OULUN YLIOPISTO
UNIVERSITY of OULU

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Pekka Ruottinen

DATABASE ARCHITECTURE DESIGN AND PERFORMANCE EVALUATION FOR QUALITY OF SERVICE MEASUREMENTS

Master's Thesis
Degree Programme in Computer Science and Engineering
January 2015

Ruottinen P. (2015) Database architecture design and performance evaluation for Quality of Service measurements. University of Oulu, Department of Computer Science and Engineering. Master's Thesis, 54 p.

ABSTRACT

Measuring and analysing the quality of experience and the quality of service are important assisting methods when optimizing network capacity, and providing the best possible user experience. They also present methods for finding and dealing with problems caused by the ever-growing internet data traffic and its number of users. Qosmet, developed at VTT Technical Research Centre of Finland Ltd, is a powerful solution for this type of network analysis. Considering that long term storage of quality of service measurement data is all numerical, this enables an efficient and well optimised database solution, one that performs well in either writing data or reading it for analysis.

The purpose of this thesis is to ascertain the optimal database architecture for quality of service measurements. An important outcome of this project is the interface between Qosmet and the implemented database. Part of the work consists also in identifying the characteristics for database optimization. In order to achieve these goals, three database architecture schemas were designed and analysed with the help of exhaustive writing and reading performance tests. Schema 1 has one, schema 2 two and schema 3 have respectively three database tables, in which the data is distributed according to its assumed frequency of use.

Following the analysis and comparison of these three database architecture schemas, the one with two tables appeared to be the best one. It achieved writing throughput of around 7,500 lines per second, and the reading performance was the second best, actually very close to the best performance. Schemas 1 and 2 were about 33 percent faster than schema 3 in writing performance. Schemas 2 and 3 performed about 20 percent better than schema 1 in reading tests. Performance analysis gave a good understanding of database optimization and features affecting its efficiency, as well as hardware requirements, bottlenecks and limits on high utilization situations. This work resulted in an efficient and verified database, which can be used in the future as part of the Qosmet solution.

Key words: database, DBMS, MySQL, optimization, Qosmet, quality of service.

Ruottinen P. (2015) Tietokanta-arkkitehtuurin suunnittelu ja suorituskykyarviointi palvelunlaadun mittaustuloksille. Oulun yliopisto, tietotekniikan osasto. Diplomityö, 54 s.

TIIVISTELMÄ

Tietoliikenneverkkojen palvelunlaadun ja käyttäjien kokeman palvelunlaadun mittaaminen ja analysointi tarjoavat apukeinoja verkon kapasiteetin optimointiin ja parhaan mahdollisen käyttökokemuksen turvaamiseen. Verkon ominaisuuksien mittaaminen ja analysointi auttavat myös löytämään ja ratkaisemaan kasvavien käyttäjämäärien sekä tiedonsiirtomäärien tuomia ongelmia. Teknologian tutkimuskeskus VTT Oy:ssä kehitetty mittausjärjestelmä Qosmet tarjoaa monipuolisen ratkaisun ja tehokkaan ympäristön tämän kaltaiseen verkkoanalysointiin. Palvelunlaatumittauksista saatava tieto on kokonaan numeerista tietoa, jonka tietokantakäsittely on tehokasta ja hyvin optimoitavissa niin luku- kuin kirjoitustapauksissakin.

Tämän diplomityön tavoitteena on löytää suorituskykyinen tietokanta-arkkitehtuuri palvelunlaatumittauksista saatavalle tiedolle ja toteuttaa rajapinta Qosmetin ja tutkittavan tietokannan välille. Osatavoitteena on myös tunnistaa tietokantaoptimoinnin mahdollistavat erityispiirteet. Näiden tavoitteiden saavuttamiseksi suunniteltiin kolme tietokantamallia, joille suoritettiin kattavat suorituskykyä mittaavat kirjoitus- ja lukutestit. Malli 1 sisältää yhden, malli 2 kaksi ja malli 3 vastaavasti kolme tietokantataulua. Tieto on jaettu tauluihin oletettujen esiintymistiheyksien perusteella.

Tietokantamallien suorituskykymittausten analysoinnin ja vertailun perusteella parhaaksi malliksi osoittautui kahden taulun malli (malli 2). Se saavutti kirjoitusnopeudeksi parhaimmillaan noin 7500 riviä sekunnissa ja ylsi lähelle parasta saavutettua lukunopeutta. Mallit 1 ja 2 olivat kirjoitusnopeudessa noin 33 prosenttia mallia 3 nopeampia. Mallit 2 ja 3 puolestaan suoriutuivat lukutesteissä noin 20 prosenttia mallia 1 paremmin. Suorituskykyanalysoinnin avulla saatiin hyvä ymmärrys tietokannan optimoinnista, tehokkuuteen vaikuttavista ominaisuuksista, laitteistovaatimuksista, järjestelmän pullonkauloista sekä korkealla käyttöasteella vastaan tulevista rajoituksista. Tämän työn tuloksena syntyi suorituskykyinen tietokanta, jota voidaan jatkossa käyttää Qosmet-järjestelmän osana.

Avainsanat: DBMS, MySQL, optimointi, palvelunlaatu, Qosmet, tietokanta.

TABLE OF CONTENTS

ABSTRACT
TIIVISTELMÄ
TABLE OF CONTENTS
PREFACE
ABBREVIATIONS
1. INTRODUCTION 7
2. DATABASE DESIGN WORK 9
    2.1. MySQL 9
    2.2. MariaDB 10
    2.3. NoSQL 10
        2.3.1. Hadoop/HBase 10
        2.3.2. MongoDB 11
    2.4. Other Databases 12
3. SOFTWARE DEVELOPMENT WORK 13
    3.1. Qosmet Solution 13
    3.2. Database Writing Module 14
    3.3. Database Requirements 16
    3.4. Database Selection 17
    3.5. Database Design 17
4. BENCHMARKING 21
    4.1. Test Plan and Performance Metrics 21
    4.2. Reading Queries 22
    4.3. Creating Test Data 23
    4.4. Analysis Methods 24
5. DATABASE PERFORMANCE VERIFICATION 25
    5.1. Writing Performance 25
        5.1.1. Write Throughput 25
        5.1.2. CPU, I/O and Memory Performance 27
    5.2. Reading Performance 31
        5.2.1. Query 1 31
        5.2.2. Query 2 35
        5.2.3. Query 3 37
        5.2.4. Query 4 42
        5.2.5. Query 5 45
    5.3. Choosing the Best Database Model 46
    5.4. Performance with the Writing Module 48
6. SUMMARY & CONCLUSIONS 51
7. REFERENCES 53

PREFACE

This thesis was done for the University of Oulu, Finland at VTT Technical Research Centre of Finland Ltd in Network Performance team. The main part of the work was carried out in the Quality of Experience Estimators in Networks (QuEEN) project. In addition the following projects supported the work: EIT ICT Labs activity Networks for Future Media Distribution (NFMD) and Next generation over-the-top multimedia services (NOTTS).

The goals of this thesis have been to ascertain the optimal database architecture for quality of service measurements, implementation of the interface between Qosmet solution and designed database and learning about the characteristics for database optimization.

I want to thank my technical supervisor Dr. Jarmo Prokkola for his patient support and guidance throughout the entire thesis process. I am grateful for the thesis supervisor Professor Marcos Katz and second examiner Professor Juha Röning for their support in the project. I would also like thank my colleague Toni Mäki for guidance with software development and technical help with the thesis. I also thank my colleague Jukka-Pekka Laulajainen for reading and commenting the thesis. I express my gratitude to every co-worker at K-wing of VTT Oulu and beyond. Without the smiles and laughter, the days at the office would have certainly felt much longer.

I thank my parents Leena and Pentti, for being always there for me and being interested in my studies and activities. All my friends, thank you for sticking around. My biggest thanks go to my dear wife Maria, who has patiently supported me during all these years. You have bared with me all the ups and downs. My older son Samu has shown unbelievable understanding and empathy for his young age. Younger son Heikki has lit up the room with his big smile and given me the energy for the final sprint.

Oulu, January 12th, 2015

Pekka Ruottinen

ABBREVIATIONS

CPU	Central Processing Unit
D-ITG	Distributed Internet Traffic Generator
db4o	Database for objects
DBMS	Database Management System
GPL	GNU General Public License
HDD	Hard Disk Drive
HDFS	Hadoop Distributed File System
I/O	Input/Output
NFMD	Networks for Future Media Distribution
NIC	Network Interface Controller
NoSQL	Not only SQL
NOTTS	Next Generation Over-The-Top Multimedia Services
ODBMS	Object Oriented Database Management Systems
ORDBMS	Object-Relational Database Management System
OS	Operating System
QMCP	QoS Measurement Control Protocol
QoE	Quality of Experience
QoS	Quality of Service
QuEEN	Quality of Experience Estimators in Networks
RAID	Redundant Array of Independent Disks
RAM	Random-access Memory
RDBMS	Relational Database Management System
SAS	Serial Attached SCSI
SQL	Structured Query Language
TPS	Transactions Per Second
UDP	User Datagram Protocol
VTT	VTT Technical Research Centre of Finland Ltd.
YARN	Yet Another Resource Negotiator

1. INTRODUCTION

The internet has become a part of everyday life for the majority of people living in modern first world society. In fact, more than 40% of the world population has an internet connectivity in year 2014 [1], and the number of devices connected to the internet is estimated to be double the global population in 2018 [2]. Annual global traffic is predicted to reach the zettabyte (10^{21}B) mark in 2016 and nearly triple the amount of the year 2013 when it was around 600 exabytes [2]. Video-focused services grow along with total traffic and they are estimated to take almost 80% share of all consumer internet traffic in 2018 [2]. Whether it is a video, voice over IP or some other real time-demanding service, consumers expect seamless and interference free user experience. Internet service providers and operators are expected to ensure this experience and the quality of service (QoS) for users.

In order to fulfil these expectations within the present day network architectures, certain QoS mechanisms and practices to monitor it have being introduced. There are several approaches to monitoring the QoS. One viewpoint is to divide the monitoring techniques into active and passive measurements. In active measurements, test packets are created in the network in order to discover the performance metrics. This approach does not give an entirely clear view since it adds extra traffic to the network, but on the other hand, it gives control over what is measured. Passive measurements offer an approach where network traffic is left untouched and only monitored as it passes by. The good thing about making passive measurements is that the traffic within the network is not increased and the actual traffic within the network is measured. The passive approach is especially good when trouble-shooting real user cases.

This work is affiliated with passive QoS measurements. The research team within VTT Technical Research Centre of Finland (VTT) has developed a tool called Qosmet, a passive QoS measurement tool [3]. Qosmet can measure a plethora of performance parameters such as delay, jitter, packet loss, connection break statistics, load, the volume of data and packet sizes. All these parameters are numerical information monitored in real time. The motivation for this work was to extend the versatile QoS measurement solution by adding a data storage and visualization implementation for non-real-time usage. Visualization is beyond the scope for this work, but data storage as well as the interface between the existing Qosmet solution and the database were designed and implemented.

The work was conducted as part of the Quality of Experience Estimators in Networks (QuEEN) project. The project is part of a European research initiative called Celtic-Plus. Also, financing the work were EIT ICT Labs activity Networks for Future Media Distribution (NFMD) and Celtic-Plus project Next generation over-the-top multimedia services (NOTTS). The QuEEN project aims at producing Quality of Experience (QoE) estimator agents. The estimator agents use models based on human perception and network QoS metrics, and communicate through the network with each other in order to adapt to changing network situations. Qosmet offers a versatile solution for QoS monitoring, and the developed database addition gives an option to store easily large amounts of measured QoS and QoE data from numerous measurements.

The data from Qosmet is all numerical and therefore makes the structure of the database quite unique. Processing numerical data is faster than, for example, handling string-based data. The major part of this work is to ascertain the optimal database architecture for this unique data set and to detect the characteristics for database

optimization. In order to achieve these goals, three different database architecture schemas were designed according to the base requirements set for the database. Exhaustive writing and reading performance tests were performed for all three schemas and these are presented in detail in this thesis work. A working database implementation and an interface between the Qosmet solution and the database were carried out as part of the work. The process helped us to understand the demands and possible points of improvement of a database solution for the QoS measurement data.

The chapters following aim to provide a detailed overview of what was done during the work and what kind of results and findings were discovered. Chapter 2 examines different database management systems (DBMS) and gives the necessary background information to support the decision for the choice of DBMS to be used. Chapter 3 is full of details about designing the interface between the Qosmet solution and the database as well as database requirements, selection and schema design. Chapter 4 introduces the performance test environment and explains the plan for writing and reading tests. It also contains the information about analysis methods and details about test data creation principles. Chapter 5 is about test results for writing and reading performance. A detailed analysis of writing and reading performance is examined before choosing the best database schema out of the three that were designed. Lastly, performance with the developed database interface is studied. Chapter 6 concludes the work by summarizing the research and taking a look at possible future development.

2. DATABASE DESIGN WORK

For a database system there are two distinctive metrics for performance; throughput and response time [4 p.249]. Throughput is measured as transactions per second, and response time expresses how long a single transaction takes from beginning to end. Many widely used services that interact with people, such as Web-based applications, rely on a good database response time. However, systems with a large number of transactions need to focus on high throughput. Knowing what type of queries are the most common can be helpful in designing a high performance database solution. For example, queries that have to join information from several tables require more resources than queries without joins. Indices that include information about the mostly frequently used queries are an option to speed up the reading transactions, but they have a downside of slowing down the database updates, because extra work is required to keep them up during the writing process.

In this section, a comparison between a selection of database solutions is made. First, MySQL, a widely used relational database management system (RDBMS) and its features are introduced. The relational model is one approach to data modelling. Other approaches include object-oriented, entity-relationship, hierarchical and network models. The relational model uses a single way to represent the data. Everything is viewed as a two-dimensional table in which data is stored [5 p.61, 131]. A new open source alternative to MySQL, MariaDB, is also surveyed as another relational database choice. Next, two NoSQL (Not only SQL) solutions, the Apache Hadoop, a software framework for building large, distributed database applications, and MongoDB, are examined.

2.1. MySQL

MySQL [6] is one of the most widely used database management systems. It was fully open source until 2008, when Sun Microsystems acquired it. Later, in 2010, Sun Microsystems was acquired by Oracle Corporation. It is still available under GNU General Public License (GPL), but also under commercial options. MySQL is a widely used RDBMS that has been around for almost 20 years and has become very popular with many web applications. It is also extensively tested and researched. There are plenty of tools available for developing, testing and administration, and both commercial and pier support from forums and guide books are easily obtainable. One of the big advantages of MySQL is that it is available for many different platforms, and numerous tools have support for the system.

MySQL has a rich feature set, including support for e.g. indexing, foreign keys, replication, partitioning and query caching. MySQL is also designed to take advantage of multiple CPUs and is a fully multi-threaded software. The database can serve thousands of tables and billions of records of data. MySQL also supports several storage engines. A database engine or storage engine is a component that MySQL uses in order to handle the reading, writing and organizing of the data in the database. The transactional database has a failsafe mechanism to come back from, for example, a power failure. If the write transaction is not completed properly, there is a way to get the data back. MySQL supports many different transactional and non-transactional databases, such as InnoDB and MyISAM. MyISAM was the default storage engine for MySQL until version 5.5.5 and the most frequently used in common applications, but

lacks transaction safety. InnoDB is the default engine from the MySQL version 5.5.5 and, being transactional, provides more safety for the data. InnoDB also supports foreign keys which uniquely connect two tables together within a database.

2.2. MariaDB

MariaDB [7] is a new open source alternative to the popular MySQL RDBMS. It was designed by the founder of MySQL, Michael Widenius, after selling MySQL to Sun Microsystems and having doubts about MySQL's future openness [8]. Even though MySQL still continues to be available with a free software license, MariaDB has gained a lot of interest among developers. For the most part, MariaDB is compatible with MySQL. Until the recent versions, the new features from MySQL are also ported to MariaDB.

MariaDB is still a fairly new solution, but as it is forked from MySQL, it includes all the same functionalities. The open source community has already developed new features and improved some of the old ones. Even though it is not as widely used or supported as MySQL, it is a strong competitor as a fully open source alternative to MySQL.

2.3. NoSQL

Even though RDBMS's are the dominant solutions for data storage, they have many shortcomings for present-day applications. Using an RDBMS for storing graphs into tables is an example of the misuse of a data model [9]. Many web applications contain features that can benefit from simpler and more flexible database systems. Data storages containing huge amount of data have to be available for reading and writing with low latency. They have to provide high performance, high availability, be easily replicable and provide a mechanism for load balancing over multiple servers [9]. These aspects have motivated research and development for new kinds of solutions, often called NoSQL. NoSQL is not a total opposite to RDBMS, and can be characterized a distributed solution which may not require fixed table schemas, which usually avoids join operations, typically scales horizontally, does not expose a SQL interface and may be open source [10]. In general, the term NoSQL is used to describe solutions that try to find an alternative way to solve the problems that a traditional relational database does not resolve well [11]. Basically, all the NoSQL solutions are designed to meet some special need and are hence difficult to be categorised for solving some general challenge or choosing only one that is suitable for a certain use case.

2.3.1. Hadoop/HBase

The Apache Hadoop [12] is a full framework designed to overcome many of the challenges described above. Hadoop is developed as a top-level Apache open-source project, Yahoo! being the largest contributor. Some of the users together with Yahoo! are eBay, Amazon, IBM, New York Times, Google and Facebook, having deployed clusters with more than 1,000 machines and several petabytes of data [13]. Common uses for Hadoop clusters are high-speed data mining applications, log analysis and machine learning.

Hadoop is designed to be a highly scalable system which can work from one server setup to thousands of servers. It is designed to deliver high-availability and reliability at the application layer instead of relying on hardware. Hadoop itself is not a database solution and one can run basically any DBMS on it. Apache has two popular NoSQL projects; Cassandra [14] and HBase [15], but a traditional MySQL is also an option. These three solutions were compared in [10], and it was shown that the features between the three are very similar, but the capabilities and performance favours the NoSQL solutions in case of big data sets.

Hadoop cosists of four main parts; the Hadoop Common, Hadoop Distributed File System (HDFS), Hadoop YARN and MapReduce engine. Hadoop Common is the main part of the Hadoop system, providing the access to HDFS or other file systems supported. It contains the files and scripts needed to start the system. HDFS is a file system designed to store very large data sets with high availability and reliability. It is also able to distribute this information at high bandwidth to user applications. Hadoop YARN is a framework inside the Hadoop framework itself. YARN's role in the Hadoop environment is job scheduling and resource management. MapReduce is a Java-based programming model for parallel processing of the data. The main idea in MapReduce is to split a problem into smaller independent sub-problems and then solve them in parallel by using several servers.

Apache HBase is a NoSQL database for Hadoop built on top of HDFS. It is an open source project for very large data sets; designed to hold billions of rows and millions of columns of data. HBase is a distributed, fault-tolerant and scalable solution.

It is good to keep in mind that HDFS and, therefore, the whole Hadoop system is designed and built for a write once, read many times pattern and, therefore, the writing operations in Hadoop are expensive. A performance evaluation study [16] between HBase and MySQL on top of Hadoop/HDFS was conducted, and it was noticed that, with a small number of concurrent users, there are no differences between the two. However, with more users and more load on the system, the HBase solution gained some advantage over the MySQL.

2.3.2. MongoDB

There are many ways to characterize NoSQL database solutions. One way to categorize Hadoop/HBase is to define it among column-oriented stores [9]. MongoDB [17], a choice from another group, documents stores and represents a solution where key value pairs are stored within so-called documents. The choice of MangoDB for this comparison was strongly influenced by its characterization in a different NoSQL group than HBase. This gives more variation in compared features.

MongoDB is an open-source schema-less solution with full index support. It supports map-reduce and provides high performance, high availability and easy scalability. MongoDB database contains one or multiple collections, which consist of one or many documents. The documents can be created on the fly and the records within them can have different numbers of attributes. The attributes can be any data types; numbers, strings, dates, arrays or sub-documents. As presented in [18], both, writing and reading speed with large data sets is much faster for MongoDB compared to MySQL. This is largely thanks to the fact that MongoDB is schema-less and the reading queries do not involve any joins from different tables. It is also noticeable that MongoDB might not be the right choice when dealing with a small amount of data, as inserting can take a long time with small data sets [19].

2.4. Other Databases

Object-oriented database management systems (ODBMS) such as databases for objects (db4o) [20] offer an alternative to more traditional RDBMS's. ODBMS use object-oriented approach whereas RDBMS's are table-oriented. db4o is an open source object database built for Java and .NET developers. In a comparison [21] between db4o and MySQL, it was noticed that inserting large amounts of new data was significantly faster with db4o. On the other hand, MySQL was much faster when selecting or deleting data from the database.

PostgreSQL is an example of an object-relational database management system (ORDBMS). The ORDBMS contains a relational database and hence is easy to understand for someone used to using relational databases. In addition to plain RDBMS, it has an object-oriented database model, and it supports objects, classes and inheritance in database schemas and queries. In other words, the data is fetched and manipulated from the database with object-oriented query language. Although PostgreSQL has some advantages over MySQL on the query usage, a large amount of research comparing the two [22] shows, that MySQL is better in performance.

3. SOFTWARE DEVELOPMENT WORK

Qosmet is a solution for passive QoS measurements developed at the VTT [3]. The goal of this work is to develop a database implementation for Qosmet and an interface between the database and Qosmet. Qosmet is an established solution, but it lacks data storage functionality. QosmetService, introduced in detail below, is capable of forming a connection to 3rd party software and relaying the measurement data there. Within this work a database writing module has been developed which receives the measurement data from Qosmet and stores it in a database.

3.1. Qosmet Solution

The Qosmet solution is capable of performing versatile QoS measurements and also evaluating the Quality of Experience (QoE) for some applications. The results can be viewed in real time and/or stored for analysing later. This thesis is focused on the latter of the two use cases. As the measurements are passive, the overheads are minimized and therefore the effects on the network performance from measurements themselves are minimal. The software solution is implemented in a way that light-weight software agents, called Qosmet Service, can be deployed in a large number of network nodes for stand by and taken into use when needed. One Qosmet Service can be used for several independent measurements.

Basic Qosmet architecture is presented in Figure 3.1. Qosmet Services are located in one or more network nodes, one serving as the primary measurement point. Qosmet Service instances communicate using the QoS Measurement Control Protocol (QMCP), which allows full remote control over the measurements. The application data traffic is not influenced by the QMCP control stream, and the application itself is completely unaware of the Qosmet measurement. The primary Qosmet Service can be set up to send the measurement data to a 3rd party software.

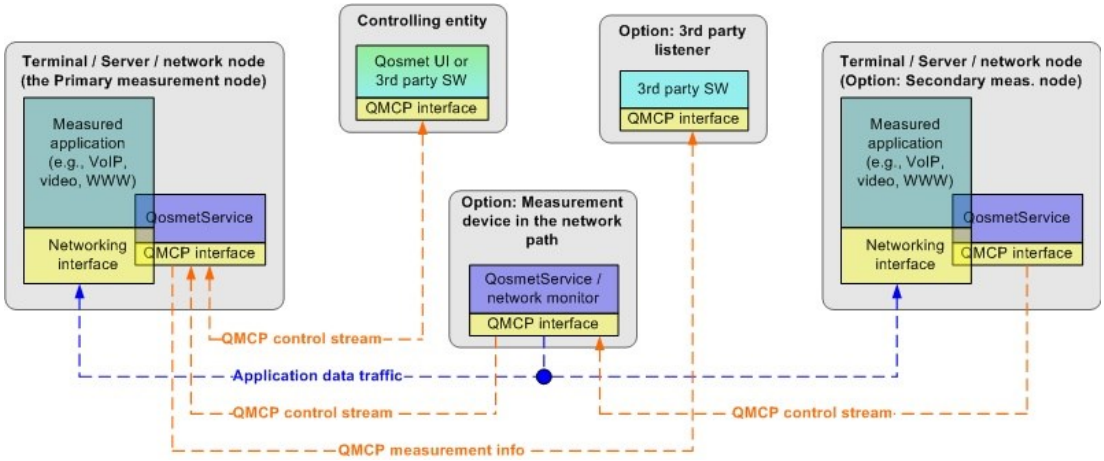

Figure 3.1. Qosmet Architecture.

The measurement data includes a plethora of performance parameters such as delay, jitter, packet loss, connection break statistics, load, the volume of data and packet sizes. A single point measurement provides some of the basic performance parameters,

whereas a two point measurement can form a full set of QoS results. The results can be produced as average values over a certain time or accurate per packet values.

3.2. Database Writing Module

The database writing module called QosmetDbWriter is independent software developed within the scope of this thesis. This command line tool works as a 3rd party listener presented in Figure 3.1 above. QosmetService forms a connection with QosmetDbWriter, which then listens for incoming measurement results. The measurement data is then processed and written to a database by means of appropriate SQL statements. The software keeps track of different measurements and is able to be connected with several QosmetServices simultaneously. A component diagram for the system is shown in Figure 3.2 below. At the top we can see the Qosmet package containing the QosmetService component taking care of the actual measurements. One or many QosmetServices can be connected to a link called QmcpServer on a QosmetDbWriter. QosmetService is a part of Qosmet core services and was developed outside this work. The database writer, QosmetDbWriter is connected to a DBMS via link called MySQL Connector/C++. The C++ connector is a C++ interface for communicating with MySQL servers and is provided by MySQL/Oracle.

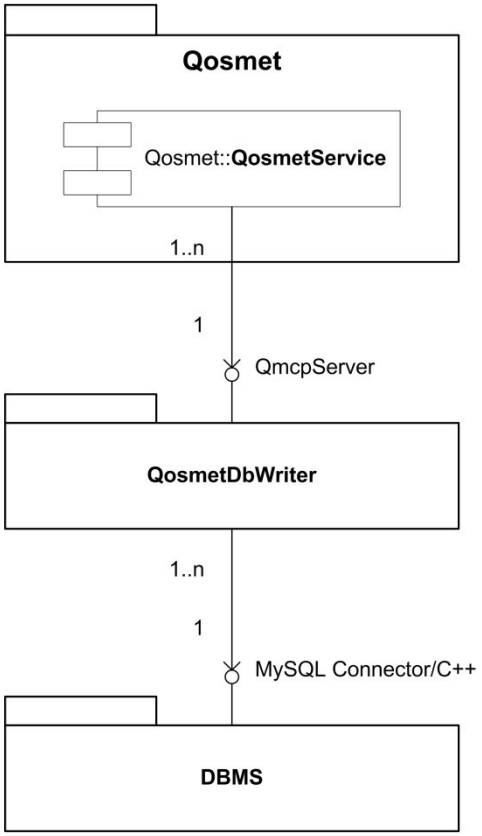

Figure 3.2. Component diagram for the database writing module QosmetDbWriter developed within the project.

The class diagram for the database writing module is presented in Figure 3.3. QosmetDbWriter is the main class of the software, and it uses the QmServer class functions from the main Qosmet package for communication with QosmetService. The internal structure of the QosmetDbWriter is straightforward. In addition to QmServer class, the QosmetDbWriter class is associated with the QosmetDbManager class. The QosmetDbWriter class is responsible for the basic user interface and keeping up the processing thread for incoming measurement results from QosmetService.

The QosmetDbManager class takes care of different kind of measurement results; starting, ending or normal results. It is associated with two additional classes, QmcpToSqlMapper and QosmetDb. QmcpToSqlMapper contains utility functions to create SQL queries from QMCP messages. These functions could also reside in the QomsetDbManager class, but for clarity reasons they are in a separate class. QosmetDb is an interface for actual database interaction. QosmetDbWriter is designed with a modular principle, and it is not tied to any particular database solution. Any database (providing a suitable API) can be integrated into QosmetDbWriter, by creating a database-specific adapter implementing the QosmetDb interface. As MySQL is used in this work, a class for MySQL database interaction is a part of the implemented solution. For demonstration purposes, another class for database communication is also presented. QosmetODBCDb class would serve the purpose of an ODBC interface, but is not implemented within this work. Both, QosmetMySQLDb and QosmetODBCDb classes inherit the QosmetDb class.

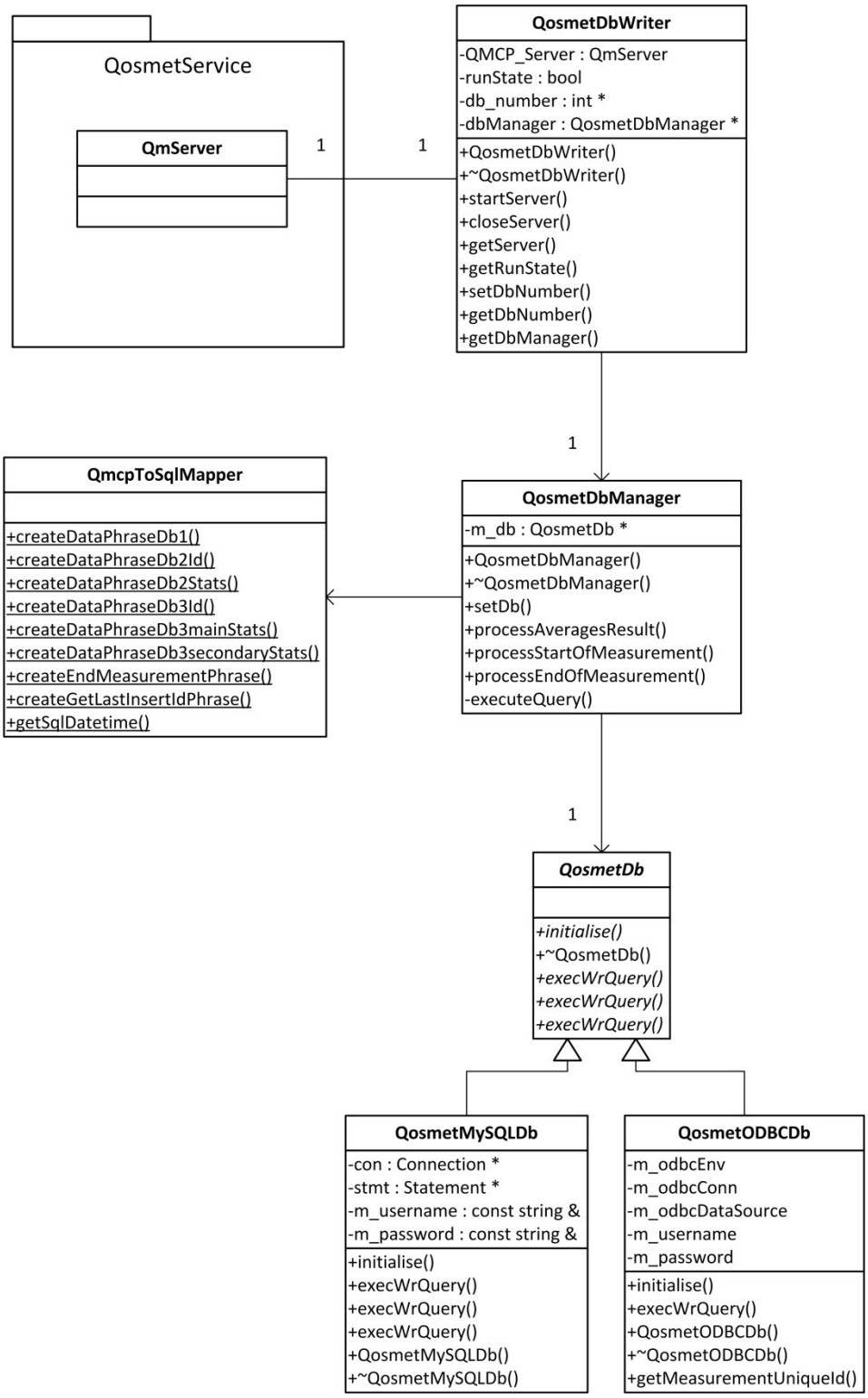

Figure 3.3. Class diagram for the database writing module QosmetDbWriter developed within the project.

3.3. Database Requirements

There are some requirements that need to be fulfilled in the database solution to be chosen. The database is designed to meet the possible situation where there are

hundreds or even thousands of clients sending network measurement data at one second intervals, on average. The speed of reading and writing of data to the database are both important, but the real time writing is the main requirement. To be able to provide extensive analysis of the data in a reasonable time frame, it is also important to have a good reading performance, but there are more ways to overcome the shortcomings of that feature than with writing.

The Qosmet measurement data and the Qosmet measurement ID have a certain number of attributes to be stored in a database. It is defined that 64 attributes is enough for the present-day situation and possible later additions. In case of relational databases, we are talking about 64 columns in a table which is not a restriction on examined solutions. It is also important to have a solution that is adaptable to a future situation. For example, it has to be possible to add more attributes later without having to implement a new database solution. In theory, the number of records can grow indefinitely as time goes by, but the database should be able to handle at least tens of millions of records. Adding more and more records will slow down the writing, as indices need to be updated, but the reading performance will suffer more.

In addition to the functional requirements, the solution chosen needs to be easily available for commercial production use. The installation process needs to be straightforward and possibly to have an option to be automated, as the database might have to be installed in different locations, for example on the client's premises.

3.4. Database Selection

There are differences between different systems in how fast they are on different kind of data sets and how well they fulfil the above described requirements. However, looking at the features of the different database management systems examined, it is clear that all the critical requirements are met by each of them. Writing and reading performance, number of attributes, number of stored records and scalability of the database are all fairly well covered and satisfied by the MySQL, MariaDB, MongoDB and Hadoop. Some of the solutions greatly exceed the requirements and have more features than actually needed in this case. For example Hadoop is intended for very large data sets and deploying it for a small solution can be seen as a waste of resources.

The most important decision criterion in this case is the last mentioned ease of installation and support. In the research group where the work was performed, the experience of new kinds of DBMSs is very limited or non-existent, whereas MySQL is a familiar solution. The experience and the knowhow of MySQL within the research group enabled sufficient support for the thesis work and for future development and maintenance of the database solution. Therefore, MySQL was selected as the database solution to be used in this project.

3.5. Database Design

The measurement ID consists of 13 parameters and the actual measurement results of 48 parameters. The measurement ID part is the same for all individual results within one measurement. Having all 13 ID parameters present makes it possible to ensure globally the differentiation of one measurement from another. Not all result parameters have non-zero values in every measurement as single point measurements do not provide thorough data. Also, there are a number of parameters that have been noticed

to be more useful in network analysis than others, such as delay, jitter, packet loss, load and throughput.

Three different approaches were chosen for the database design. In the first schema represented in Figure 3.4, all measurement result parameters are stored in one table. Using only one table results in good writing performance, because there is no need for structure or extra parameters to link the tables. On the other hand, the reading performance can degrade significantly as the table size grows.

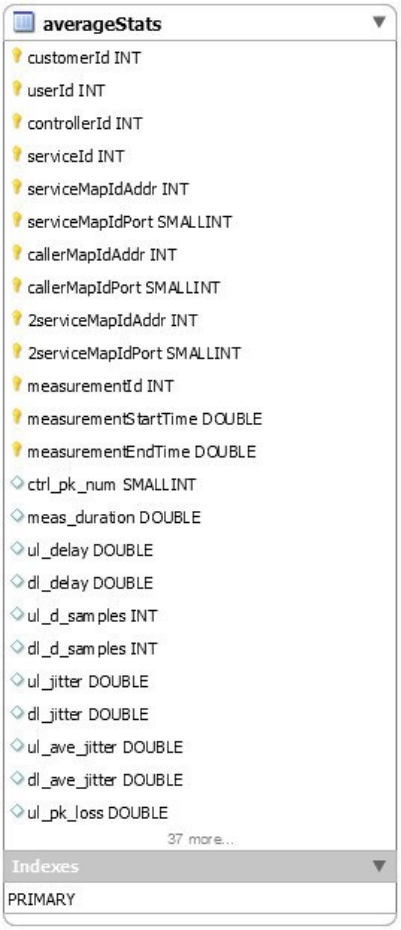

Figure 3.4. Database schema 1. The name of the table is at the top and the first few parameters are listed with their data types. Database index (primary key marked with yellow) is listed at the bottom.

The second schema, represented in Figure 3.5, introduces an approach where the measurement ID is in one table and the results in another. This approach saves unnecessary repetition of data as the measurement ID is the same within one measurement. In the second approach, the measurement ID is stored once to the database and updated once when the measurement ends. With long measurements, this approach could save thousands of database accesses. The downsides of this approach are the need of extra parameters when using the MySQL key structure and degraded writing performance when using several tables. The need for extra parameters is much smaller, however, than the benefit gained from the amount of data saved from measurement IDs.

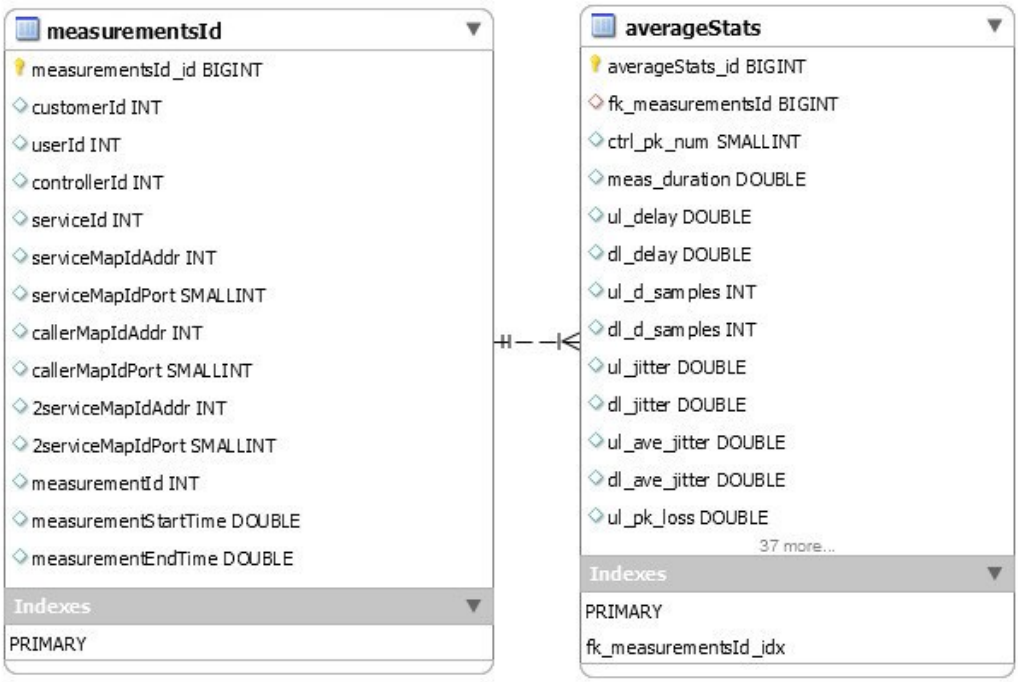

Figure 3.5. Database schema 2. The name of the table is at the top and all parameters from measurementsId and the first few from averageStats are listed with their data types. Database indexes (primary index marked with yellow) are listed at the bottom. The information between the tables is linked with MySQL's foreign key relationship.

The third schema, seen in Figure 3.6, takes the approach from the second schema a little further. The measurement ID is in one table as it was in previous case, but the result parameters are now separated into two different tables. What are thought to be the most used parameters in the analysis process are in one table called the mainStats. The less used ones are in another table, called secondaryStats. This approach has the same advantages and disadvantages as the schema 2, when compared to the first one. Separating the measurement ID from the measurement data parameters reduces the amount of data stored. Dividing the data parameters on the other hand increases the amount of stored data, as additional parameters for table ID number and foreign key need to be added. This might cause decreased writing performance for the database as more data is written. The overall reading performance depends on the use case, as it is probably fairly quick to fetch information on the most common measurement parameters, but reading from both measurement data tables could be slower than in schema 2.

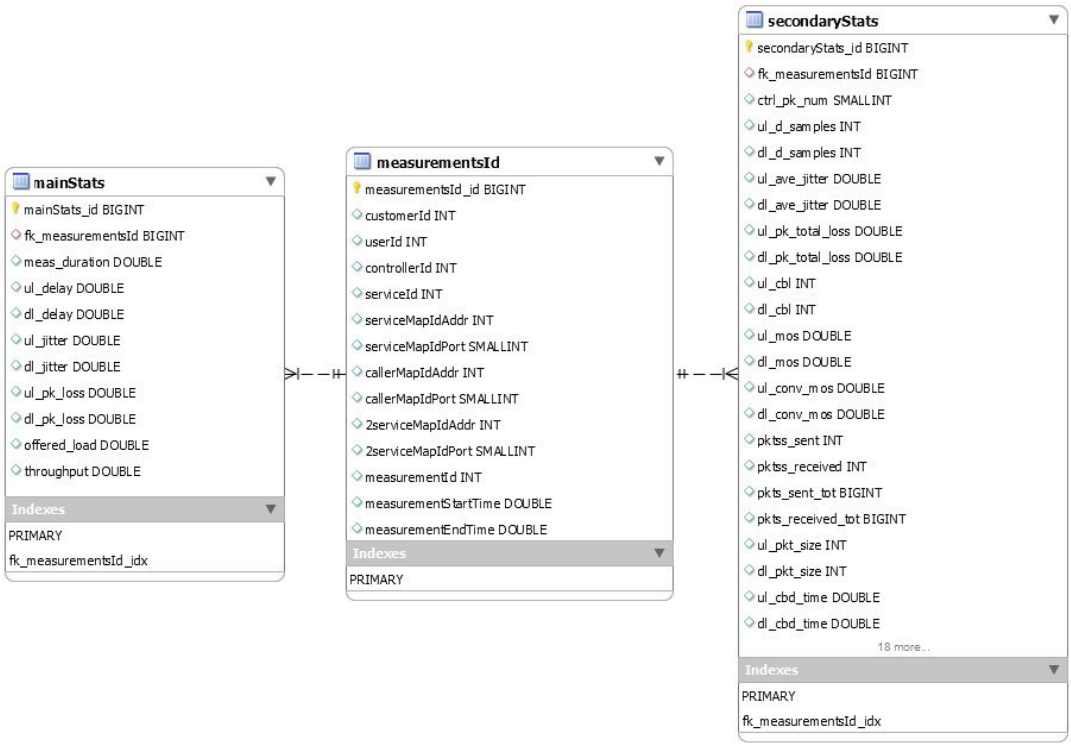

Figure 3.6. Database schema 3. The name of the table is at the top and all parameters from measurementsId and mainStats as well as first few from secondaryStats are listed with their data types. Database indexes (primary key marked with yellow) are listed at the bottom. The information between the tables is linked with MySQL's foreign key relationship.

4. BENCHMARKING

The database performance tests were performed on a server located at VTT's premises. The details about the hardware are listed in Table 4.1 below. The configuration included two quad core processors with hyper-threading enabled and 16 Gigabytes of RAM. Storage media consisted of three SAS (Serial Attached SCSI) connected hard drives combined together with RAID 5 (Redundant Array of Independent Disks) technology. 92 Gigabytes of disk space was allocated solely for the test environment. Network and network interface for the configuration were with Gigabit speed. The operating system was run on top of VMware virtualization software. The operating system installed was a Linux distribution Ubuntu with version 12.04 and MySQL Database with version 5.5. Even though the operating system was run on top of virtualization software, the hardware resources were dedicated solely to the database system during testing. Only the storage space was restricted to the above-mentioned 92 Gigabytes.

Table 4.1. Hardware specifications for the database server used in performance tests

Parameter	Specification	Comments
CPU	Intel E5520 @ 2.27 GHz, 8 cores, 2 sockets	Hyper-Threading supported.
RAM	16 GB	
HDD	3 x Seagate 300GB, SAS, 15000rpm, 3Gb/s	RAID5 (total capacity 553GB). 92GB for test environment.
NIC	Broadcom NetXtreme II 5709, Gigabit, Quad port	
OS	VMware ESXi, 4.0.0.0, 219382 Ubuntu 12.04	MySQL 5.5

4.1. Test Plan and Performance Metrics

The database performance was tested when writing QoS measurement data to the database and reading the data from the database. The reading tests are explained in detail in the two sections following, but the writing tests are fairly straightforward. Different numbers of writing threads, from one to 128, write the data to a database. The number of consecutive rows written is also increased from one to 10,000. Increasing the number of threads, we can see how the database and the database writing module developed handle large numbers of data sources and what the writing limits of the DBMS are. With different numbers of consecutive rows written, we emulate the different measurement sampling update intervals and also create enough traffic to seek the writing limits of the DBMS.

For writing, throughput is the analysed metric and for reading the metric is total execution time. Throughput measures how many lines per second the database has written. This is then viewed against the number of threads writing to a database. Total execution time indicates how many seconds it took for the entire query to perform per one client. Total execution time is viewed against the number of clients reading from the database.

4.2. Reading Queries

Two qualities were to be taken into account when queries for database reading performance tests were designed. First, the test queries needed to represent the real life use cases of a database for QoS measurement data. Secondly, all three designed database schemas needed to be tested thoroughly. Five different queries were designed and these are represented in Table 4.2. They are explained in more detail and represented with different values of x in Chapter 5. Queries are named and numbered from Query 1 to Query 5. The second column in the table shows which measurement parameters are used in each query to constrain the result set. The three database schemas compared in this work are presented in Section 3.5. All three schemas contain all the measurement parameters but these are distributed in one, two or three tables. In order to see the differences in performance when querying with different constraints, we need to focus the constraints to each table and to different combinations of tables. With queries and constraints covering all three tables in schema 3, we also cover the two tables in schema 2 and the one table in schema 1. The third column in Table 4.2 shows the different tables from schema three covered with appropriate queries. For instance, customer_id is found in table *measurementsId*, ul_delay in *mainStats*, etc. Detailed queries with different values of x are represented in Chapter 5.

Table 4.2. Reading performance test queries. Columns represent the query names, the result constraining measurement parameters and the appropriate tables for schema 3

Name	Query constraint	Db tables (in schema 3)
Query 1	customer_id = x	*measurementsId*
Query 2	ul_delay < x	*mainStats*
Query 3	ul_dupl < x	*secondaryStats*
Query 4	customer_id <=> x AND ul_delay < x	*measurementsId* and *mainStats*
Query 5	ul_delay < x AND ul_dupl <> x	*mainStats* and *secondaryStats*

Some possible variations of constraining parameters were omitted in order to keep the number of test cases reasonable. However, the five selected queries cover the main points of interest. Firstly, three queries fetch the data from each of the different tables in schema 3. Query 4 focuses on *measurementsId* table, whose parameters construct a MySQL primary key, and one of the other tables. The parameters within the primary key are indexed and hence prepared for fast query performance. Query 5 examines the situation when data constraining parameters are from two non-indexed tables; *mainStats* and *secondaryStats*. The query from *measurementsId* and *secondaryStats* was considered to be very similar to Query 4 and was omitted.

In a real life situation, one might search the database and retrieve all the data from one customer using the *customer_Id*. In another case, the data could be filtered according to delay and/or throughput e.g. using *ul_delay* and *throughput*. To get as close to real cases as possible, one should also take into account that some queries should return only few results and some queries fairly large amounts of data. The constraint parameters chosen are considered to be one of the most common parameters used in QoS analysis, and therefore the queries represent real life search queries fairly well.

During the reading tests, we came up against several MySQL features that resulted in new complementary queries. With Query 1, it was noticed that a *straight join* query used with schema 2 resulted in significantly faster query times than a regular join query. With schema 2, as well as with schema 3, the data is fetched from several tables and needs to be joined together before being delivered to the ordering process. In some cases, the optimizer within the MySQL can put the tables in wrong order. In these cases, it is possible to use straight join to force the desired handling order of the tables.

It was also noticed that MySQL has an option to enable or disable a parameter called query cache. With query cache enabled, some queries run very quickly. Differences with and without this option were compared with Query 3. The query cache size determines which queries it affects. With Query 3, a comparison was also made of how the results vary when the query returned data only from one measurement results table or from all the tables within the schema.

4.3. Creating Test Data

For reading performance tests, uniform data for all three database schemas needed to be created. The goal was that the data represents all the different real life use cases as well as possible. Here it is considered to have one very long test run, five fairly long ones, 50 short ones and 500 very short ones. These test runs are represented in Table 4.3. If the sampling time is one second, these test runs represent time scales from approximately 15 minutes to one week. The number of test runs also represents the value for *customerId* in the database. This means that, for customer ID 1 there are 600,000 rows in the database, for IDs 2-6 there are 86,400 rows for each and so on.

Table 4.3. Uniform data for reading performance tests

Number of test runs	Rows per run	Time scale represented (one sec update interval assumed)
1	600,000	1 week
5	86,400	24 hours
50	10,000	3 hours
500	1,000	15 minutes
556	**2,032,000**	-

All three database schemas are filled with the same amount of data where most of the data is random, but some variables are incremented to have uniform distribution within a test run. The uniform distribution is important when performing the reading tests. With this principle, it is possible to design such queries that return the desired number of rows. If all data were to be random, it would be impossible to emulate real life reading cases. This does raise one issue to consider though. Now that the data for some variables is not random, there is a change that it has effected on query speed when reading the data. It is possible that the reading operations are easier and thus faster than with all random data.

To be able to perform meaningful query cases, all database tables need to include at least one incremented variable. Schema 3 sets the requirements, since its three tables include all the same information as schemas 1 and 2 include in their instances. The

following variables are incremented in the reading test cases: from *measurementsId* table, *customer_Id.* From *mainStats* table, *ul_delay* and *dl_delay*. From *secondaryStats* table, *ul_dupl* and *dl_dupl.*

4.4. Analysis Methods

All test cases for database writing performance were performed for all three database schemas. 11 different cases with an increasing number of threads or writing clients, from one to 128, were performed. Threads were set to write the same number of lines per case, but each case were repeated to have nine cases with different numbers of lines written. Hence 99 test cases per schema were performed so as to determine the writing performance.

Writing to the database was performed using the developed database writing module with slight modifications. The software was modified to create a determined amount of random data for the database. The script starting the writing module gave the number of consecutive rows as a parameter. This means that writing one line per client started the module for each line, but when writing, for example, 100 lines per client, the client started once and wrote 100 lines and so on.

In order to have enough statistical accuracy, a minimum of five rounds of test runs were run for each test case. For some cases, more rounds were run to meet the set requirement. The sample mean average [23 p.1053] is

$$\bar{x}=\frac{\sum x_i}{n}, \tag{1}$$

where x_i is a value for single sample and n is the total number of samples was determined first. Now the standard deviation [23 p.1053]

$$s=\sqrt{\frac{\sum(x-\bar{x})^2}{n}} \tag{2}$$

was used to determine the standard error [24 p.46]

$$s_{\bar{x}}=\frac{s}{\sqrt{n}}. \tag{3}$$

The standard error is then divided by the average and this percentage needs to be less than 5%. In other words, each test was repeated at least five times, but as many times as necessary in order to meet the decision criteria of standard error falling below the set limit of 5%.

For the results, number of threads is multiplied by the number of rows written for each thread. This total number of written rows is divided by the average time taken to execute the entire test cycle. In the chapter following we use the measure lines/s to analyse the differences between database schemas. We will see how a different number of clients and different number of written lines per client affects performance.

5. DATABASE PERFORMANCE VERIFICATION

5.1. Writing Performance

5.1.1. Write Throughput

The following three figures represents the database write throughput against the number of threads. Different lines in the graphs, from one to 10,000, represent different numbers of consecutive rows written. The measured unit for throughput in these cases is *lines/s*.

The results for all test cases in database schema 1 are shown in Figure 5.1. Threads here represent clients that write data to the server. Examining the results for different numbers of threads, it can be seen that one or even two clients do not reach the maximum write throughput, and hence are not able to take full advantage of the server. According to these results, it takes eight clients writing simultaneously to the server to reach the maximum potential. We will also notice that, in order to take the full potential into use, it is not optimal to write only one line at a time per client. In order to reach the full potential for each client, they should write at least around 50 rows consecutively. In these tests, the writing software is started for each round separately and hence affects the measured write throughput. Naturally, it is more optimal to write many lines within one round or as it is in a real use case, during the continuous execution of the software. This means that in real life you would not restart the writing software between measurement cases.

For the cases writing 50 or more lines consecutively, the maximum performance is around 7,000 lines per second and is reached with eight clients. For cases where only one, five or ten rows are written, it looked as if the maximum performance was reached with eight or 16 clients, but test runs with 128 clients proved that assumption wrong. Unfortunately, it was not possible to run more tests with more clients. However, it is probable that the maximum performance is around seven to eight thousand lines per second, since this was clearly the limit reached with more consecutive lines written.

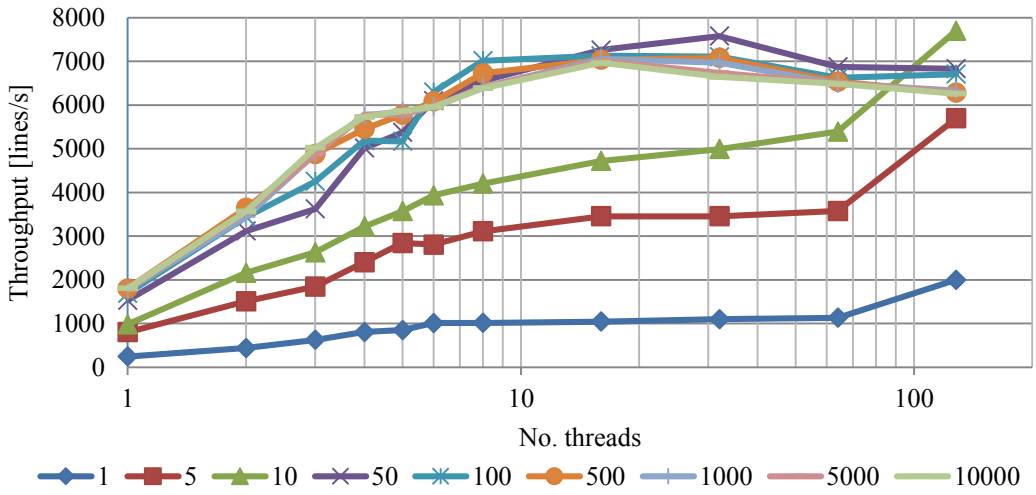

Figure 5.1. Write throughput against number of threads for database schema 1. Lines one to 10,000 represent different number of consecutive rows written.

Database schema 2 has two tables where results should be written. Figure 5.2 shows that results for schema 2 are very similar to the schema 1 results shown earlier. Despite writing to two different tables, the maximum performance for schema 2 is even slightly better. With three or more clients and more than 50 lines written, the performance is about 500 lines per second better, but this has to be considered being a statistical fluctuation and hence insignificant. For one line written, the results are almost same. For five and ten lines written, schema 2 is slower for the most part. Where schema 1 had the best result with 128 clients and ten lines written, now schema 2 is also slower in that case.

It seems that, in general, schema 2 is worse than schema 1 with a small number of lines written, but better with a large number of consecutive lines written. One explanation for the better results for large numbers of rows written is that now the key structure is in a separate table from the rest of the measurement results. It always takes some effort from the database to update the key, and with schema 1 it was more resource-demanding as all the data is within one table. For schema 2, the key is in a separate table, and it is necessary to write it only once per measurement, whereas for schema 1 all the data, key included, is written for every measurement result. Therefore the overall amount of data written is larger for schema 1 than for schema 2.

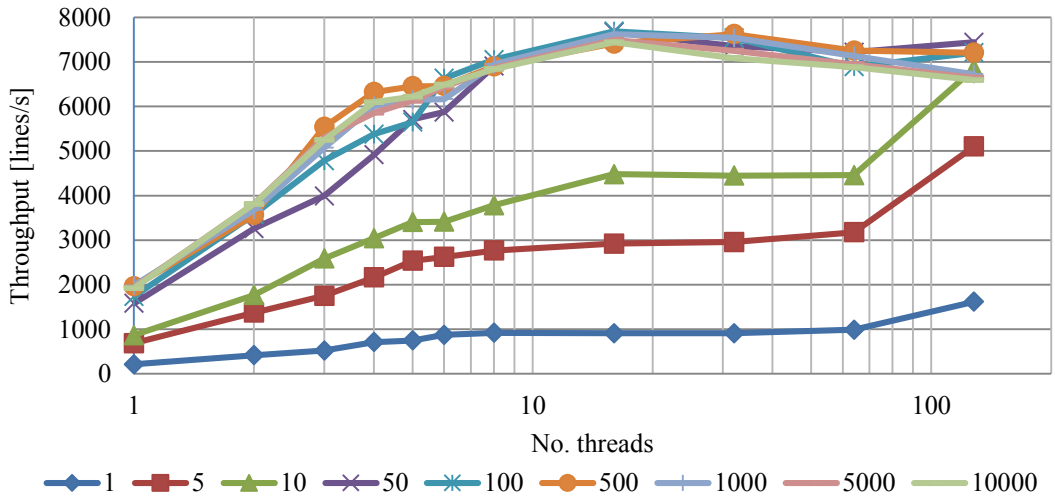

Figure 5.2. Write throughput against number of threads for database schema 2. Lines one to 10,000 represent different number of consecutive rows written.

Figure 5.3 shows the throughput against the number of threads results for database schema 3. For the third database schema, the results are significantly slower than for the first two. If only one line is written, the result is fairly close to the other two database schemas, but all the others are slightly or very much slower. The maximum performance is achieved with cases where 50 to 10,000 lines were written and is around 5,000 lines per second. Comparing the results in Figure 5.3 with the previous two figures, we can see that this is about one third less than the maximum performance of schema 2. Much less variation between cases also occurs. The drastically increased performance with 128 clients seen in schema 1 and schema 2 is now much smaller.

In schema 3, the second table from schema 2 where measurement results are stored is now split into two separate tables including more frequently and less frequently used

results. Writing the data to three tables seems to have a major effect on performance, now and the benefits of having the key structure in its own table has diminished.

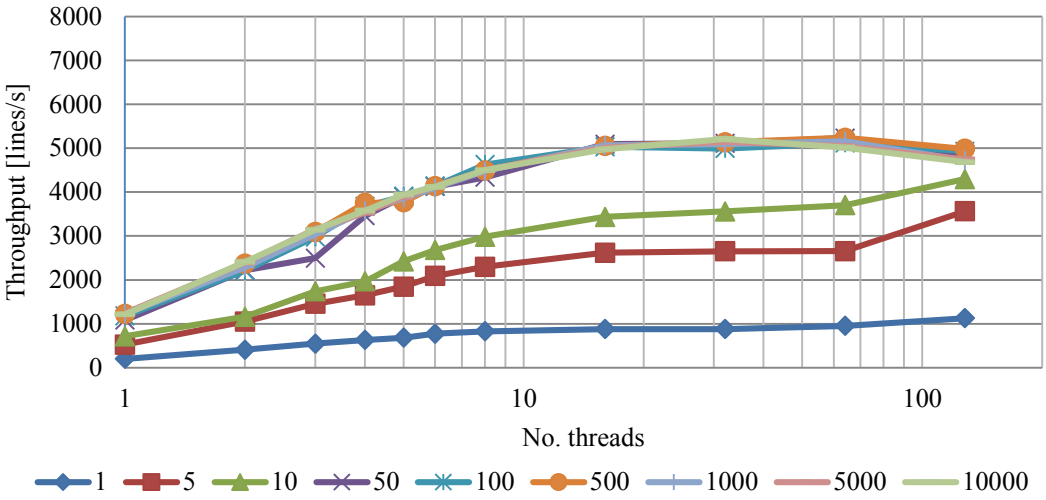

Figure 5.3. Write throughput against number of threads for database schema 3. Lines one to 10,000 represent different number of consecutive rows written.

5.1.2. CPU, I/O and Memory Performance

The server performance metrics for 10,000 rows written cases were recorded with Linux' sar tool [25]. The sar (System Activity Report) command is a system activity monitor used to store various metrics from the operating system. The recording interval for sar was set to minimum, which is one second. In order to obtain reliable measurement results, the running time for the test needed to be long enough. Test runs for one thread and database schema 2 were the fastest, and they ran from 0.004 seconds to around five seconds for one to 10,000 consecutive written rows respectively. For 5,000 consecutive written rows the time for one test run was less than three seconds and therefore it was decided to run server performance measurements only for 10,000 rows case. This way each test run lasted more than five seconds and a minimum of four samples was obtained after omitting the preceding and trailing samples from the peak load times. There are various features that can be monitored with sar, but in our case CPU load and I/O metrics are the most important things. MySQL server utilizes memory also in the background after the actual query execution is completed, and therefore it is not possible to accurately see how much memory is actually used for query execution.

System level (kernel) execution of the CPU utilization for database schema 1 and schema 3 are shown in Figure 5.4 together with the information when CPUs were idle waiting the system to perform an outstanding disk I/O request. System time includes time spent servicing hardware and software interrupts. Figure 5.5 presents the percentage of time that the CPUs were completely idle without any outstanding disk I/O requests and the time that CPUs were processing user lever application requests. Results for schema 2 are left out from the figures below because they are almost identical with schema 1 results.

MySQL and InnoDB were run with the default values which can be also seen with the hardware performance results below in Figure 5.4 and Figure 5.5. Even though the

server has two quad-core processors, the InnoDB parameter called *innodb_write_io_threads* is set to four by default. With this parameter, it is possible to restrict the number of background threads that service write operations. Looking at I/O waiting and CPU's idle percentage, it is clear that the progress until four clients is noticeable. After five or more clients, the change becomes slower. This does not necessary mean that all eight CPU cores were not used during the writing process. It is only that there are a maximum of four simultaneous I/O threads for writing operations. The other cores can be used by other processes.

The system usage time increases quickly to around 15–20% and stays more or less constant when introducing more than four clients. System time in general is considered a bad thing and can be a sign of, for example, too much context switching. Optimizing the software code might also help the situation. However, it is also possible that in this case the high system usage is caused more by the MySQL than the database writing module. To support this supposition, we should look at the database 3 curve in Figure 5.4. In database 3 there are many more writing operations than in databases 1 and 2. As seen in the previous section, the writing throughput is also similar with these system CPU usage results. Databases 1 and 2 are close to each other, but database 3 differs from the other two. It is possible that having more writing operations and greater need to keep up the table indices results in more system usage time.

Together with the CPU's I/O waiting proportion, it is good to examine the I/O transaction information in Figure 5.6. One method to measure the MySQL performance is to measure the Transactions per Second (TPS) [26]. Write requests per second (wtps) and the total amount of data written to hard drive in blocks per second (bwrtn/s) are represented for all databases. One block equals 512 bytes of data. Only write transactions are presented, since these tests were concentrated on the write performance of the database, and the number of read transactions was close to zero. There are similar characteristics between I/O wait in Figure 5.4 and write requests per second in Figure 5.6, as the results increase until about three or four clients and start decrease with more clients. It looks as if the maximum here is connected to the number of I/O write threads discussed above. It is also evident that there is a correlation between the number of transactions and length of I/O wait. However, it is interesting to see that number of block writes per second decreases only a little after the maximum values achieved with three and four clients. This is most likely a result of using the InnoDB insert buffer, which uses memory to cache the inserted data. Buffered data is then written to disk when the appropriate part of the database is handled. In the test situation, the database seems to have taken advantage of the insert buffer as the total amount of inserted data remains rather constant, but the number of write requests decreases drastically. With 128 clients, the number of write transactions is less than a fifth of the measured maximum value. It seems that bigger chunks of data per write transaction were performed with more clients.

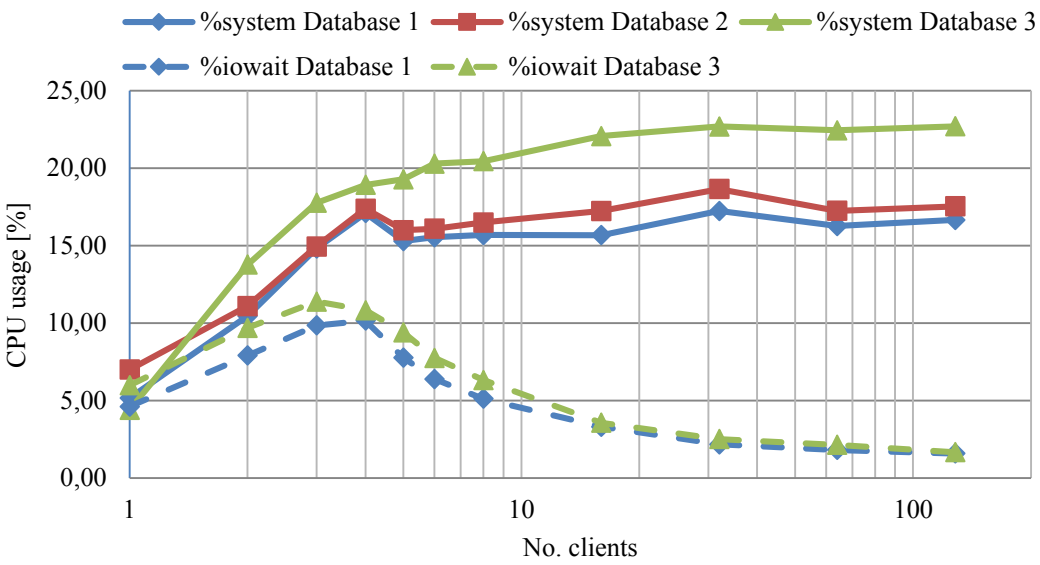

Figure 5.4. CPU usage at system level (%system) and I/O waiting percentage (%iowait).

As analysed earlier, the CPU's idle time decreases quickly until the four clients' case and then continues to decrease at a much slower rate. A similar process can be seen with application level usage in Figure 5.5 (*%user*), but now the change in application level growth speed is much less than the change in idle use decrease speed. The system level and user level processes are using most of the CPU time, and hence it seems that changing the MySQL settings to exploit more writing threads would be a good choice so as to increase the overall writing performance of the database. On the other hand, the I/O waiting time increases together with more clients up to a maximum of four writing threads. It is possible that taking more CPU cores into use would not help the situation, but would just make it worse. In fact, the number of write transactions starts to drop already after three clients. This is something that would require more research and testing on MySQL buffers and settings. Also, optimizing the code for the database writing module can have an effect on CPU usage and performance.

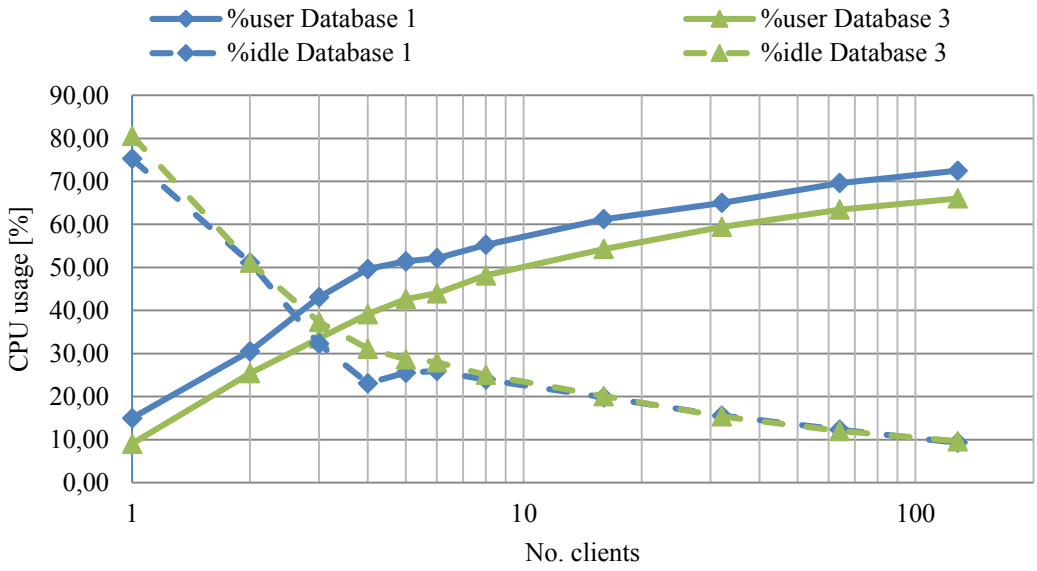

Figure 5.5. CPU usage at application level (%user) and percentage that is totally idle (%idle).

One piece of research [26] indicates that the optimal number of threads is more than four, as speculated above. However, the table size in the referred research is much smaller than in these tests, and there are no results for write-only testing. Nonetheless, more testing and experimenting with MySQL settings is recommended. After all, the maximum writing data rate achieved with database 2 and about 71,000 blocks written per second results in about a 36 MBps writing speed for the hard disk. There is still room for improvement, considering the average write throughput values for similar disks are well over 100 MBps [27].

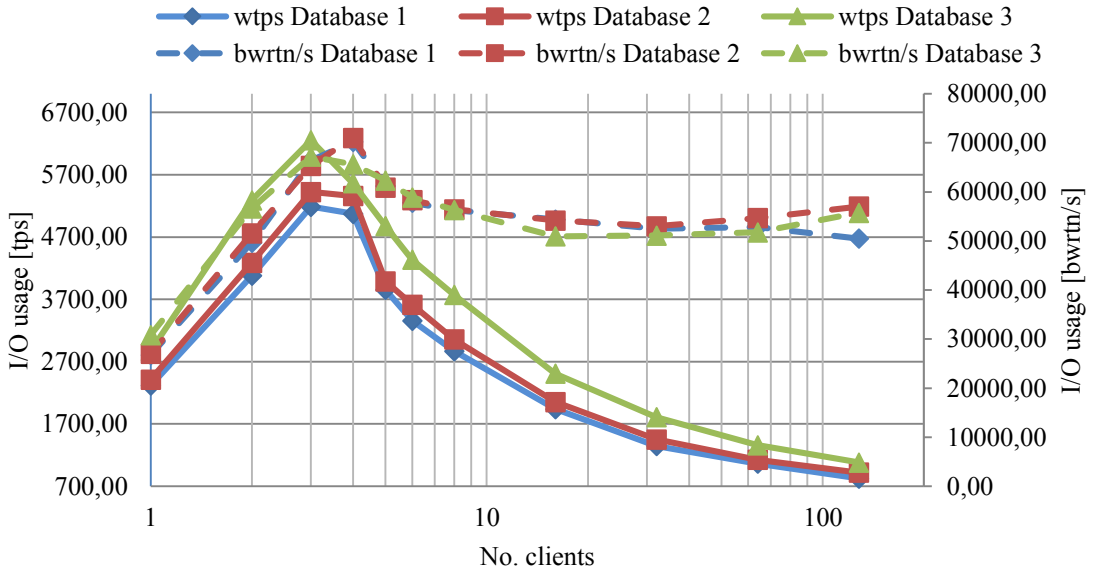

Figure 5.6. I/O information for all databases. Write requests per second (wtps) on the primary axis on the left and block writes per second (bwrtn/s) on the secondary axis on the right.

Figure 5.7 represents the total memory usage for all three database schemas. There are only small differences between the schemas. Until six or eight clients, the memory usage increases at the same rate for all the schemas. When introducing more clients, we start to see a difference between the schemas. Schema 1 has only one table to write to, whereas schemas 2 and 3 have two and three tables respectively. With more tables, there are more write transactions and the InnoDB write buffer is taken advantage of. Schema 1 has some advantage over the other two with 16 and 32 clients, but already at 64 clients, the system runs out of free memory in that schema too. The writing performance results in the previous section suggest that the maximum throughput was achieved already with eight clients, but considering the memory usage it would be interesting to see whether adding more memory resources would increase the write throughput. There is also research [26] to back up this supposition.

On the other hand, these results hint at the fact that the database writing module might be the bottleneck for writing performance. The maximum writing performance is achieved with eight clients, as shown in previous chapter. However, examining the CPU, I/O and memory metrics, we see that none of those are fully utilized with eight clients.

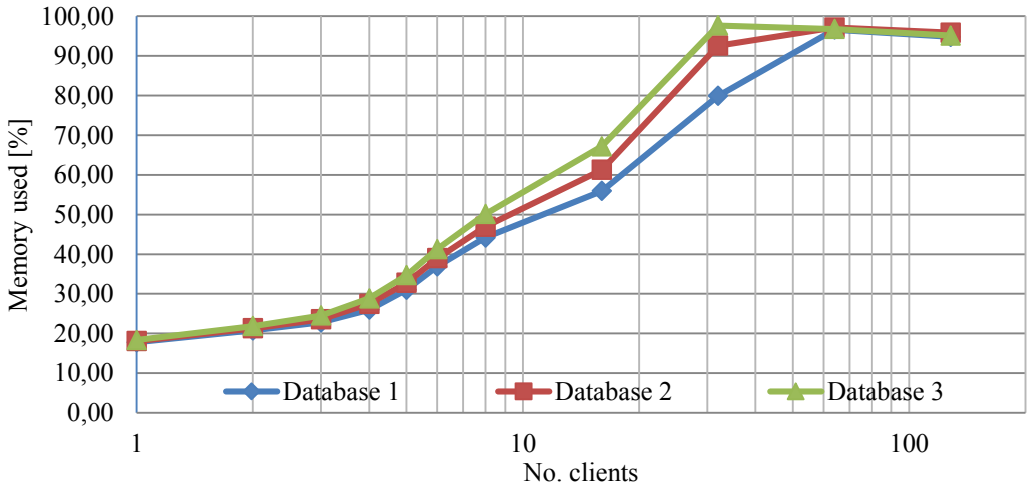

Figure 5.7. Total memory usage for all three database cases.

5.2. Reading Performance

Results for all five reading queries presented in Table 4.2 in section 4.2 are analysed in the following pages. In the analysis work, all results are considered and examined but only a selection of them is presented in graphs here, because many of the test cases produce similar results. The results are represented in graphs where total execution time of a query in seconds per client is displayed on the y-axis against the number of clients on the x-axis.

5.2.1. Query 1

Query 1 uses *customer_id* field to select certain numbers of results from all the data. This reflects the situation where one might want to search all the results for a certain customer. This field is part of the measurement id and the key structure for all database

schemas and therefore relatively fast to access. The measurements for Query 1 include four different cases returning 1,000, 10,000, 86,400 and 600,000 rows depending on the customer. In practice, this query denotes a case where all measurement data for a certain customer is fetched from the database. While analysing the query and the results, it was noticed that query optimization was needed to see the best results for all situations. Therefore, the second case in Query 1 includes an additional part presented in Figure 5.10. All test cases performed for Query 1 are shown in Table 5.1.

Table 5.1. Query 1 test cases

Name	MySQL query	Rows fetched
Case 1	customer_id=60	1,000
Case 2	customer_id=10	10,000
Case 3	customer_id=6	86,400
Case 4	customer_id=1	600,000

In Case 1 only 1,000 rows are fetched from the database and the test run is fast. The results between the database schemas are almost identical, as seen in Figure 5.8. This can be seen with all queries with about 1,000 rows, and is not characteristic only of Query 1. The general magnitude for performance times with small result sets are between 0.0005 and 0.002 seconds/client. Analysing all the results, it is seen that the maximum performance is achieved with three or in most cases four or more clients. The explanation for this phenomenon can be found from the server hardware specifications. The server has a quad-core processor and hence can serve query requests simultaneously from up to four clients.

Figure 5.8 Total execution time against number of clients for Case 1 in Query 1.

In case 2, database schema 3 deviated significantly from the other two as seen in Figure 5.9. The slowest one was schema 1, as it was required to go through each row during the search. Schemas 2 and 3 have separate tables for the id and were also assumed to be faster than schema 1. When finding the results for a certain customer, there are significant differences between schema 1 and schemas 2 and 3. For schema 1, it is necessary to go through all the rows, whereas for schemas 2 and 3, a certain

customer can be found from the *id* table which includes only one line of data for each measurement. Then, knowing the desired full ID, it is rather quick to search the measurement results from the other tables. However, it was noticed that the query was not working as anticipated for schema 2. The results were close to the ones for schema 1 but they should have been fairly closely related to schema 3. When analysing the query execution in MySQL, it was noticed that the query was actually going through the data in non-optimal order. It was actually going through each row in the *averageStats* table and comparing the data to the *id* table. This behaviour was corrected by adding a straight join option to the query. In this way, the execution was forced to go through the data in the desired order.

When fetching data from several tables, join expression was used in the queries. This expression is used to join rows based on a common field between them. In MySQL, there is a built-in query optimizer which tries to perform the given queries in the most optimal way possible. In some cases, the optimizer can fail and the tables are joined in a non-optimal order and the number of rows searched through can become much bigger than anticipated. In such cases, it is possible to use the straight join expression to force the optimizer to join the tables in the listed order.

Figure 5.10 shows Case 2 with straight join for schema 2 in more detail, and it is seen that now schema 2 is actually slightly faster than schema 3 performance.

Figure 5.9. Total execution time against number of clients for Case 2 in Query 1 with all Database model results.

Figure 5.10. Total execution time against number of clients for Case 2 in Query 1 where straight join syntax is used with database 2 and viewed with database schema 3 results.

Case 3 results are shown in Figure 5.11, where we see that database schema 2 and schema 3 are equally fast, but significantly better in performance than schema 1. The big difference in execution times can still be explained with the differences in query execution. Schema 1 goes through the entire database, whereas schema 2 and schema 3 manage with a few hundred inspected rows.

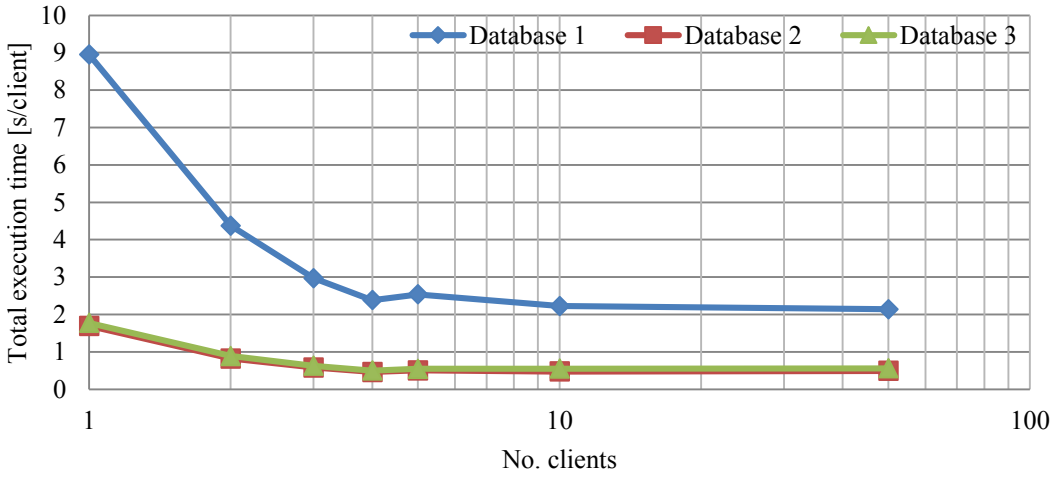

Figure 5.11. Total execution time against number of clients for Case 3 in Query 1.

In Case 4, shown in Figure 5.12, the amount of fetched data is already quite large, and thus, the data transfer time becomes the dominating factor instead of the query execution time, and differences between database schemas diminish. Schemas 2 and 3 are still somewhat faster than 1, but the difference is very small compared to earlier cases with smaller result sets, where the query execution time was dominating the total execution time. This change is shown in Figure 5.12 together with the drastic change when enough clients are brought to test the scenario. For this case, it was not possible to run the test with 50 clients, but 20 were used instead. Even with this number of

clients, the data transfer load becomes clear as it slows down the performance significantly. The time to perform the actual query is now only a minor factor, as the data transferred from the server to the client machine becomes the major factor.

Figure 5.12. Total execution time against the number of clients for Case 4 in Query 1.

5.2.2. Query 2

For Query 2 *ul_delay* field, one of the primary measurement results was chosen for result search criteria. Where Query 1 used a field included in the key structure, it is not the case now. It is assumed that the performance will be worse for this query which includes four basic test cases returning 556, 2,780, 27,800 and 278,000 rows. All the cases are represented in Table 5.2 below. In practice the Case 1 would fetch all the rows from the database where uplink delay is less than one millisecond.

Table 5.2. Query 2 test cases

Name	MySQL query	Rows fetched
Case 1	ul_delay < 1	556
Case 2	ul_delay < 5	2,780
Case 3	ul_delay < 50	27,800
Case 4	ul_delay < 500	278,000

Case 1 results are shown in Figure 5.13 and do not differ significantly from the Query 1 results. Total execution times are so small, even though the time difference is large, even 50%, they can be considered to be in the same magnitude range.

Figure 5.13. Total execution time against number of clients for Case 1 in Query 2.

Case 2 and 3 end up with an almost identical performance. The results for Case 3 are shown in Figure 5.14 and the database in schema 2 clearly is the slowest of the three database schemas for this query. For these cases, the data transfer from server to client is not a major factor, because the number of rows fetched is small. Hence the query execution is a more dominant factor for performance. For schema 1, query is searching the values from only one table, which seems to be rather efficient in this case. Looking at the results, it seems that it is nearly as efficient is to search from relatively small table and to combine data from two other tables as is done in schema 3. However, the search from a large table and then combining the data from another table as done with schema 2 seems to be less efficient. This phenomenon could be explained with the assumption that searching from a bigger table (schema 2) is a slower process than executing a search from a smaller table (schema 3). This might be a consequence of the DBMS's memory settings. If the searched data does not fit into a memory buffer, the query execution slows down, which is probably the case here with schema 2.

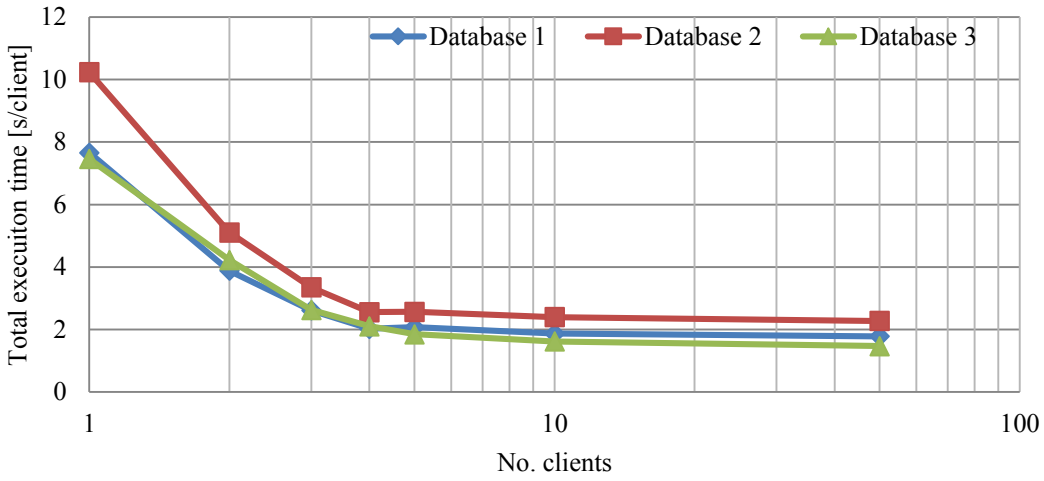

Figure 5.14. Total execution time against number of clients for Case 3 in Query 2.

Case 4 for Query 2 is similar to what we saw in Query 1, but has some significant differences as seen when examining Figure 5.15. First, there are fewer rows fetched, and second, the normal 50 clients was the maximum measured. Database schema 1 was the slowest for Query 1 Case 4, but now it is the fastest one, whereas schema 2 now has the worst performance of all three. The data transfer time becomes more significant with a large number of clients, but it seems that there is still room for a query execution speed to make a difference, as schema 1 is still faster than the other two database schemas. It seems that now schema 1 has benefitted from the fact that it does not need to combine data from several tables.

Figure 5.15. Total execution time against number of clients for Case 4 in Query 2.

5.2.3. Query 3

Query 3 consists of four different situations. The measurements were performed with MySQL query cache in use and without it, and within both of these situations there were two individual situations with different amounts of data fetched from the database. For database 1, there is no difference between these two approaches, but databases 2 and 3 have several tables in them and first results were fetched only from one table and then from all the tables. This has a minor effect on query speed, but mostly on the data transfer times. All 16 test cases for Query 3 are shown and explained in Table 5.3. All four different situations in Query 3 were performed with four setups returning result sets with 556, 2,780, 27,800 and 278,000 rows. This query reflects a situation where one wishes to fetch all results where there is a certain number of duplicates in the uplink connection. In some cases, it might be enough just to see how many results fit the limit without fetching all the information from all the tables and, therefore, one table cases are tested. In many cases, it is useful to see the complete result set and fetch data from all the tables.

Table 5.3. Query 3 test cases

Name	Query cache	Tables	MySQL query	Rows fetched
Case 1	YES	1	ul_dupl < 1	556
Case 2	YES	1	ul_dupl < 5	2,780
Case 3	YES	1	ul_dupl < 50	27,800
Case 4	YES	1	ul_dupl < 500	278,000
Case 5	YES	ALL	ul_dupl < 1	556
Case 6	YES	ALL	ul_dupl < 5	2,780
Case 7	YES	ALL	ul_dupl < 50	27,800
Case 8	YES	ALL	ul_dupl < 500	278,000
Case 9	NO	1	ul_dupl < 1	556
Case 10	NO	1	ul_dupl < 5	2,780
Case 11	NO	1	ul_dupl < 50	27,800
Case 12	NO	1	ul_dupl < 500	278,000
Case 13	NO	ALL	ul_dupl < 1	556
Case 14	NO	ALL	ul_dupl < 5	2,780
Case 15	NO	ALL	ul_dupl < 50	27,800
Case 16	NO	ALL	ul_dupl < 500	278,000

There are no major differences between Case 2 and 3 returning 2,780 and 27,800 rows in this setup. Figure 5.16 clearly brings out the differences between the three databases in these two cases. As the data is fetched only from one table with schema 2 and schema 3, it can be assumed that the different amounts of data transferred are now shown in the results. Database schema 3 is the fastest as schema 2 and schema 1 follows accordingly. The difference between each database is slightly smaller as more reading clients are introduced to the setup.

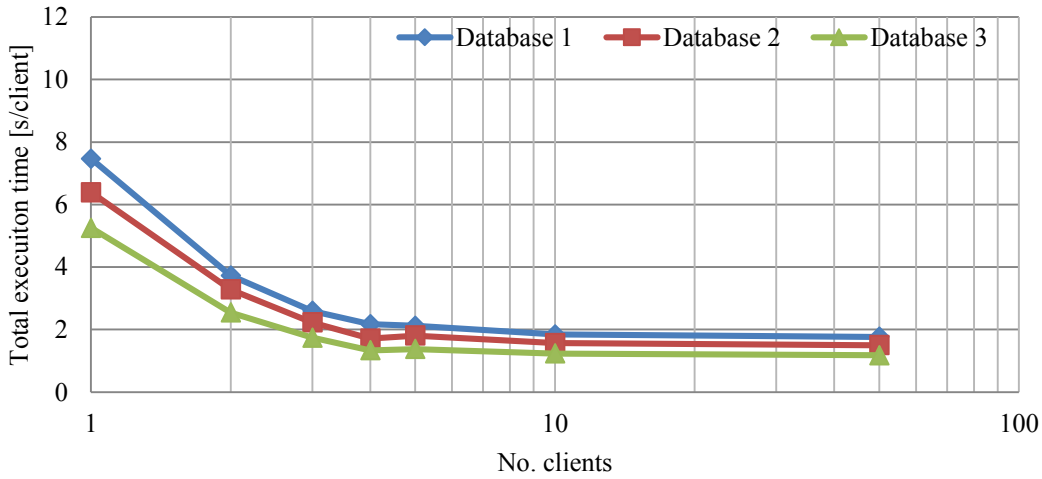

Figure 5.16. Total execution time against number of clients for Case 3 in Query 3.

In Case 4, 278,000 rows are fetched from the database and the resulting time is now noticeably slower for all databases. This is the case especially with a small number of clients. Figure 5.17 points out an interesting situation with database 1 and 50 simultaneous clients running the query. The speed collapses with too many clients

loading the server. Database schema 1 is returning all the data, but in schema 2 and schema 3 only appropriate data from one table in this case. This is why the dip in performance is not yet seen with all database cases. This can be seen later on Figure 5.20 and is analysed more below.

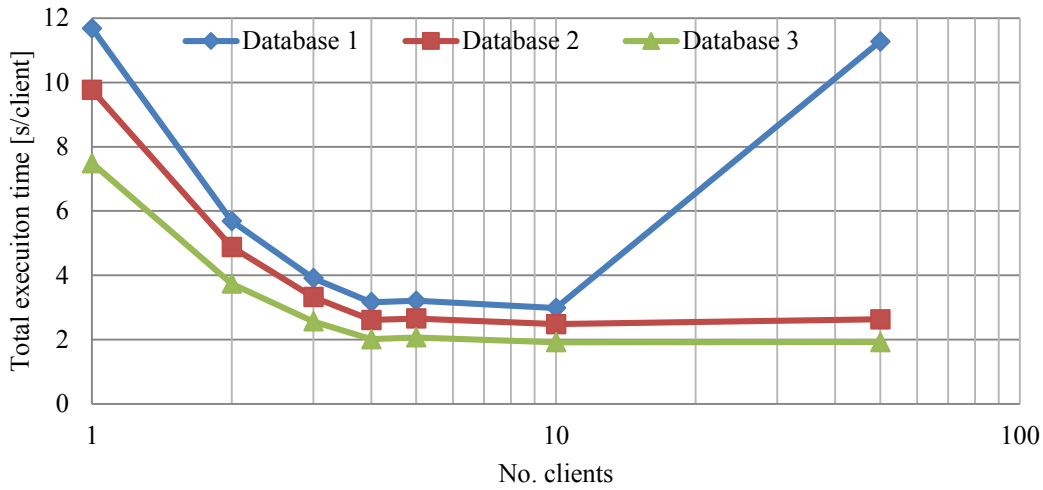

Figure 5.17. Total execution time against number of clients for Case 4 in Query 3.

Figure 5.18 represents the Case 5 in Query 3. In this situation, the MySQL query cache is still in use, but now the data is fetched from all tables also with schema 2 and schema 3. Fetching 556 rows of data and using the query cache is very fast, and the results vary very little, as seen in Figure 5.18. The results for Case 1 are almost identical. Results like this are not very reliable, but certainly show the general magnitude of performance times with small result sets. The results with 556 returned rows are very similar to the previous cases and queries with same number of returned rows. However some significant differences can be found in the performance with 2,780 and 27,800 rows.

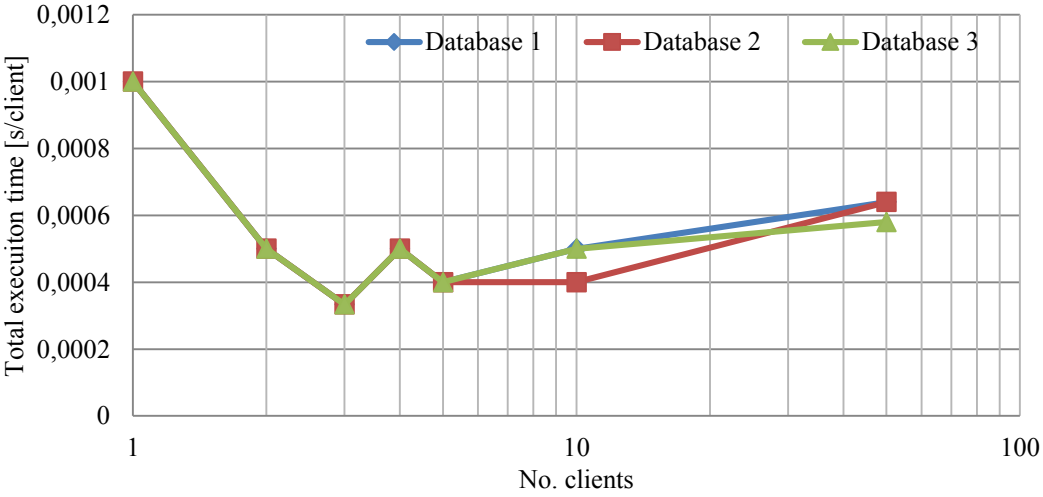

Figure 5.18. Total execution time against number of clients for Case 5 in Query 3.

Case 7 is shown in Figure 5.19, and we can see that the magnitude of the performance remains the same as with Case 3. However, the order of the database schemas is now different. Now schema 3 is the slowest followed by schema 1 and schema 2 as seen in Figure 5.19. The change in the order is seen most clearly with a small number of clients and the difference from earlier cases with more clients is not a big one. The change is mostly explained with more data transferred now that the data from all tables is fetched. It seems, though, that the effect of more data is still not a big one, as schema 2 still remains faster than schema 1. The faster query speed is still a major factor. The reason why query for schema 2 is the fastest in Case 7 remains uncertain. There are several things that have an effect. The size of the queried table has an effect together with the memory buffer as discussed in previous section. InnoDB page structure offers explanations as examined at the end of this section. Joining data from several tables and the use of the query cache also play a role in query performance.

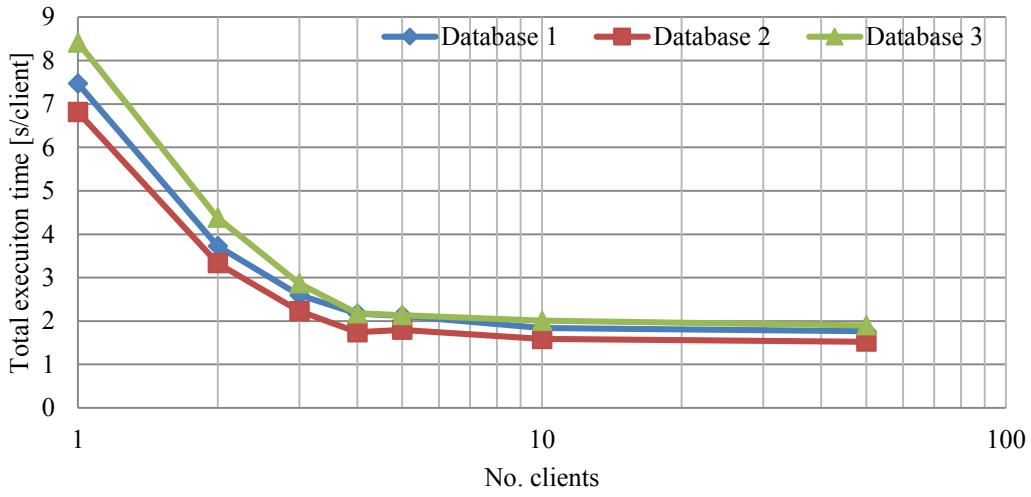

Figure 5.19. Total execution time against number of clients for Case 7 in Query 3.

Now that the transferred data is significantly bigger when 278,000 rows are fetched from the database, the decreased speed especially with large number of clients is clearly seen in Figure 5.20. The figure points out now that the amount of data required to transfer from the database server to a client machine can be a factor if there are a large number of readers. Comparing Figure 5.20 with Figure 5.19, we can also see that now the data transfer has become the dominating factor as schema 2 and schema 3 seem to have almost equal performance with 278,000 rows whereas the difference between the two was clear with 27,800 rows. The reason why schema 1 is faster than the two other database schemas can be found from the fact that it does not have to join the data from several tables as in schemas 2 and 3, which have the data divided into two and three separate tables. These results also give support to the assumption made above that with 27,800 rows the amount of data was less significant than the query speed.

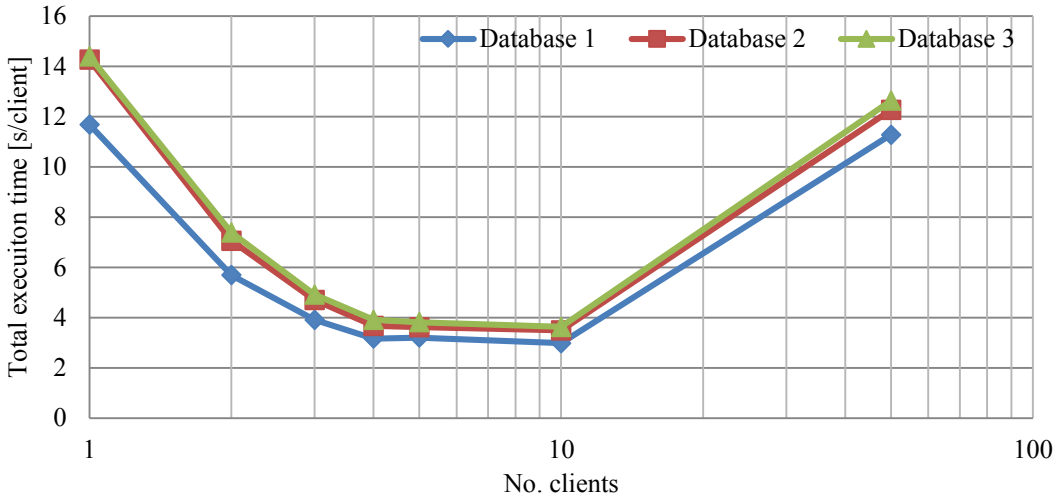

Figure 5.20. Total execution time against number of clients for Case 8 in Query 3.

Figure 5.21 below shows the difference between enabling and disabling the MySQL query cache, as it is now disabled in Query 3 from Case 9 forward. Now the result for fetching 556 rows is more like the results got with the query cache and 2,780 or 27,800 rows rather than the ones seen in Figure 5.18. The query cache has a size limit option, and this is why faster performance in these tests can only be seen on 556 row cases. By changing the size of the query cache, it is possible to optimize the database performance for reading purposes. The results for Cases 10, 11 and 12 do not differ too much from the ones discussed earlier when the query cache was used and data was fetched from only one table.

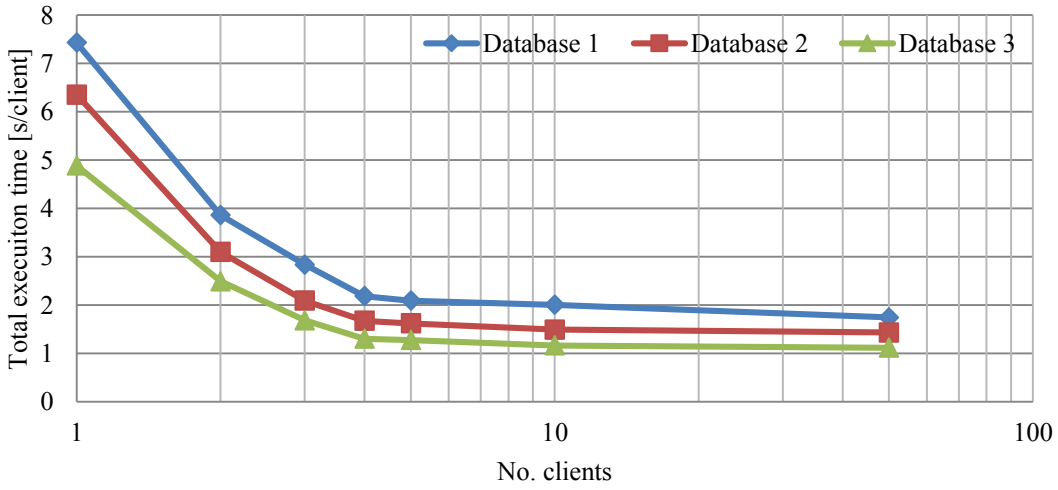

Figure 5.21. Total execution time against number of clients for Case 9 in Query 3.

The last situation for Query 3 is MySQL query cache disabled and data fetched from all tables. Results for Case 16 are similar to those shown in Figure 5.20. Cases 13, 14 and 15 on other hand result in a different order between the database schemas. As seen in Figure 5.22, now schema 1 is the fastest followed then by schema 3 and schema 2. Schema 2 is now the slowest, compared to the situation shown in Figure 5.19 where it was the fastest. Now the amount of transferred data is the same for all three database

schemas, and hence the reason for this change is most likely to be found in the query speed. Without the query cache, schema 2 performance drops more than the others.

To understand more about the query speed, we need to analyse theInnoDB page structure of the MySQL storage engine. InnoDB stores the records into a structure called a page. The page has a fixed size, 16 KB as a default, and it contains the actual data (i.e. rows) and some header information. The more columns a table has, the fewer rows fit into a single table, as there is more data in a single row. As a table has a fixed size, this also leads to more disk access to read a certain number of rows. In our case, schema 3 has smaller tables with fewer columns than schema 2, which can lead to faster queries.

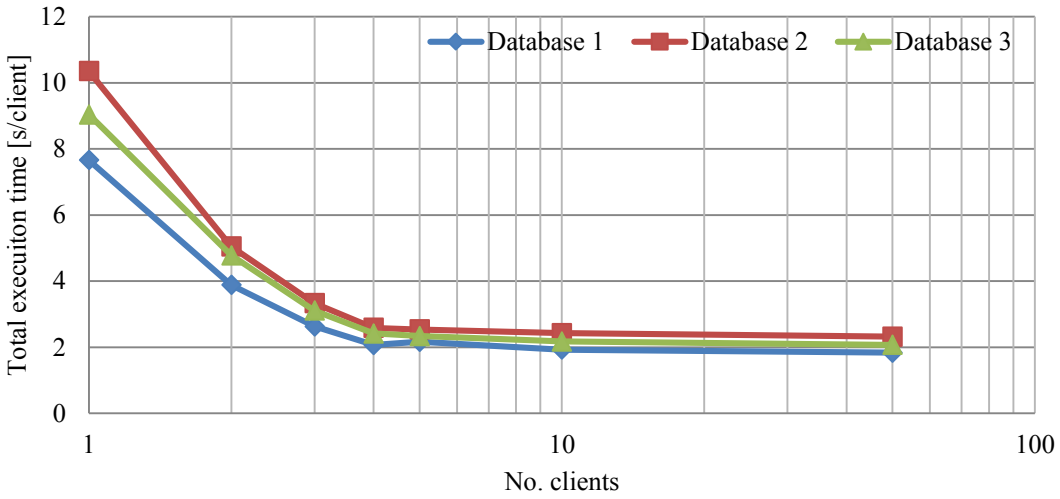

Figure 5.22. Total execution time against number of clients for Case 15 in Query 3.

5.2.4. Query 4

Query 4 combines information from two different fields. These fields, *customer_id* and *ul_delay*, are in different tables in schema 3. *customer_id* is in *id* table and also part of the key structure, where *ul_delay* is in the *main* table. These test cases should show whether there is any difference in performance when searching information from several fields and tables comparing to previous queries where only one field was used as search criteria. In practice, one might wish to search, for example, for all results with an uplink delay smaller than 500 and belonging to a customer with ID number 60. There are five different test cases all presented in Table 5.4 below. Cases 3 and 4 are similar, and the queries fetch the same number of rows, but use different field criteria for search query.

Table 5.4. Query 4 test cases

Name	MySQL query	Rows fetched
Case 1	customer_id=60 AND ul_delay < 500	500
Case 2	customer_id=10 AND ul_delay < 5,000	5,000
Case 3	customer_id=6 AND ul_delay < 50,000	50,000
Case 4	customer_id > 56 AND ul_delay < 100	50,000
Case 5	customer_id < 6 AND ul_delay < 50,000	250,000

Case 1 results are similar to the ones for Query 3 Case 5 presented in Figure 5.18. Case 2 is similar to Query 1 Case 2, where Databases 2 and 3 have a total execution time of approximately tenths of a second and schema 1 of several seconds. Performance for schema 1 is practically the same for both queries, and schema 2 and schema 3 are presented below in Figure 5.23. Comparing these similar cases we see that, even though the number of rows fetched are halved from 10,000 to 5,000, the total execution time is not a half. Comparing Figure 5.10 and Figure 5.23, we see a roughly 40% decrease in execution time. From this, we can conclude that making a query from two fields in different tables is in fact a slower process than a query from only one table.

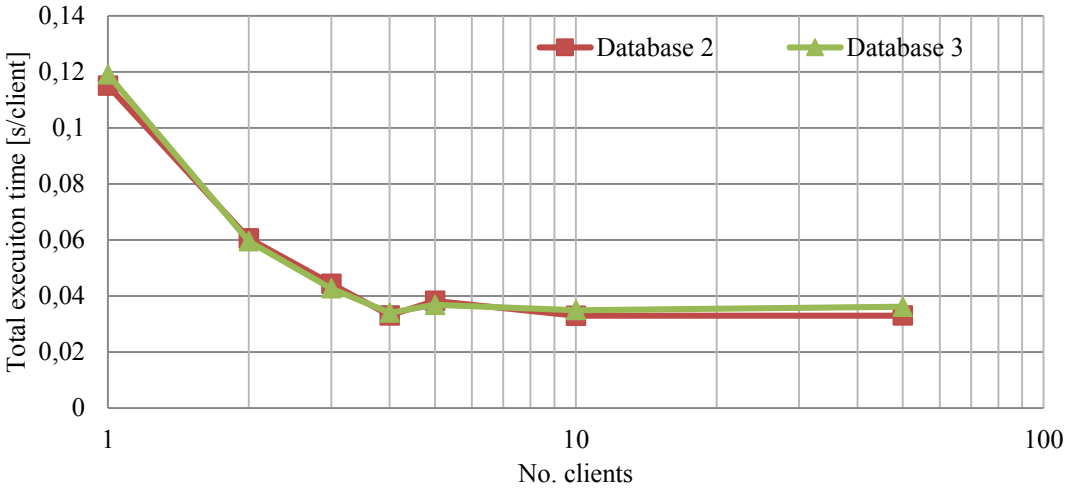

Figure 5.23 Total execution time against number of clients for Case 2 in Query 4 without database schema 1.

Cases 3 and 4 are presented in Figure 5.24 and Figure 5.25, respectively. Even though they both fetch the same number of rows, the results differ greatly. The queries are very different from each other, and it seems that this has an effect on schema 2 and schema 3, but has no effect on schema 1. Schema 1 has all the information in one table, and it seems that for this query the difference between having everything in one table or several tables, as for schema 2 and 3, is the dominating factor. For schema 3, the execution time in Case 4 is little more than double compared to Case 3. For schema 2, the same applies for five or more clients, but with fewer clients the increase in execution time is more. Case 3 has similar results to Query 1 Case 3 which is also a very similar query, but Case 4 results are quite unique within these tests. In schema 3, results are searched from *id* and *main* tables which are both rather small, whereas in schema 2 there are only two tables and the query is searching from both. For both, schema 2 and 3, the query first limits the results for certain clients and then searches for a more restricted set of results limiting them with an uplink delay. Schema 3 benefits this more than schema 2 as the query is faster for a smaller table. It seems that, if a query has to look up information from several tables, the table size has some effect on the execution performance.

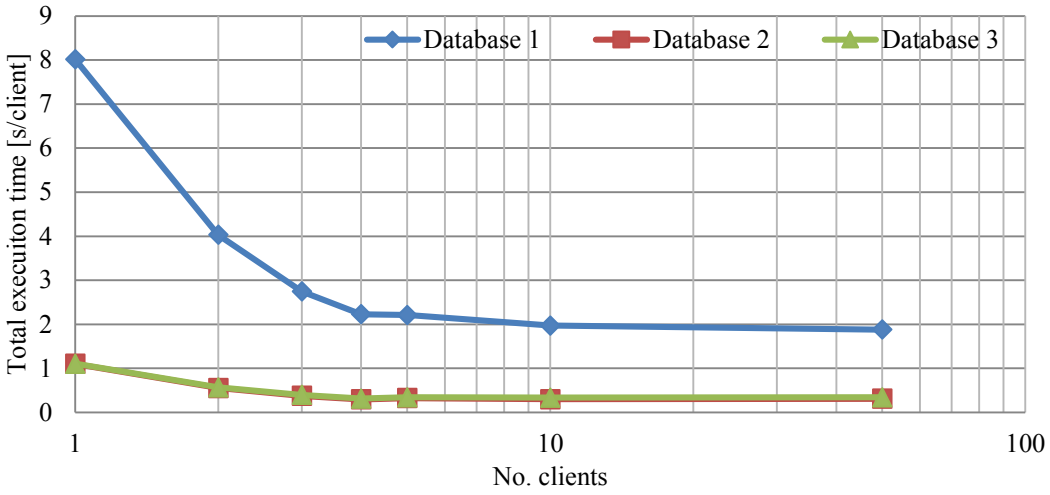

Figure 5.24. Total execution time against number of clients for Case 3 in Query 4.

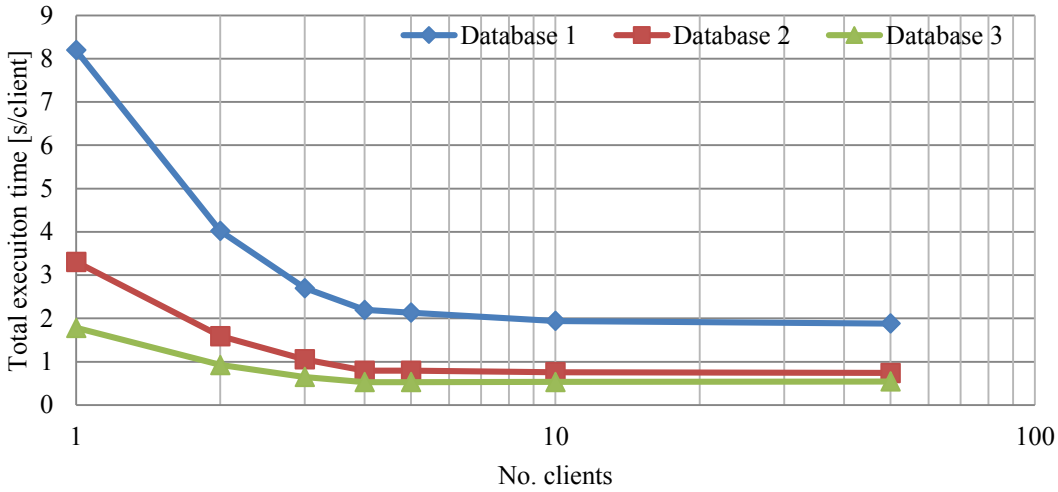

Figure 5.25. Total execution time against number of clients for Case 4 in Query 4.

Case 5 for Query 4, shown in Figure 5.26, has some similarities with previous cases with a large number of rows fetched. Execution time per one client with a large number of clients is around the same as with one client. This is common for all queries as long as the resulting data set size is big enough for the transfer time to become the major factor in execution time. Results are in line with other cases for Query 4 as well as for Query 1 where *customer_id* field is also the search criteria. Database schema 1 is the slowest, but schema 2 and schema 3 do not differ very much. Now that the data transfer time is the major factor, differences seen in Case 4 between schema 2 and schema 3 disappear.

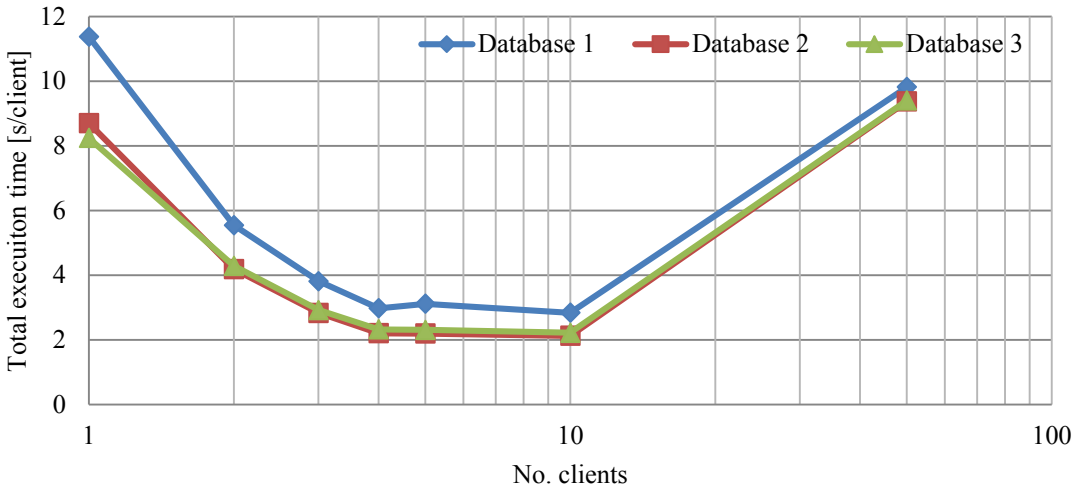

Figure 5.26 Total execution time against number of clients for Case 5 in Query 4.

5.2.5. Query 5

Where Query 4 combined the *id* and *main* tables, now Query 5 is combining *main* and *secondary* tables. This situation now brings all the database schemas closer together, as the *id* table for database schema 3 is small and includes the index structure. When running queries for *ul_delay* and *ul_dupl*, the search in schema 3 has to go through both large tables. Cases 3, 4 and 5 all return the same number of rows, but now the performance effect of different restriction criteria is examined. This denotes a practical case where the user wants to see all the measurement results restricted by uplink delay and duplicate values. All seven test cases are shown in Table 5.5 below.

Table 5.5. Query 5 test cases

Name	MySQL query	Rows fetched
Case 1	ul_delay < 1 AND ul_dupl < 1	556
Case 2	ul_delay < 10 AND ul_dupl < 10	5,560
Case 3	ul_delay < 50 AND ul_dupl < 50	27,800
Case 4	ul_delay < 100 AND ul_dupl > 50	27,800
Case 5	ul_delay < 500 AND ul_dupl > 449	27,800
Case 6	ul_delay < 100 AND ul_dupl < 100	55,600
Case 7	ul_delay < 500 AND ul_dupl < 500	278,000

All cases for Query 5 relate closely to some cases in Query 2. Case 1 has results that are similar to Query 2, Case 1 shown in Figure 5.13. Case 2 is almost identical with Cases 3, 4, 5 and 6, but just slightly faster. Case 5 is shown in Figure 5.27 and differs from cases 3 and 4 only in schema 3 and small number of clients. In Case 3 and 4 schema 3 closely follow the performance of schema 1, but as we see in the figure below, there can be some query cases found with a slightly different performance. Case 6 is in correlation with the previous three cases as well as with Query 2 Case 3 shown in Figure 5.14. Case 7 correlates to Query 2 Case 4 shown in Figure 5.15 and supports the behaviour seen in all queries with large quantity of results.

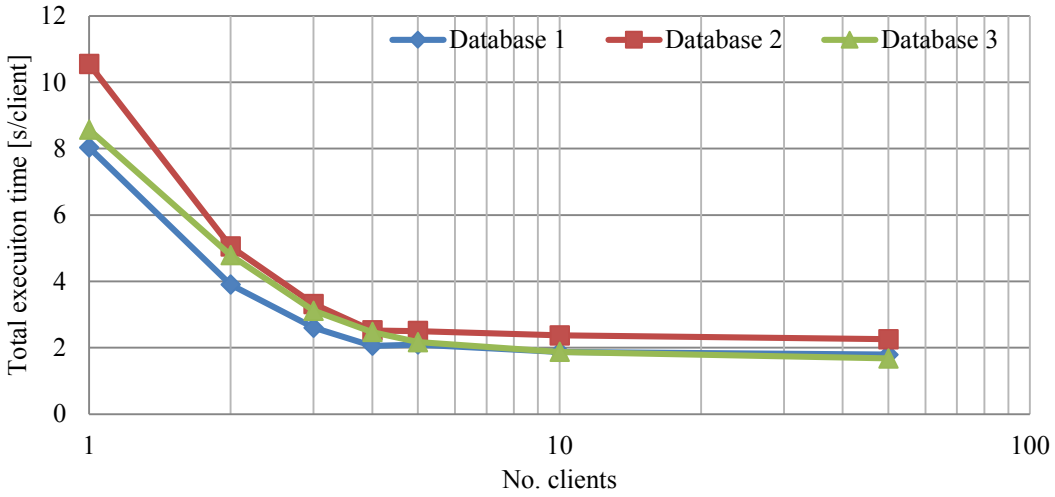

Figure 5.27. Total execution time against number of clients for Case 5 in Query 5.

5.3. Choosing the Best Database Model

The best database model of the three compared was chosen by giving points to each of them, based on their writing and reading performance tests. For writing, throughput (lines/s), and for reading, total execution time (s/client), are used for point calculations. For writing performance, the method of determining the points was rather simple. The following was performed for each database schema. First, throughput average [23 p.1053]

$$\bar{s}_i = \frac{\sum_{i=1}^{n} s_i}{n}, \tag{4}$$

where s_i is a single throughput value and n is the total number of values, was calculated over all the results from a single database schema. Points for each database schema were determined using the principle that the best schema is given ten points and the other two less than ten with correct proportion based on their throughput average. The point value

$$x_{dbi} = \frac{\bar{s}_i}{\max(\bar{s}_1;\bar{s}_2;\ldots;\bar{s}_n)}\,10, \tag{5}$$

where *max* is the maximum value of all throughput averages, and s_i, is calculated for each database schema.

A similar method was used to determine the points for reading performance. In reading tests, there are five queries for each of the three database schemas. Each query contains a different number of test cases. First we determine the execution time average [23 p.1053]

$$\bar{t}_{ci} = \frac{\sum_{i=1}^{n} t_i}{n} \tag{6}$$

for each test case. t_i is a single execution time and n is the total number of values. Because reading performance is measured by execution time, the best value is the smallest one. This is the opposite from throughput in writing performance. To determine the points for each case, we use

$$x_{ci}=\max[M_1;M_2;\ldots;M_n]+\min[M_1;M_2;\ldots;M_n]-M_i, \quad (7)$$

where *max* is the maximum value of all M_i and *min* the minimum value of all M_i.

$$M_i=\frac{\bar{t}_{ci}}{\max(\bar{t}_{c1};\bar{t}_{c2};\ldots;\bar{t}_{cn})}10 \quad (8)$$

defines the M_i used in the previous equation. *max* is the maximum value of all $\bar{t}_{ci}$. For each five queries, we calculate the point average

$$\bar{x}_{qi}=\frac{\sum_{i=1}^{n} x_{ci}}{n}, \quad (9)$$

where n is the total number of cases in the query. Table 5.6 shows all points for each database schema and query. We can see that schema 1 has a large variation between the different queries, whereas schemas 2 and 3 have much more regular results.

Table 5.6. Average points for each database schema and query in reading performance. Maximum point for an individual test case is 10

	Db schema 1	**Db schema 2**	**Db schema 3**
Query 1	5.10	9.99	9.53
Query 2	9.81	8.29	9.59
Query 3	9.28	9.65	8.49
Query 4	4.39	9.65	9.93
Query 5	9.75	8.00	9.64

Finally, one point value for each database schema in reading performance is calculated with

$$x_{dbi}=\frac{\sum_{i=1}^{n} \bar{x}_{qi}}{n}, \quad (10)$$

where n is the total number of queries. Points for both writing and reading tests are shown in Table 5.7. The writing performance was weighted as a factor of two and hence those points were doubled and added to the points for reading performance. Schema 2 received the most points, being the best in writing performance and coming a close second in reading performance. Schema 1 was a close second in writing performance, but did not do very well in overall reading performance. It's also noticeable that schema 3 was the best in reading performance, but trailed significantly in writing. When comparing the designed schemas with the above-mentioned criteria and considering the importance of writing performance, it was clear that the schema 2

was chosen as the best one and used in future development. With a different kind of emphasis, the results might be different, though schema 2 is very constant in both reading and writing performance. Schemas 1 and 3 clearly have weaknesses in either writing or reading performance.

Table 5.7. Average points for each database schema and for both writing and reading performance. Maximum point for an individual test case in reading performance was 10. Maximum point for writing performance is also 10. For total points from reading performance and doubled points from writing performance are counted

	Db schema 1	**Db schema 2**	**Db schema 3**
Points for writing	9.70	10.00	6.81
Weighted points for writing	19.41	20.00	13.61
Points for reading	7.66	9.12	9.43
Total points	**27.07**	**29.12**	**23.04**

5.4. Performance with the Writing Module

After the choice of the best database model from the three tested, tests with the developed database writing module were performed. These tests were performed only with the chosen schema 2. Test setups included four and five laptops, depending on the cases. In order to create as realistic a traffic as possible to be stored on a database, some artificial data traffic was created to test the network. The D-ITG (Distributed Internet Traffic Generator) tool was used so as to create five UDP packets per second, with 512 Bytes each. The traffic rate was not the optimal to give a value for each stored row in the database, but more data would have congested the connection and hence reduced the overall performance. The averaging interval for Qosmet was set to 50ms, resulting in 20 measurement results for each second and each client. The number of clients running the two point measurements were 10, 20 50, 100, 120, 150, 200, 250, 300 and 350.

The measurement was carried out as illustrated in Figure 5.28. The network analytics was performed between the primary measurement nodes and the secondary measurement nodes. Primary measurement nodes sent the data to the database server. All the computers were connected with a wired link of 10Gbps and were equipped with 1Gbps network interfaces. The database server was also equipped with a 1Gbps network interface. First, the tests were made with one writing module running on a database server and two pairs of measurement computers (primary measurement nodes) sending the measurement results to the module. Both pairs had an equal number of clients running on them and sending the results. The same principles apply to the two writing module setup, where the measurement setup was identical, but the results were sent to two independently running writing modules. They were run on the same database server and writing the results to one database. During the experiment it was noticed that the computers running the secondary services were limiting the number of clients. No more than 150 clients per secondary service could be run without a significant loss of measurement data. In order to ascertain the performance with three writing modules, a third computer running a secondary service (secondary measurement node 2) was introduced to the setup.

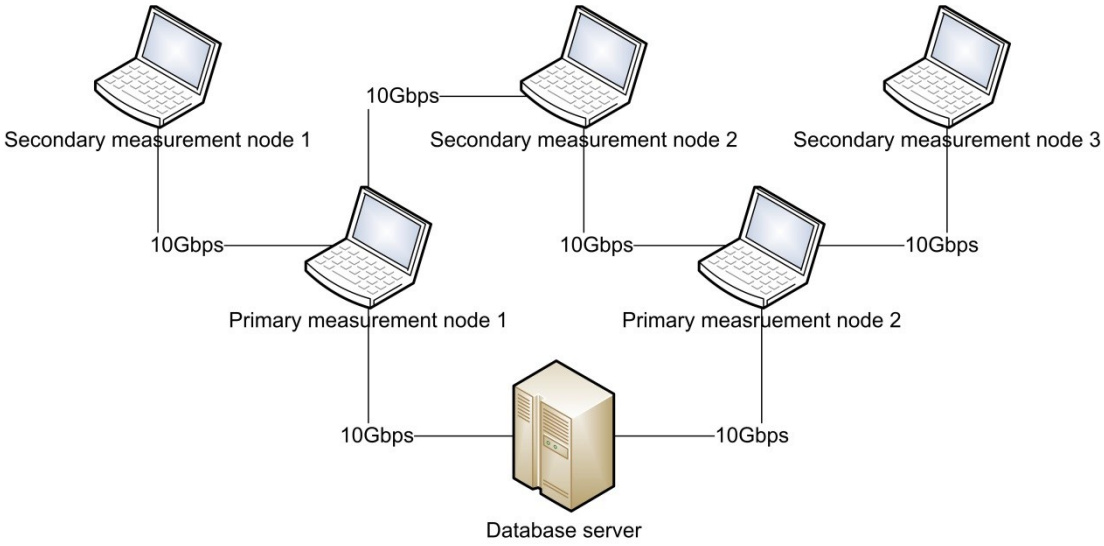

Figure 5.28. Test setup for performance tests with writing module. All the computers, including the database server, were equipped with 1Gbps network interfaces.

At the beginning of the measurements, it was noticed that the writing module had a bug in the thread loop. The module had a sleep interval of 100ms after each new measurement information was written to the database. A quick fix to avoid the problem was to fix the code to wait only if there are no measurements in the buffer. Not long into the measurements it also became clear that the writing module was one of the bottle necks in the system and was buffering the incoming results with many clients sending the data. The timestamp found from the measurement data represents the actual time when the measurement took place and hence is insufficient to express the time taken for all the measurement data to be stored in the database. To overcome this shortcoming, a new timestamp value was added to *measurementId* table in database. Column *edited_time* represent the time when any of the values from the appropriate row is updated. With the help of this last edited timestamp and the timestamp for end of measurement, it was possible to determine how long it actually took to write all the data to the database, and whether the buffer for the writing module was in use.

Figure 5.29 illustrates the principle of how to determine the time when all *n* measurements are sending the results to database. This is important in order to ascertain exactly how many lines are written in the time the entire measurement is running. All individual measurements or clients are running simultaneously in a time slot when the last client starts and the first client stops. In principle, the lines are stored to the database in the order in which they arrive. According to this principle, the total number of lines written is *measurement 1 – measurement n*. For example 280 – 80 resulting to 200 lines. Accordingly, the time taken to write those lines is *n start time – 1 end time*.

The writing module could process only a certain number of results in a given time and hence in cases where there were many clients, some results were buffered for future processing. The length of this buffer was determined from the last inserted row in the database and from the time all measurements had ended. Basically, the buffer tells us how long it took to write all the results to the database after the measurement was over.

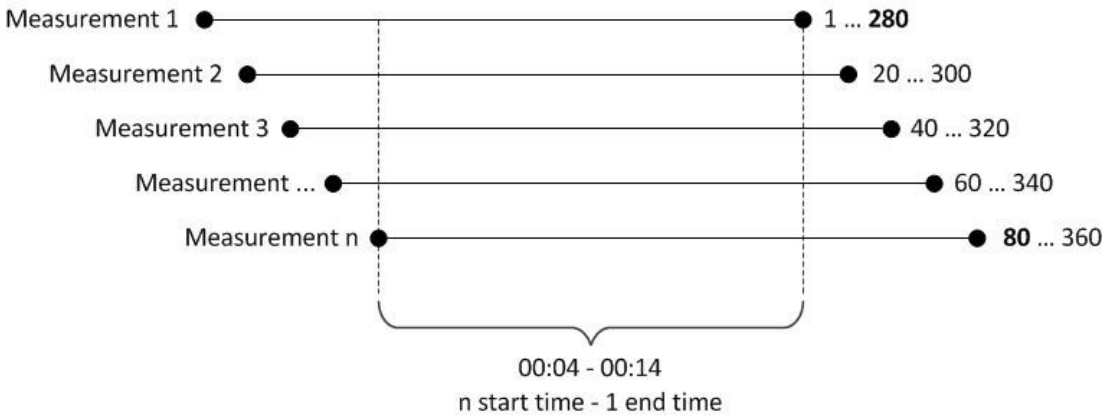

Figure 5.29. Starting and ending times and points for n measurements.

Figure 5.30 below show the results for all measurements performed for one, two and three writing modules. The length of the buffer in seconds is also illustrated in the same figure. With one and two writing modules, it became clear that the module implemented has a threshold of around 2,000 lines/second. This indicates that, in some cases, it is beneficial and efficient to run several writing modules even on a single database server. With this approach, it is possible to take all the power into use. Introducing a third writing module, we were able to reach the maximum capacity of the database server. Comparing the results to Figure 5.2, it is seen that the maximum write performance for the database with three threads or clients is around 5,000 lines/s. Results from these writing module tests give the maximum performance which is very close to the numbers in the database-only writing tests.

The buffer length shows that the writing module cannot process the results as fast as desired, and as more and more results arrive the time to process them increases.

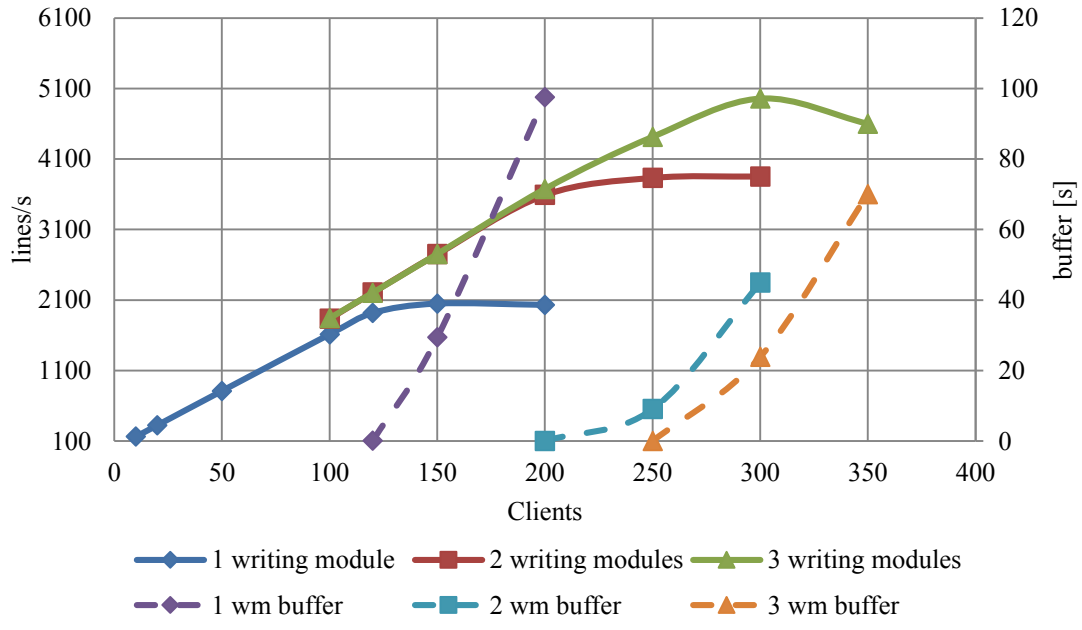

Figure 5.30. Results for writing tests with the writing module.

6. SUMMARY & CONCLUSIONS

The ever-growing internet data traffic and number of users are causing problems which are difficult to find and verify in a network which becomes more and more complex. QoE and QoS measurements, analysis and continuous monitoring are good methods to find and deal with these problems. Qosmet, developed at VTT for measuring QoS, is a powerful solution for network analysis. The main goal of this thesis was to create an optimal database architecture for QoS measurement data and to implement an interface between the database and the Qosmet solution.

Three different database schemas were designed and thoroughly analysed with writing and reading performance tests. Schema 1 has a simple structure with only one table containing all the data fields. Schemas 2 and 3 add more tables, having two and three tables respectively. Both schemas have one table dedicated to data fields that identify each individual measurement. The second table for schema 2 contains all the measured statistics. For schema 3, these measurement statistics data fields are divided into two separate tables according to their assumed importance.

To determine the writing throughput, 99 test cases with different numbers of writing threads and number of rows written, were performed for each of the three database schemas. The results for schemas 1 and 2 were very close to each other, whereas schema 3's throughput fell about 33% compared to the other two. The maximum write throughput of around 7,500 lines per second was achieved with 16 writing clients in schema 2. For writing tests, hardware performance of the database server was also measured and analysed. These results gave a good understanding of hardware requirements, bottlenecks and limits on high utilization situations.

Reading performance was measured with five different database queries, which all included four or more test cases. The queries represent some real-life use cases and test the schemas while keeping in mind the different roles of tables. The results for reading performance in schema 1 vary a great deal, and although the schema is best out of the three in some queries, it performs significantly worse in some. Therefore, it is found to be the worst in overall reading performance. Schemas 2 and 3 are much more consistent with regards to their results than schema 1, and both end up with about a 20% better total performance. Schema 3 has the best overall reading performance ahead of schema 2. To find the best database schema from the three tested, the results from both writing and reading tests were scored and the schemas were put in point order. Schema 2 ended up being the best, with constant results from both writing and reading. Schema 1 performed well in writing, but not as well in reading performance and was hence second best behind schema 1. Despite being the best in reading performance, schema 3 was the last of all compared schemas in overall results, because it had poor writing results.

A major part of the work was to design and implement an interface between the database and Qosmet solution. This writing module interface works as a third party listener for the Qosmet solution, receiving the measurement data from Qosmet measurement nodes. The writing module processes the received data and writes it to the database. Database writing performance was also tested together with the module. These tests were performed only with schema 2, as it was chosen to be the best one of the compared schemas. The results were in line with the previous writing test results and also indicated that the writing module implemented has a threshold of around 2,000 lines per second. This threshold can be exceeded by running several modules

simultaneously. These tests also gave valuable information about the limits of Qosmet when running a large number of measuring instances on one computer.

First and foremost, this work provided a working database solution for Qosmet and an interface between the database and Qosmet. With the help of the database solution, it is now possible to store QoS measurement data easily from dozens or even hundreds of simultaneous measurements. Also, results from long measurement campaigns can be stored reliably. The database storage makes it possible to develop adaptive or even automatized result analysing features to existing diverse QoS measurement solution. For big data storages, it is essential to have a well optimized and designed implementation and DBMS installation. Comprehensive performance testing has provided a great deal of valuable knowledge and experience about database optimization and features affecting the efficiency. All this can be taken into use on future development of the solution.

There are many ways to improve the performance of the database or optimize it for certain use case scenarios. We have speculated on the effect of various MySQL system variables when talking about the CPU, I/O and memory performance on writing tests. For example, it might be useful to experiment with the effect of number of write I/O threads or size of InnoDB buffer pool and other MySQL buffer allocations. Whereas there are possibilities of improving writing performance, there are also options to influence the reading performance. It became evident, that by changing the size of the query cache size, it is possible to optimize the database performance for reading purposes. Several choices to optimize either writing or reading performance were discovered, but we should not ignore seeking the best settings for combined performance where both writing and reading are performed simultaneously. Apart from working on finding the best database variable settings, optimizing the writing module code should not be put aside. There are signals about writing module being a bottleneck when analysing the CPU and I/O results for writing tests as well as performance tests with the module. In addition to optimizing the completed implementations, the most important step in the future is to add data analysing features to the Qosmet solution.

7. REFERENCES

[1] Internet live stats: Internet Users (2014) URL:http://www.internetlivestats.com/internet-users/. Accessed 15.12.2014.

[2] Cisco Visual Networking Index: Forecast and Methodology, 2013-2018 (2014) URL:http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html. Accessed 15.12.2014.

[3] Qosmet – A solution for measuring Quality of Service (2014) URL:http://www.cnl.fi/qosmet.html. Accessed 13.11.2014.

[4] A. Silberschatz, H. F. Korth and S. Sudarshan (2006) Database System Concepts. New York, McGraw-Hill.

[5] J. D. Ullman and J. Widom (2002) A First Course in Database Systems. Upper Saddle River (NJ), Prentice Hall.

[6] MySQL (2014) URL:http://www.mysql.com/. Accessed 19.05.2014.

[7] MariaDB (2014) URL:https://mariadb.org/. Accessed 29.04.2014.

[8] L. M. Nyman (2013) Freedom and forking in open source software: The MariaDB story. Nordic Academy of Management 2013 (ISSN 2298-3112) Nordisk Företagsekonomisk Förening.

[9] R. Hecht and S. Jablonski (2011) NoSQL evaluation: A use case oriented survey. Proc. International Conference on Cloud and Service Computing (CSC), 2011. DOI: 10.1109/CSC.2011.6138544.

[10] B. G. Tudorica and C. Bucur (2011) A comparison between several NoSQL databases with comments and notes. Proc. 10th Roedunet International Conference (RoEduNet), 2011. DOI: 10.1109/RoEduNet.2011.5993686.

[11] C. Strauch (2011) NoSQL databases. Stuttgart Media University. URL:http://www.christof-strauch.de/nosqldbs.pdf. Accessed 28.04.2014

[12] Apache Hadoop (2014) URL:http://hadoop.apache.org/. Accessed 28.04.2014.

[13] Hadoop Wiki | PoweredBy (2014) URL: http://wiki.apache.org/hadoop/PoweredBy. Accessed 12.08.2014.

[14] Apache Cassandra (2014) URL:http://cassandra.apache.org/. Accessed 28.04.2014.

[15] Apache Hbase (2014) URL:http://hbase.apache.org/. Accessed 28.04.2014.

[16] M. N. Vora (2011) Hadoop-HBase for large-scale data. Proc International Conference on Computer Science and Network Technology (ICCSNT), 2011. DOI: 10.1109/ICCSNT.2011.6182030.

[17] MongoDB (2014) URL:https://www.mongodb.org/. Accessed 29.04.2014.

[18] Zhu Wei-ping, Li Ming-xin and Chen Huan (2011) Using MongoDB to implement textbook management system instead of MySQL. Proc. IEEE 3rd International Conference on Communication Software and Networks (ICCSN), 2011. DOI: 10.1109/ICCSN.2011.6013720.

[19] A. Boicea, F. Radulescu and L. I. Agapin (2012) MongoDB vs oracle -- database comparison. Proc. Third International Conference on Emerging Intelligent Data and Web Technologies (EIDWT), 2012. DOI: 10.1109/EIDWT.2012.32.

[20] Database for objects (2014) URL:http://www.db4o.com/. Accessed 12.08.2014.

[21] K. E. Roopak, K. S. S. Rao, S. Ritesh and S. Chickerur (2013) Performance comparison of relational database with object database (DB4o). Proc. 5th International Conference on Computational Intelligence and Communication Networks (CICN), 2013. DOI: 10.1109/CICN.2013.112.

[22] S. Ray, B. Simion and A. D. Brown (2011) Jackpine: A benchmark to evaluate spatial database performance. Proc. IEEE 27th International Conference on Data Engineering (ICDE), 2011. DOI: 10.1109/ICDE.2011.5767929.

[23] E. Kreyszig (1999) Advanced Engineering Mathematics. John Wiley & Sons.

[24] R. Seppänen, S. Tiihonen, M. Kervinen, R. Korpela, L. Mustonen, A. Haavisto, M. Soininen and K. Varho (1996) MAOL-Taulukot. Helsinki, Otava.

[25] R. Andresen (2004) Monitoring linux with native tools. Proc. 30th Annual International Conference of the Computer Measurement Group, Inc., 2004.

[26] M. Ahmed, M. M. Uddin, M. S. Azad and S. Haseeb (2010) MySQL performance analysis on a limited resource server: Fedora vs. ubuntu linux. Proc. 2010 Spring Simulation Multiconference. Orlando, Florida, DOI: 10.1145/1878537.1878641.

[27] Tom's Hardware, Write Throughput Average: h2benchw 3.16 (2014) URL: http://www.tomshardware.com/charts/enterprise-hdd-charts/-04-Write-Throughput-Average-h2benchw-3.16,3376.html. Accessed 11.12.2014.