transaction of table-level 2PL just without explicit table locks. Here the UPDATE statement acquires an implicit exclusive lock on the modified row, which is held until transaction commit.

shared transaction	exclusive transaction

SELECT count(a-id) FROM *Table A*
WHERE id = *value-a*
FOR SHARE;
SELECT count(b-id) FROM *Table B*
WHERE id = *value-b*
FOR SHARE;
SELECT count(c-id) FROM *Table C*
WHERE id = *10000* + *value-c*;
	BEGIN;
	UPDATE *Table A*
	SET a-id = *other-value*
	WHERE id = *value-a*;
	SELECT pg_sleep(0.1);
	UPDATE *Table B*
	SET b-id = *other-value*
	WHERE id = *value-b*;
	SELECT count(c-id) FROM *Table C*
	WHERE id < *10000* + *value-c*;
	END;

Fig. 5. Structure of shared and exclusive transactions for row-level 2PL.

QPN Model. Figure 6 shows the QPN model of table A for row-level 2PL. This model differs from the QPN model for table-level locking in the representation of the locking mechanism. Here, transactions are blocked waiting on row-locks to be released. The waiting queue for each row of the five rows is represented by the *lock-wait-A-row* place, which is an immediate queueing place. The *used-lock-store-A* and *unused-lock-store-A* are complementary ordinary places that hold the *lock* tokens, one token (lock) colour for each row. The maximum number of each *lock* token colour defined for the *unused-lock-store-A* place is greater than the maximum number of shared transactions. Hence, the total number of tokens, irrespective of colour, in the *unused-lock-store-A* place is more than five times the maximum number of shared transactions. The *used-lock-store-A* place holds identical token colours as that of the *unused-lock-store-A* place but their number is set to zero.

When the *enter-table-A* (or B) transition is enabled, a transaction entering table A randomly chooses a row and enters the corresponding *lock-wait-A-row* place to request the appropriate lock for the row. A shared transaction in a spe-

cific *lock-wait-A-row* place will require one of the corresponding lock tokens from the *unused-lock-store-A* place then deposit it into the *used-lock-store-A* place and enter the table for service. An exclusive transaction will require the maximum defined number of lock tokens for the corresponding row and deposit them into the *used-lock-store-A* place and enter the table for service. Therefore, all shared transactions will be able to access a row simultaneously if no exclusive transaction is currently updating that row. An exclusive transaction will be forced to wait if there is at least one shared transaction or an exclusive transaction accessing the row. This process is modelled within the *acquire-lock-A* transition.

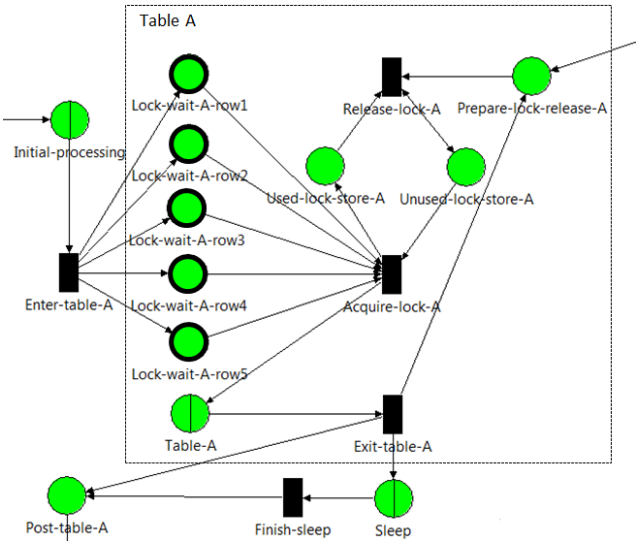


Fig. 6. QPN model of table A for row-level 2PL.

The shared transactions release row-locks on table A (or B) immediately after leaving the *table-A* place by depositing a token in the *prepare-lock-release-A* place, enabling the *release-lock-A* immediate transition. The *release-lock-A* transition randomly chooses a token colour that corresponds to a row that has been locked by a shared transaction, i.e. token colours that have a nonzero amount in the *used-lock-store-A* and the *unused-lock-store-A* places. Then one token is removed from each of the *used-lock-store-A* and the *unused-lock-store-A* places and two tokens are deposited into the *unused-lock-store-A* place[2].

Exclusive transactions release locks after leaving the *final-processing* place (Fig. 1) by depositing a token in the *prepare-lock-release-A* place of table A (and B), enabling the *release-lock-A* transition. The *release-lock-A* transition randomly chooses a token colour that corresponds to a row that has been locked

[2] This functionality should have been modelled with an inhibitor arc; however, QPME2.0 does not support inhibitor arcs [3].

by an exclusive transaction, i.e. token colours that have the maximum number of tokens in the *used-lock-store-A* place. In the same fashion as shared transactions, this number of tokens is removed from the *used-lock-store-A* place and deposited into the *unused-lock-store-A* place.

Results. Figure 7 shows that transaction performance is now better than that of table-level locking, as contention is at the row-level, with transactions blocked only if they want to acquire conflicting locks on a row. The error percentages of the QPN model are in Table 3.

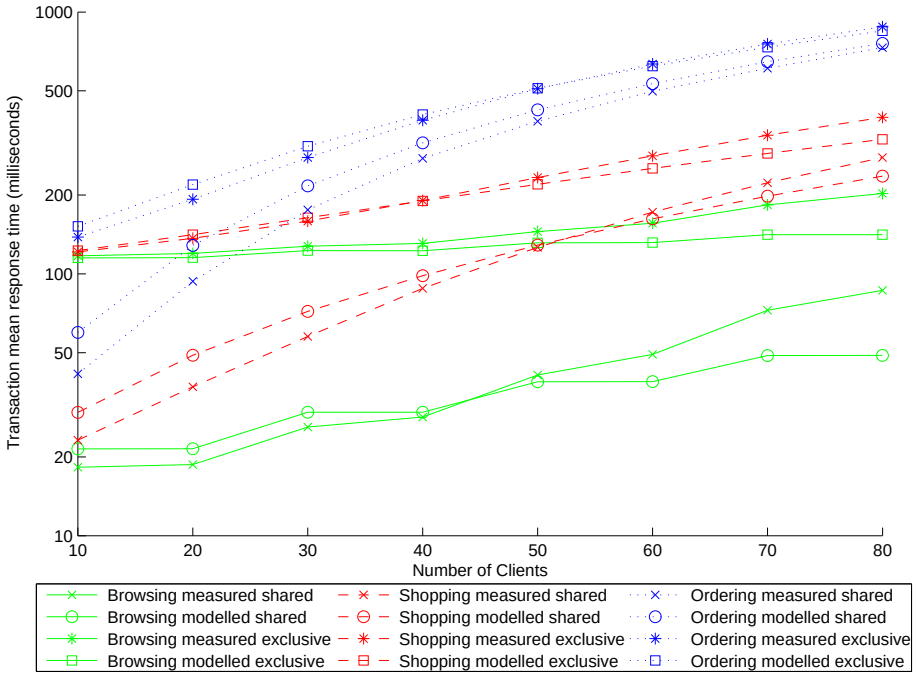


Fig. 7. Mean response time for TPC-W workload for row-level 2PL.

For the exclusive transaction, the model's accuracy for the browsing workload is similar to that of the table-level locking QPN. For the shopping workload, as the number of clients increase over 60, the model overestimates the performance of the exclusive transaction. This is likely due to the effect of increased processing time for the exclusive transaction due to increased SELECT FOR SHARE statements in the system. The effect of low contention is clear in the ordering workload; the model underestimates the performance of the exclusive transaction for number of clients 30 and below, then its accuracy increases as the number of exclusive transactions increase. The model's level of accuracy in-

Table 3. Error percentages for row-level 2PL.

	browsing error (%)		shopping error (%)		ordering error (%)	
# of clients	shared	exclusive	shared	exclusive	shared	exclusive
10	17.55	-1.84	27.73	1.08	43.97	9.86
20	14.96	-3.62	31.93	3.34	37.21	13.70
30	13.92	-3.75	24.72	2.92	23.51	10.45
40	4.39	-6.20	11.90	-0.42	14.59	5.09
50	-5.74	-9.36	1.98	-5.78	10.60	0.38
60	-21.27	-15.60	-5.73	-10.40	6.92	-1.73
70	-32.80	-22.96	-11.17	-14.80	5.79	-2.80
80	-43.60	-30.41	-15.06	-17.55	3.78	-3.54

creases when the lock contention is similar to that of the shopping and ordering workloads of table-level locking. This is most likely at 40 clients for the ordering workload.

For shared transactions, when the number of clients is 40 or less, the model underestimates their performance for all workloads. In addition to the effect of a light load on the processor, the QPN approximates the release of row-level locks by randomly selecting which same type row-lock to release upon transaction commit. This is in contrast to the actual system in which the transaction will release the same row-lock it acquired when it began execution. This will lead to over-blocking of transactions in the model, especially when there are less exclusive transactions in the system. Over-blocking has a higher effect on the response times of the shared transactions because they must wait for a longer running exclusive transaction to release its locks. The effect of over-blocking diminishes when lock waiting dominates transaction execution time for number of clients above 40 for the ordering workload.

In this scenario, there is less contention for locks in the system in comparison to table-level locking for the same number of clients, thus affecting the accurate prediction of the model for low contention settings. However, the QPN model still maintains its ability to follow the mean response time trend for both transactions for all workloads, as shown in Fig. 7.

5 QPN Model of Multi-version 2PL

Measured System. Multi-version Strict 2PL is similar to the default locking mechanism for PostgreSQL [12]. In this case, shared transactions read snapshots of the data representing the most recent consistent state regardless of the current state of the database [11]. Therefore, shared transactions do not block exclusive transactions and exclusive transactions do not block shared transactions. Alternatively, an exclusive transaction modifying a row blocks other exclusive transactions from accessing that row. This is because exclusive transactions modify the current consistent state of the database and hold exclusive locks on rows

until transaction commit. The structure of both transactions is similar to the transactions of table-level 2PL, without explicit LOCK TABLE statements, and the shared transaction is enclosed within one BEGIN/END block.

QPN Model. The QPN model for table A for multi-version 2PL is identical to the QPN model for row-level 2PL in Fig. 6, with the addition of an edge between the *enter-table-A* transition and the *table-A* place. This edge represents the shared transactions bypassing lock checking and accessing the rows of table A. There is no edge between the *exit-table-A* transition and the *prepare-lock-release-A* place, as no locks are held by shared transactions. Each token colour in the *unused-lock-store-A* place is initialized to one, i.e. only one exclusive transaction can hold a lock on a given row at any time. Lock release for exclusive transactions is identical to that of row-level locking except that only one token is transferred for each exclusive transaction when releasing its locks.

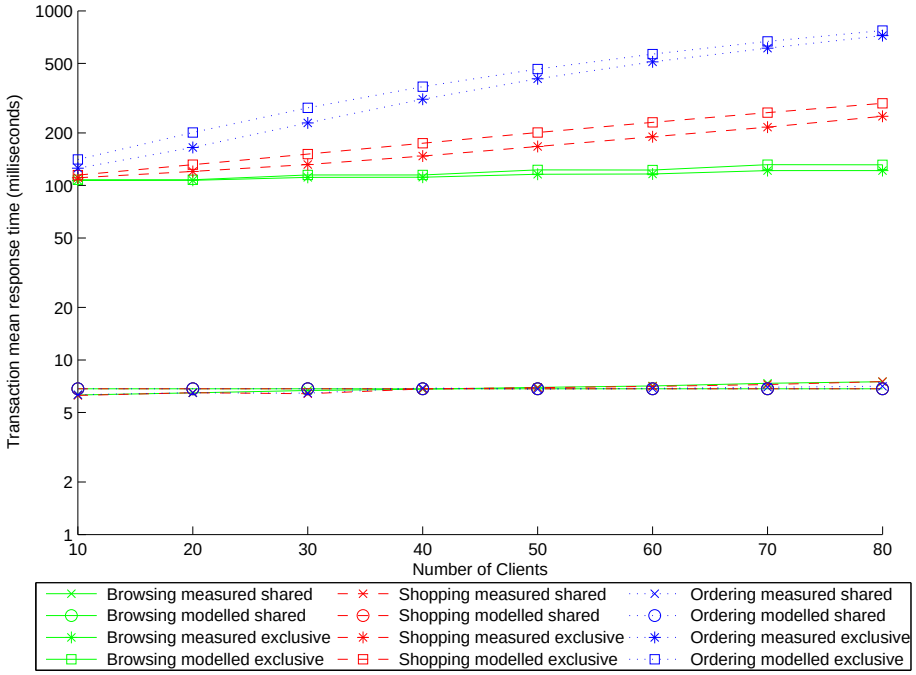


Fig. 8. Mean response time for TPC-W workload for multi-version 2PL.

Results. In this scenario, the shared transactions run through the system unaffected by the exclusive transactions. This lowers the contention for locks in the system in comparison to table-level and row-level locking for the same number of

clients. From Fig. 8 and Table 4, the QPN model gives an excellent prediction of mean response times for the shared transaction for all workloads. As the model does not represent the processing effects on the shared transaction, it overestimates their performance for small number of clients and underestimates their performance for larger number of clients.

Table 4. Error percentages for multi-version 2PL.

	browsing error (%)		shopping error (%)		ordering error (%)	
# of clients	shared	exclusive	shared	exclusive	shared	exclusive
10	8.48	0.92	9.07	3.62	8.18	12.24
20	5.56	0.81	5.40	9.43	5.46	21.83
30	2.30	3.22	6.82	14.80	5.98	21.97
40	0.60	2.93	0.29	18.51	-1.11	17.83
50	-1.84	5.87	-1.51	20.23	0.32	13.67
60	-3.70	5.37	-2.98	21.01	-0.59	10.74
70	-6.99	8.18	-5.38	21.03	-1.92	9.31
80	-8.81	7.89	-9.00	18.79	-3.21	6.87

For the exclusive transaction, the model gives an excellent prediction for the browsing workload. For the shopping workload, the model gives accurate estimates of the exclusive transaction response times up to 20 clients. For 30 clients and above, the model underestimates the performance of the exclusive transaction. In this case, locking contention does not dominate the transaction execution time, in addition to the effect of the random release of locks in the QPN model leading to over-blocking. This also applies to the ordering workload for less than 30 clients. For 40 clients and above, when locking contention starts to dominate exclusive transaction execution, the model error rates start to decrease. Even though the accuracy of the model for all workloads is not similar to that of previous locking mechanisms, the QPN model closely follows the performance trend of the exclusive transaction with increasing number of clients.

6 Conclusions and Future Work

In this paper, we have presented a modular and flexible queueing Petri net model of a relational database system. We were able to model a case study database system using a high-level QPN model that was easily adapted to reflect different concurrency control mechanisms implemented in commercial DBMSs. These simple QPN models scaled for large number of clients and accurately predicted the trends of the shared and exclusive transactions. The results demonstrate that the QPN models were able to give accurate predictions of the mean response times of transactions for moderate and high contention workloads.

The QPN models presented were simple models of the database logical level that do not require detailed modelling of the underlying hardware architecture.

The models have the potential to be applied to logical layer database modelling to evaluate lock contention for systems in which hardware provisioning is sufficient. For such scenarios, these models are able to give indications of the ability of the DBMS, and therefore the database design, to cope with different workloads under different concurrency control schemes.

For low contention scenarios or when the bottleneck is at the physical hardware layer, a layered model in which logical and physical resources are represented would perhaps be more suitable. More realistic row access scenarios representing skew in data access and data distribution will need to be investigated. Moreover, as the QPN models have an intuitive structure that directly reflects the database design, they are a suitable candidate for an automated mapping tool between database system specifications and QPN models. These are issues for future work.

References

1. Bause, F.: Queueing Petri Nets–A Formalism for the Combined Qualitative and Quantitative Analysis of Systems. In: Fifth Intl Workshop Petri Nets and Performance Models (1993)
2. Coulden, D., Osman, R., Knottenbelt, W.J.: Performance Modelling of Database Contention using Queueing Petri Nets. In: 4th ACM/SPEC International Conference on Performance Engineeering (2013)
3. Kounev, S., Spinner, S.: QPME 2.0 User's Guide. Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany (2011), http://descartes.ipd.kit.edu/fileadmin/user_upload/descartes/QPME/QPME-UsersGuide.pdf
4. Menasce, D.A.: TPC-W: a benchmark for e-commerce. IEEE Internet Computing, 6, 3 (2002) 83–87
5. Microsoft Corporation: Set Transaction Isolution Level (Transact-SQL). (2013), http://msdn.microsoft.com/en-us/library/ms173763%28v=sql.110%29.aspx
6. Olofson, C.W.: Worldwide Relational Database Management Systems 2012–2016 Forecast. International Data Corporation, Doc # 236273. (2012) www.idc.com
7. Oracle Corporation: MySQL 5.6 Reference Manual. Internal Locking Methods. (2013), http://dev.mysql.com/doc/refman/5.6/en/internal-locking.html
8. Osman, R., Awan, I., Woodward, M. E.: QuePED: Revisiting Queueing Networks for the Performance Evaluation of Database Designs. Simulation Modelling Practice and Theory, 19, 1 (2011) 251–270
9. Osman, R., Knottenbelt, W.J.: Database System Performance Evaluation Models: A Survey. Performance Evaluation. 69, 10 (2012) 471–493
10. Ramakrishnan, R., Gehrke, J.: Database management systems. McGraw-Hill, Boston, Mass. (2003)
11. Silberschatz, A., Korth, H.F., Sudarshan, S.: Database System Concepts. McGraw-Hill (2011)
12. The PostgreSQL Global Development Group: PostgreSQL 9.0.12 Documentation. (2012), http://www.postgresql.org/docs/9.0/static/index.html
13. The Transaction Processing Performance Council: TPC-W BENCHMARK version 2. (2003), http://www.tpc.org/tpcw/