MICROSOFT SQL SERVER 2005: RETHINKING CODE PLACEMENT

Jacob Elkjær Hansen
Morten Kirkegaard Hansen
Erik Hejlskov

Department of Computer Science
Aalborg University

14^{TH} of June – 2006

Aalborg University
Department of Computer Science, Fredrik Bajers Vej 7E, DK-9220 Aalborg Øst

Title:

Microsoft SQL Server 2005:
Rethinking Code Placement

Thesis Period:

DAT6,
February 1, 2006 – June 14, 2006

Thesis Group:

d631a

Group Members:

Jacob Elkjær Hansen
Morten Kirkegaard Hansen
Erik Hejlskov

Supervisor:

Bent Thomsen

Number of Copies: 7

Number of Pages: 98

Abstract

This thesis examines the possibilities introduced by the integration of CLR in Microsoft SQL Server 2005. The focus of the examination is to determine how the CLR integration affects the design of a distributed system concerning the performance and scalability criteria. The goal of the thesis is to define a metric empirically, which is able to determine whether a given functionality achieve the best performance and scalability on the application server or database.

In order to define the metric, the parameters affecting performance and scalability are determined. A subset of these parameters is varied in a full factorial experimental design to determine the impact of each parameter. The measurements obtained from the performed experiments are compared statistically. Thereby, determining the differences between code deployed on the application server and the two technologies – T-SQL and SQLCLR. Subsequently, the obtained result is used to define the metric.

In order to make the metric easy to use for the developer, it has been implemented into a tool with a simple user interface, where the different parameters can be specified.

Preface

This thesis is the product of a four–month process of studying and developing a metric that can help developers of distributed enterprise systems decide where functionality should be placed; application server or database. The focus has been on examining the performance and scalability differences between the application server and the two technologies – T-SQL and SQLCLR. We have studied existing works in the fields of performance and scalability to help outline the parameters included in our metric. The metric proposed is established by empirical studies.

It is expected the reader has basic knowledge in computer science and experience in computer programming and databases. Especially experience in the C# and T-SQL languages as well as Microsoft SQL Server 2005, is an advantage. It should be noted however, that different qualifications are needed for different chapters in the thesis. The target audience for the *Microsoft SQL Server 2005* and *Test* chapters is primarily system developers and programmers, whereas the rest of the chapters are for project leaders and decision makers. A summary of the thesis is presented in Appendix N.

In addition, *Italic* font is used when referring to the factors and parameters used in our metric.

Morten Kirkegaard Hansen

Jacob Elkjær Hansen

Erik Hejlskov

Contents

Introduction

A hot topic when developing distributed systems is the issues around utilizing the available resources most optimal, concerning placement of functionality. In distributed systems, the system architecture often resembles a three–tier architecture, consisting of a client–tier, application–tier, and database–tier. Each of these tiers deploys a number of distinct services, such as user interaction in the client–tier, data processing and functionality in the application–tier, and storing of data in the database–tier. As we discovered in our DAT5 project [1], there exist a number of object–relational mapping tools that can not only boost the performance of the system, through e.g. caching of data and queries at the application–tier, but also bridge the gap between the object–oriented and relational paradigms.

However, with the release of Microsoft SQL Server 2005 (MSSQL2005), the data processing and functionality, often placed in the application–tier, is possibly subject to change from primarily being placed in the application–tier. With the release of MSSQL2005, developers now has the option of writing managed code in any of the .NET supported languages, such as Visual Basic .NET or C# and deploy it on the database. The managed code in form of stored procedures (SP), functions, aggregates, types, or triggers can then be executed within the database engine using the hosted Common Language Runtime (SQLCLR). Thus, it is now possible to develop the functionality for the system in both the application–tier and database–tier and with an added benefit of programming in the same programming language. Previously, this was usually done using T-SQL in the database–tier and a .NET language in the application–tier. Thereby, relocating functionality from the application–tier to the database–tier in order to e.g. improve performance does not even require knowledge of multiple languages.

The new option of placing functionality on the database–tier is especially important, since one of the most important goals in distributed software development is to achieve high performing software that scales well. Most often developers try to ensure the software satisfies this goal by utilizing various techniques such as metrics and tests, thereby ensuring the software comply with the stated criteria and the product specification.

History shows the consequences of neglecting performance in systems combined with bad planning, as such, effort should naturally be put into planning when developing new software. One example of a system that performs poorly is the Danish employment exchange's control system *AMANDA*. The whole development process was riddled with flaws and inaccuracies that in the end resulted in a doubling of the budget to around 500 million DKR, poor performing database connectivity, unsatisfactory functionality, and a highly complex user interface. In effect, this resulted in reduced productivity of around 30–50% equalling approximately two millions DKR per day [2]. The morale of the story is quite self–explanatory and especially the issues around the database connectivity is of special interest to us, which for *AMANDA* was highly ineffective, resulting in long response times and erroneous searches.

One of the problems the hosted SQLCLR introduces is that no metric or tool currently exist that is able to provide the developer with information about when it is beneficial to place a particular piece of functionality in the application–tier or the database–tier. Thus, the developer is currently only provided with guidelines from which he must decide whether functionality should be placed on the application–tier or the database–tier and what technology should be used in the case of the database–tier [3]. We want to define a metric that can help make this decision, by performing a number of experiments, which cover a range of different scenarios. As such, the developer is provided with a tool that can help determine the placement of functionality as early as the design phase, thereby minimizing the cost that any changes in decisions might impose.

Problem Statement

> *Given T-SQL and the integration of the SQLCLR in MSSQL2005, when is it beneficial, with respect to performance and scalability, to place functionality on either the application server or the database and should T-SQL or SQLCLR be used on the database. Furthermore, define a metric that can be used by developers as early as the design phase to help decide where to place the functionality and which technology to use.*

Focus and Scope

As specified in the problem statement, we only examine T-SQL and SQLCLR even though it is also possible to use Extended Stored Procedures (XP) on the

database. However, with the integration of SQLCLR, the use of XP is declining, since SQLCLR can perform the same operations as XP, but with better security and reliability [3]. Because of this, XP is not included in the comparison. Furthermore, for fair comparison the database and application server use the same operating system. The choice was made to use Windows Server 2003, which has support for the .NET Framework and is the application server of choice for MSSQL2005 in TPC [4]. As such, the metric is not generic but only applicable for MSSQL2005 and for applications developed in the C#, since it is the most popular .NET language [5].

As already mentioned, the task is to develop a metric, which can give an indication of whether a piece of functionality in a distributed system should be deployed in the application–tier or database–tier. A 3–tier model is examined but only locality issues between the application–tier and database–tier will be determined. The application–tier is referred to as application server throughout the report, and the database–tier is referred to as database throughout the report. The metric can be used as early as design time and determines the location for best performance. When discussing functionality, it should be understood as a part of a larger system that performs a specific task. Furthermore, it has to concern communication with a database, as either ad–hoc SQL or manipulation or validation of the data. However, it does not necessarily have to be e.g. an entire class; it can also be individual methods or a subset of a method. For example, a rather simple piece of functionality is listed in Section 5.2, which traverses the data obtained from a query and returns a string. In addition, when implementing the functionality for the database, it is done using SP. Focusing only on SP is because the functionality on the application server should correspond to the functionality on the database, i.e. it should be able to retrieve, insert, and delete data, which is only possible by using SP.

In particular when developing the metric, we are interested in the performance and scalability criteria, since they are critical for the operation of distributed systems, both for the application server and the database. However, other criteria applicable for distributed systems could have been considered as well, such as those specified in the International Organization for Standardization (ISO) 9126 standard; reliability, interoperability, security, portability, and maintainability [6].

For our metric, we intend to target the small to medium–sized enterprise (SME). Not choosing to span the metric to large enterprises is because of limited resources such as hardware and software. Furthermore, to develop the metric all the parameters influencing performance and scalability must be evaluated for different levels to provide an accurate metric. As this approach results in a large amount of combinations, only a number of parameters are evaluated for a set of levels in

this thesis. The parameters not evaluated are left out for various reasons, either through a preliminary analysis or because of unpredictability.

The following is a definition of the performance and scalability criteria used in the development of the metric. Even though there is a distinction between performance and scalability, it does not mean they are not closely related. This is because performance issues might not become apparent until the system is put under an increased load and scalability cannot be defined unless performance requirements have been specified.

Performance

As the computer industry becomes more competitive, the developers strive to develop systems with high performance that still retain high quality and low cost [7]. Therefore, the performance of a computer system becomes a key criterion during all phases of development. In this thesis, performance is defined as a measure of response time. Thus, if a scenario has a lower response time than another, it can be said to perform better.

Currently, best practice for tuning software, hardware, and numerous other performance related factors already exists plentiful. However, our metric provides a measurement for the best performance trade–off by specifying whether it is most beneficial to place functionality on the application server or database. By using it, the developer is able to decide where the functionality should be placed.

The parameters used in our metric are *computations*, *number of queries*, *type of queries*, *hardware*, *workload unit*, *data volume*, and *locality*, which are described in Section 3.2. In order to determine how each of the parameters affects performance, four measures are analysed, which are described in detail in Section 4.5; response time, throughput, CPU utilization, and network utilization.

Scalability

A dominant theme in the development of distributed systems is that systems should be able to operate effectively at different scales. Scalability relates to the ability of a system to remain effective as the workload is increased, by providing more resources [8]. Since it is not possible in this thesis to provide more resources to the hardware configuration because of software limitations, scalability is defined as the ability to handle increased load compared to response time.

To help achieve scalability in enterprise applications, there are two possibilities; either scale out or scale up.

- **Scaling out**
  Adding more machines to a system, thereby distributing the load on the system across more than a single machine. However, even though several machines are used, they still operate as a single machine.

- **Scaling up**
  Adding more resources to a single machine, such as adding either additional or faster processors, faster or more memory or disk drives. Thereby, it is able to handle an increased load while still operating as a single machine.

Each of these two options poses benefits and drawbacks. By scaling up and maintaining a single machine, the fault tolerance is decreased, since there is a single point of failure and it is more difficult to do maintenance without affecting the operability of the system. However, compared to scaling out, there is less management and no overhead from synchronizing several machines. These benefits and drawbacks shall therefore be considered when buying more resources to a system.

In our metric, neither scaling out or scaling up is possible, since resources are not available to change the hardware configurations. Instead, the metric considers two different hardware scenarios: one with a heavy–duty database and a lightweight application server, and one with a heavy–duty application server and a lightweight database. The hardware scenarios are further described in Section 4.3.3. The reason for having two different hardware scenarios is to examine how hardware affects the placement of functionality. This examination does not tell how *locality* is affected by scaling up or scaling out, but whether the majority of resources should be placed on the database or the application server.

Outline

The thesis is divided into five parts; *Background Knowledge*, *Metric Development*, *System Development*, *Discussion*, and *Appendix*.

Background Knowledge contains the knowledge gained by examining the work already made by others. Furthermore, is a description of various tools and applications utilized throughout the thesis. Lastly, the concepts and functionality available in MSSQL2005 related to SQLCLR and T-SQL are introduced.

Metric Development presents an outline of the metric proposed in this thesis. Furthermore, a description of the included and excluded parameters is presented. The included parameters have been determined to affect performance and scalability

the most, whereas the excluded parameters have minimal influence. The parameters serve as the basis for the metric and are used in the experimental design, which contains the design decisions for the experiments performed to outline the metric.

System Development examines the required steps in order to develop the application simulating the experiments of the experimental design. Furthermore, a presentation of code snippets for each of the three possible placement of functionality is given. The results gathered from performing the experiments are presented and the various findings are discussed and concluded on. Lastly, the metric is defined using the results gathered from the experiments and an implementation of a tool, which uses the metric for determining the best locality of the functionality.

Discussion contains a discussion of the validity of the approaches and results in this thesis, as well as an overall conclusion of the thesis. The *Discussion* is ended with an examination of the possible further developments of this thesis.

The *Appendix* lists various data, statistics, configurations, and graphs used throughout the thesis.

Part I

Background Knowledge

Chapter 1

Related Work

This chapter contains a description of other work that is used in the development of the metric, including a presentation of the Software Performance Engineering (SPE) techniques and the TPC-H benchmark, which is utilized in this thesis. Finally, the chapter contains a presentation of existing guidelines for deciding where functionality should be placed, the Compiere ERP solution, and the statistical tool JMP.

1.1 Software Performance Engineering

SPE is an example of an existing technique used to improve software performance in the same area as the metric presented in this thesis. SPE [9] is a collection of techniques that helps developers to create software that fulfils the required performance objectives, such as response time and throughput. Among the techniques described in SPE is an estimation of the amount of instructions used by a particular functionality. We have used these considerations about instructions to outline the *computations* factor.

When the SPE techniques are used on a given application it is not applied on the entire application as the techniques are time consuming. Instead, only a subset of a given application is analysed. More specifically, the Pareto principle is used to find the 20% of the software that is executed 80% of the time [9]. SPE differs from standard software development in one particular way, namely the phase where performance issues are considered. In standard software development, the performance issues are usually dealt with after the implementation. The software is tested against the outlined requirements and tuned as needed. SPE however, validates the performance objectives at all stages of the development process. The

benefit of this approach is that if a flaw is discovered at design time, the cost of correcting it is less than discovering the same flaw after the implementation.

The SPE collection of techniques contains principles for improving software performance, recommends patterns and describes anti–patterns for the design, techniques for eliciting performance objectives, techniques for gathering data needed for evaluation, as well as evaluation guidelines for the development process. In order to estimate the performance, the software is modelled and decorated using the approaches described in the following nine steps. The nine steps illustrated in the activity diagram in Appendix G.1 are used throughout the software development process, utilizing the various techniques available in SPE.

1. **Assess performance risk**
   Different risks might appear during the process of software development. The developers need to identify these risks and determine the impact of each of them. When a risk is identified the probability of happening as well as the severity of damage is determined.

2. **Identify critical use cases**
   The use cases are identified during the software analysis. Some of these use cases are considered critical for the software. The critical use cases are found by examining the software with e.g. the rule of Pareto in mind.

3. **Select key performance scenarios**
   Each of the critical use cases consists of a set of actions performed by the user. Among these actions, only a subset has a significant impact on the performance of the software, which is identified like the critical use cases.

4. **Establish performance objectives**
   When the key scenarios are identified, each of them should have quantitatively specified criteria associated with it, such as response time and throughput.

   The steps 5 through 8 are repeated until performance problems are solved

5. **Construct performance models**
   In order to evaluate the scenarios, each of them is converted into an execution model. The models specify the processing steps for the scenarios and characterize the performance of the software according to the included factors, such as workload or multiple users.

6. **Determine software resource requirements**
   Software resource requirements concern the main resources utilizing com-

putational capacity, such as number of access to database and number of executed instructions.

7. **Add computer resource requirements**
   When the software resources are identified, an analysis of the extent of computer resource consumption is needed. Each of the software resources utilizes hardware resources such as CPU, disk I/O, or network bandwidth.

8. **Evaluate the models**
   The results of the models contain information about the execution time of each of the scenarios as well as the resource utilization. Hence, a thorough analysis of the influence of the factors is needed in order to minimize the risk for a potential performance problem.

9. **Verify and validate the models**
   The constructed models are continuously verified and validated during the software development. The verification and validation of the models encourage the developers to reflect the performance of the software as accurate as possible.

1.2 Transaction Processing Performance Council

TPC is a non–profit corporation that develops performance benchmarks for transaction processing and databases. A transaction process is for example an update of a database with new data such as airline reservations or inventory control [10]. Performance is in most cases measured as transactions per unit of time. The benchmarks are widely available and anyone who wants to run one of the TPC benchmarks can do so freely. However, the results of the benchmarks can only be published if TPC approves it, in order to protect the validity of the benchmarks.

The TPC-H is a decision support benchmark and consists of a number of business–oriented ad–hoc queries and concurrent data modifications. The benchmark illustrates appropriate and commonly used queries and data scenarios, with a focus on large volumes of data and complex queries. The TPC-H database schema, which is illustrated in Appendix D, models a database corresponding to a company that sell, distribute, and manage a product. When developing our metric some of the ideas and concepts from the TPC-H benchmark are used. However, instead of fully implementing the benchmark, only the database schema and two queries, which are relevant for this thesis, are used.

1.3 SQLCLR–Based Programming

In the article *"Using CLR Integration in SQL Server 2005"* [3] different SQLCLR features are described and compared to T-SQL and XP. Furthermore, the article provides high–level guidelines for the issue of code placement on the application server vs. the database. The guidelines in the article indicate the SQLCLR is an alternative to the procedural features of T-SQL and placing logic in the middle tier.

A situation where the SQLCLR performs better than T-SQL is when performing complex calculations. Another use for the SQLCLR is for procedural logic to evaluate tabular results that can then be queried in the `FROM` clause of a `SELECT` statement or in another Data Manipulation Language (DML) statement. However, the SQLCLR should not be used to write procedural code that is expressible by the declarative features in T-SQL, as the performance of SQLCLR is lower under this circumstance.

One of the differences between SQLCLR and T-SQL becomes apparent when writing data access code. This is due to the approach of managed code, where queries are represented by dynamic strings, that are parsed as parameters to methods in the ActiveX Data Objects .NET (ADO.NET) Application Programming Interface (API). In T-SQL, queries are embedded in the procedural code, but the performance of data access in T-SQL is better than that of SQLCLR.

A scenario where SQLCLR code outperforms T-SQL is when doing processing on forward–only and read–only row navigation. This navigation is in T-SQL implemented using a `CURSOR`, which is faster than the `SqlDataReader` class, but any additional processing of data, is faster with the SQLCLR. Still, T-SQL performs better than SQLCLR when submitting SQL statements without any need of additional processing. This is due to the overhead of the SQLCLR traversing additional layers of code to switch between managed code and SQL statements.

A scenario where T-SQL and the SQLCLR perform equally well is when transferring results back to the client. T-SQL transfers the rows produced by the `SELECT` statement and managed code transfers the result using the `sqlPipe` object.

A benefit of SQLCLR is that managed code is very similar to the code on the application server, which makes it more portable than T-SQL. The article also states however, that developers should be careful not to deploy too much functionality on the database, due to the additional load this implies. Therefore, the article provides two guidelines for code to be placed on the database:

- **Data Validation**
  By using the database for data validation the functionality concerning data

validation is centralized and not distributed to different tiers in the system. Hence, making the code for data validation easier to maintain.

- **Network Traffic Reduction**
  If a lot of data has to be extracted from the database but only a small amount of the data is returned as a result of e.g. an analysis of the data, the system could benefit from putting the functionality on the database. Thereby, reducing unnecessary network traffic.

1.4 Enterprise Resource Planning

Enterprise Resource Planning (ERP), are systems used to integrate and support various business processes. The main goal of ERP systems is to integrate the departments and functions in a company into a stand–alone system that spans across all of the company's needs. Because of this, ERP systems contain a large range of modules to help in a number of business activities, such as accounting, sales, inventory, quality management, and human resource management, to name a few. One of the benefits of this approach is that by combining all these modules into a single system, it is easier for different branches in a company to communicate between each other [11]. Because ERP systems are complete solutions, it is advantageous to use as basis for statistical analysis of general applications focused for e.g. our SME target group. Furthermore, the article in Section 1.3 indicates that applications like the ERP solutions contain functionality that might benefit from the integration of SQLCLR on the database.

1.4.1 Compiere

Compiere is an open source ERP and Customer Relationship Management[1] (CRM) solution targeted for SME, distribution chains, and franchise systems [12]. Some of the modules included in Compiere are Customer Relations, Performance Analysis, Supply Chain, and Web Store. Since Compiere covers the SME target group of the metric developed in this thesis, the system is used to gather statistics to supplement some of the choices taken concerning the parameters in our metric. The reason for using the Compiere solution is as the article presenting Safe Query Objects[2] [13] states:

[1] ERP is often referred to as a back-office system with no direct contact with the customer, whereas CRM is a front-office system, which deals with the customer, such as a web store.

[2] Safe Query Objects is a technique for representing queries as statically typed objects and uses the Compiere solution for evaluation.

> *"The Compiere ERP application provides a sophisticated range of queries for evaluation."*

Examining the Compiere solution, reveals 871 uses of `SELECT` statements spanning from joins, to dynamic and parameterized queries. As such, it appears to provide a large range of queries and thus, the implemented functionality containing these queries can be expected to span across a reasonable range of different functionalities. An examination of these functionalities enables us to extract the characteristics for further use.

1.5 JMP

JMP is an application developed by SAS Institute [14], which we use to perform various statistical analyses of data. The application provides build–in macros for common statistical designs such as full factorial design, which is described in Section 4.1. When sample data is loaded into the application, it is possible to analyse it using the provided functionality. The application is able to calculate mean, variance, standard deviation etc.

Chapter 2

Microsoft SQL Server 2005

MSSQL2005 is the latest in a line of databases produced by Microsoft, which has previously collaborated with Sybase [15] to develop databases. However, the partnership was terminated and a new line of databases was created exclusively by Microsoft, ranging from the early version 4.2 to 2000 and lately 2005. Some of the most significant improvements in MSSQL2005 compared to MSSQL2000 are the integration of the .NET CLR, XML support, a new management tool set, and an enhanced T-SQL language [16]. A description of the T-SQL language, as well as the integration of the CLR in MSSQL2005 is presented in this chapter, with an emphasis on functionality used in this thesis.

2.1 Transact-SQL

T-SQL is an extension to the SQL language and complies with the American National Standards Institute SQL-99 standard [17]. T-SQL is integrated by Microsoft as a native programming language in their database and is commonly used by database developers to write scripts and procedural code. Even though T-SQL is a programming language with basic logical operations, local variables, and support for creating, calling, and using functions it is not nearly as powerful and expressive as other similar languages such as Pascal and ALGOL.

T-SQL has a large array of functionality and continues to be extended in each new version of the SQL Server. Some noteworthy additions in MSSQL2005 are the `TOP` operator, new join types for the `FROM` clause, improved error handling, and random data sampling [17]. However, instead of making a thorough description of the T-SQL language, there is a focus on the basics of T-SQL as well as SP, which are relevant for understanding the T-SQL functionality used to develop our

experiments.

2.1.1 Language Constructs

In T-SQL, there are three classes of statements also known from standard SQL:

- **Data Definition Language (DDL) statements**
  Used to create elements in the database
- **Data Control Language (DCL) statements**
  Used to determine access privileges
- **Data Manipulation Language (DML) statements**
  Used to modify or query data

In this section, as well as the rest of this thesis, the focus is on the DML statements, which include the `SELECT`, `UPDATE`, `DELETE`, and `INSERT` statements.

T-SQL provides, unlike standard SQL, a control-of-flow, which includes several keywords, such as `BEGIN`, `END`, `BREAK`, `IF`, `ELSE`, `RETURN`, and `WHILE` [18]. Thus, it is possible to construct structures that are more complex. It is for example possible to enclose a series of statements in a `BEGIN...END` block and break out of the block with `BREAK`. It is also possible to execute a statement repeatedly as long as a condition is true with the `WHILE` command and evaluate conditions with the `IF` command. Another enhancement is local variables, which is used to contain values of either system–supplied or user–defined types. The local variables are useful in a single script, but not outside the script, since global variables are not supported. An example of a simple block of statements is illustrated in Listing 2.1. Line 4, contains a declaration of a local variable of type `decimal` called `@CustOrder`. The variable in line 5 is assigned the value returned by the query.

2.1.2 Stored Procedures

A SP should be understood as a small program, which is stored physically on the database. To write a SP in MSSQL2005, a language is required, which can be either T-SQL, covered here, or a .NET language such as C#, which is covered in Section 2.2.3. The SP is able to:

- Accept input parameters

- Return single or multiple output values
- Return a value indicating whether the SP was a success or failure
- Perform operations on the database or call other SP

SP can be divided into two classes, the system–defined SP and the user–defined SP. SP is in the remaining of the thesis referring to user–defined SP, whereas system–defined SP is named so explicitly. System–defined SP, which are installed with SQL Server, are actually executed continuously in the database environment. Actions such as table modifications and administrative operations are in fact system–defined SP executed each time one of these actions is triggered.

The SP can be customized to suit the need of the developer as illustrated in Listing 2.1. The purpose of this SP is to retrieve the prices of all the orders made by a specified customer. The SP uses standard T-SQL syntax and operates on the TPC-H database.

One of the benefits of using a SP is the direct execution in the database engine; meaning database requests are usually processed faster, since the SP has direct access to the data. Furthermore, since it is possible for a SP to do computations and operations on large sets of data, the result returned to the user can be a subset of the data, thus limiting the amount of data transferred across the network. Other benefits include security mechanisms and sharing of application logic.

Listing 2.1: T-SQL Stored Procedure

```
1  CREATE PROCEDURE usp_cust_order (@CustID int)
2  AS
3  BEGIN
4    DECLARE @CustOrder decimal
5    SELECT @CustOrder = O_TOTALPRICE
6    FROM dbo.ORDERS AS o
7    INNER JOIN dbo.CUSTOMER AS c ON o.O_CUSTKEY = c.C_CUSTKEY
8    WHERE C_CUSTKEY = @CustID
9    RETURN @CustOrder
10 END
```

2.2 SQLCLR

A new feature in MSSQL2005 is the integration of the Microsoft .NET Framework 2.0 CLR into the database. The integration of the CLR allows developers to write SP, types, functions, triggers, and aggregates in any of the 25+ supported .NET languages, such as Visual Basic .NET, C++, and C#. The SQLCLR is an

alternative to XP, but with better security, a larger amount of class libraries, and object–oriented capabilities among other things. The integration of the SQLCLR was made with the following design goals in mind: [16, 17, 19]

- **Security**
  Database administrators should have full control of the .NET code running on the database and have a simple security mechanism through the introduction of the `SAFE`, `EXTERNAL ACCESS`, and `UNSAFE` permissions.

- **Reliability**
  Critical applications are expected to have an uptime of 99.999% or better [17]. Therefore, the user code should not be able to perform operations that e.g. overwrite the memory buffer or internal data structures, which is possible with XP, as these operations might bring down the database.

- **Scalability**
  It is common the database must be able to support multiple concurrent users. Hence, it is important the .NET user code containing functionality including threading or high memory demands do not affect the overall scalability of the database.

- **Performance**
  Performance is very important in enterprise applications, thus code written in a .NET language and deployed inside the database must perform just as well as code deployed outside the database.

2.2.1 The Microsoft .NET Framework 2.0 CLR

The CLR is a virtual machine much like the Java Virtual Machine is for Java and is an execution environment that handles compilation, loading, and execution of a hosted application. Furthermore, it provides several services such as memory, thread, and I/O management, as well as type–safety, garbage collection, and security. When a given application is executed in the execution environment, the code is referred to as managed code, whereas unmanaged code refers to code that is run outside and not under control of the CLR, such as Win32 functions or COM objects. This is because managed code in the CLR resides in an application domain, which should be understood as a lightweight process, unlike Win32 functions and COM objects, which are isolated in different memory address spaces. The application domain is kept under control of the CLR, which also ensures that other application domains do not use the same memory address space. Because

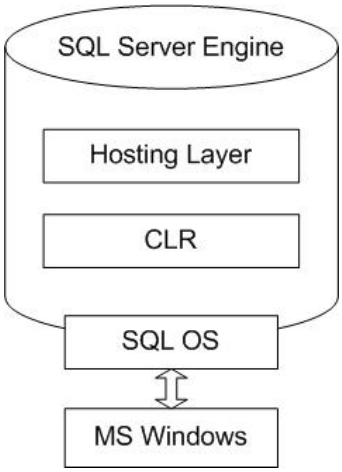


Figure 2.1: CLR in MSSQL2005

of the CLR's architecture, it is possible to host it by other processes, in this case MSSQL2005.

For a more detailed description of the CLR, consult our previous project [1].

2.2.2 Structure of the SQLCLR

The SQLCLR is hosted in MSSQL2005 as illustrated in Figure 2.1, where the operating system (SQL OS) handles connections to the database as well as memory, threads and synchronization of the database. The hosting layer handles communication and coordination between the SQLCLR, the SQL Engine, and the SQL OS. The coordination includes memory management, security, garbage collection, deadlock detection, and assembly loading of .NET code. [20]

Because the MSSQL2005 and the SQLCLR have different internal models for memory and thread handling as well as security, an extension to the hosting API in the .NET framework had to be made. This extension allows the database and the SQLCLR to work together by having the database control the resources available or making recommendations on how they should be managed. This essentially means the database at all times has control of memory usage by either accepting or rejecting memory requests made by the SQLCLR. The benefit of using this approach is the database and the SQLCLR do not compete for the memory, since the database always has higher precedent unless otherwise specified, as such the overall memory usage can be within the limit specified by the user.

Furthermore, the database also handles any thread scheduling for the SQLCLR,

which means the SQLCLR has to call the database's API in order to create threads, no matter if the thread is used for internal use or for executing user code. In order for the database's thread scheduler to schedule other tasks if a thread is waiting on a synchronization object, the SQLCLR calls the database synchronization objects [16].

2.2.3 Stored Procedures

With the introduction of the SQLCLR, it is possible to write SP in any of the .NET supported languages besides T-SQL, as described in Section 2.1.2. The SP written in a .NET language are implemented as `public static` methods and are able to return various results, where the return type of the methods can be either `System.Int32`, `SqlInt32`, or `void`. Furthermore, and probably more important; it is possible to return string messages and tabular results.

In order to return a tabular result from a SP, the `Send()` method in the `Pipe` class is used as illustrated in Listing 2.2 in line 12. The SP is invoked in the same manner as a standard T-SQL SP.

Listing 2.2: SQLCLR Stored Procedure

```
1  ...
2  public partial class StoredProcedures
3  {
4      [Microsoft.SqlServer.Server.SqlProcedure]
5      public static void CustOrder(int custid)
6      {
7        using(SqlConnection connection = new SqlConnection("
             context connection=true"))
8        {
9           connection.Open();
10          SqlCommand command = new SqlCommand("SELECT
               O_TOTALPRICE FROM dbo.ORDERS AS o INNER JOIN dbo.
               CUSTOMER AS c ON o.O_CUSTKEY = c.C_CUSTKEY WHERE
               C_CUSTKEY = " + custid + "", connection);
11          SqlDataReader r = command.ExecuteReader();
12          SqlContext.Pipe.Send(r);
13       }
14     }
15 };
```

The functionality implemented in Listing 2.2 corresponds to the SP implemented using T-SQL, which was illustrated in Listing 2.1. As Listing 2.2 illustrates, the implementation of a SP in SQLCLR follows standard language syntax, in this case C#. However, the connection is modified to a `context connection` in order to achieve a better performance when running the SP on the database. The `context connection` is specified in line 7.

2.2.4 The SqlContext Class

Accessing data usually involves a data provider for the ADO.NET such as Open Database Connectivity or Object Linking and Embedding DB providers. These providers run outside the database server and make an explicit connection to the database to send queries and retrieve results. In MSSQL2005, a new data provider has been introduced that runs inside the database. The provider is able to access the database directly through the context in which it runs, which is done using different classes in the ADO.NET data provider model. Most of these classes are located in the `Microsoft.SqlServer.Server` namespace and an example of an inside data provider is the `SqlServer` provider. The different data providers need to implement the same interface, thereby the same code can be used with different data sources and the only code differing is the code for opening the connection. This specification is called the General Programming Model for Managed Providers (GPM) and ensures that with relative ease it is possible to move functionality from the application server to the database [17].

The `SqlContext` class is a part of the `SqlServer` provider. When running on the database, SP and functions are executed as part of the user's connection, thus a new connection is not required, thereby removing unnecessary network overhead. The `SqlContext` class is a helper class that retrieves the particular context and calls the corresponding methods [21]. To use the existing connection, the `context connection` has to be set to true, as specified previously in line 7 in Listing 2.2. There are however a few limitations to using the `context connection`, such as only having one active `context connection` at a time for each client connection and no support for Multiple Active Result Sets [22].

Part II

Metric Design

Chapter 3

Metric

This chapter contains a description of the parameters included in the metric, the parameters not included, and considerations that have affected the development of the metric. The description is a general presentation of the parameters as well as the reason(s) for either including them or excluding them. In the two following chapters, there is a differentiation between the following terms:

- **Experimental Design**
  The overall design strategy, for example full factorial or fractional factorial.
- **Experiment**
  One of the experiments performed in the experimental design.
- **Parameters**
  All that influences the performance and scalability in the metric.
- **Factors**
  A subset of the parameters that are examined in the experiments.
- **Levels**
  The different values or ranges for the factors and parameters.

3.1 Preliminaries

The purpose of the thesis is to determine in which situations functionality should be placed on either the application server or the database concerning performance and scalability. To make the comparison fair, it is also necessary to determine

whether the functionality on the database should be written for T-SQL or SQL-CLR.

In order to create a metric that with 100% accuracy determines where the functionality should be placed, all possible scenarios would have to be explored. Instead, the metric gives an indication, since the experimental design used to develop the metric is focused on a number of specific factors with a limited set of levels. Therefore, the metric is only valid concerning the factors and levels that are evaluated in the experimental design. This obviously limits the validity of the metric; however, the choice was made because the metric should not be limited to a specific scenario, but on the other hand exploring all possible scenarios would be too vast. Instead, the middle ground, where a number of parameters are evaluated, is chosen.

3.2 Metric Outline

The parameters selected in this chapter, is the foundation for the experiments described in Chapter 4. One of the purposes of the experiments is to determine how much each of the parameters affects performance and scalability in order to establish the weight for each of the parameters in the metric. Another purpose is to determine whether any of the parameters causes bottlenecks and thereby introduces deviations in response time.

Since the experimental design is limited to a number of parameters, the parameters not included have to be – where possible – set to a specific level, which remains unchanged during the execution of the experiments in the experimental design. The reason for doing so is to reduce uncertainties concerning variations between the experiments performed on the different combinations of parameters. As such, even though a parameter is excluded, the level for the parameter is still important to specify. This is because situations might occur where the metric indicates a faulty answer due to one of the excluded parameters being completely different than the system used on.

In situations where the metric exceeds the levels specified in the experimental design for both included and excluded parameters, the metric will likely give a misleading indication.

The parameters described below are discovered by examining performance guidelines for databases [23, 24]. The parameters are included in the metric if they are essential to performance and scalability of a distributed application. A short description of each of the parameters as well as a reason for including them is provided. The excluded parameters are described in Section 3.3.

3.2.1 Computations

An application usually contains a set of different functions. These functions consist of various computations in order to enable the application to provide the correct functionality. The computations should be understood as loops, method invocations, etc. and the amount of these affects the execution time of the function.

When implementing an application in an object–oriented language, two major issues influence the performance. First, the creation and destruction of objects have to be limited as they have a negative impact on performance. The overhead of object creation and destruction depends on the complexity of the objects and how the memory is allocated [9]. Second, the amount of method invocations in the application. Each invocation consumes between 25 and 100 machine instructions for standard programming languages and hardware or even more depending on the arguments and return values of the method [9].

3.2.2 Data Volume

The data volume in real world applications varies greatly depending on the size of the enterprise, from less than a gigabyte (GB) to several terabytes (TB) of data [10]. Therefore, it is important to evaluate different data volumes in the performed experiments to cover different enterprise sizes. As the scope of the project is SME, an estimation of the data volumes used by these enterprises is needed.

In addition, when building an application utilizing a database it is important not only to consider the capacity currently needed for the application, but also to consider future capacity requirements. The data volume needed for storing the business data is likely to change as the enterprise evolves and the amount of orders, customer information etc. grows. When the data volume grows, the ideal placement of functionality might be different from that at design time. Therefore, it is important to take into account how much the data volume is expected to grow and the lifetime of the developed software.

3.2.3 Hardware

One of the reasons for considering where to place functionality is to take best advantage of the available resources. A possibility is to use the hardware resources in parallel instead of sequential. Thereby, the application server is able to query multiple databases and concurrently work on other tasks. However, if the functionality is instead placed on the database, hence requiring the database to process both the query and the functionality, the task will be executed sequentially in

the database. Therefore, the optimal configuration of the hardware in the system depends on both the executed application as well as the available hardware. Furthermore, the processing capability of the system influences the placement of the functionality, as a lightweight database do not have additional CPU cycles that can be used on functionality besides handling queries. Whereas a heavy–duty database might have spare CPU cycles and thus can handle additional functionality. Therefore, it is important to examine how different hardware configurations affect the placement of functionality.

3.2.4 Number of Queries

It is common that in an application with database interactions, some of the functionality retrieve data from the database. In order to retrieve the data, the functionality must execute one or multiple queries. All of these queries introduce a network round trip, which varies in duration according to the execution time of the query, available hardware, etc. Each query takes up a certain amount of CPU time according to their complexity. Hence, the *number of queries* has an influence on the utilization of CPU time on both the application server as well as the database, since more processing has to be done. Therefore, the execution time of a particular functionality depends on the contained number of queries.

3.2.5 Type of Queries

As described in Section 2.1, there are three types of SQL statements; DDL, DCL, and DML, where the most used statements are the DML statements [25], which include `SELECT`, `UPDATE`, `DELETE`, and `INSERT`. As such, the focus is primarily on the DML statements. A consideration to take into account when choosing the queries is the complexity of the queries, such as whether the queries should involve retrieving data from multiple tables through joins or use aggregates to e.g. calculate the sum of a column.

Another consideration is whether the data a query returns is used to take a particular action upon. Relevant actions might include using a subset of the returned data to perform an `UPDATE` or an `INSERT` on another table, while returning another subset of the data to the client. Obviously, being able to perform this on the database is likely to not only affect the performance of the system, but also the amount of network utilization.

3.2.6 Workload Unit

When working with distributed systems, an important parameter to consider is the amount of users that access the system. Therefore, it is important to examine how well the system scales when different amount of users interact with the system. The term *workload unit* (*WU*) is defined as a measure of the load on the system, but we assume a one–to–one relation between one unit and the increased load on the system made by one user. For example, the resources needed to handle 50 users correspond to 50 *WU*.

3.3 Excluded Parameters

Beside the previously mentioned parameters, several other parameters have been considered but not included in the metric for different reasons. These parameters as well as the corresponding level are listed and discussed below.

3.3.1 Installed Software

Installed software refers to the services and software included in Windows Server 2003 as well as any third party software required for a given enterprise application to operate. The environment used for conducting the experiments is cleaned for unnecessary software and services in order to gain the most optimal conditions. In effect, this means that beside vital services, only the experiments are running on the application server, and only MSSQL2005 is running on the database server. The choice was made because simulating all different combinations of running software would be too massive an undertaking and outside the budget of this thesis.

3.3.2 Cache

A cache is a storage mechanism that maintains a collection of recently used data. As new data is retrieved, it is added to the cache and if the cache is full, existing data will be replaced. The benefit of this approach is that by caching the data closer to where it is used, access to the cached copy of the data will be many times faster as opposed to e.g. fetching it from a remote database or computing a set of data for each client request. It should be noted that cache can be found in several different places in a computer; the CPU, disk, operating system, and as we examined in our DAT5 project [1] also at the application level.

However, caches at the CPU and disk are not manageable. As such, mainly the caching happening on the database is of concern. Instead of comparing different caching scenarios on the database however, the focus is on tuning the cache to achieve better performance, which is essential for any distributed application utilizing a database [26].

3.3.3 Batch

A batch is a single, or multiple T-SQL statements, which are grouped into a single unit that can be executed on the database. Take an example where it is required to retrieve data, update a row in a table, and then insert a new record. If these three tasks had to be executed in an ad–hoc SQL manner twice a second, this would result in 518,400 queries a day. If these three tasks were put into a batch, this would result in 172,800 queries a day. In essence, that means by consolidating work through batching makes more effective use of the network since fewer round trips have to be made. However, this is not without drawbacks, since an error in one query will result in the failure of the entire batch.

As indicated, the use of batch provides in some scenarios a performance benefit. However, there are two reasons for excluding batch in the experimental design. First, the Compiere solution, described in Section 1.4.1, has not yet implemented the batch functionality. Hence, the statistic gathered from the solution does not contain any indication of the usage of batch. Second, the use of batch changes the scenarios where e.g. two queries are executed. For example, if the second query requires data from the first query in order to create a new query or insert data into the database.

3.3.4 External Resources

External resources should be understood as accessing external files, the registry, networks, or making remote connections to other application servers or databases and are not considered due to the following reasons. First, the use of external resources inside the database is generally considered inappropriate, as they can be unpredictable or unavailable [24]. Second, accessing external resources are not standard elements in a system, but more specializations for each individual system. Third, when considering remote connections, establishing a connection to a secondary database from either the application server or the primary database implies equal overhead, such as network round trips. Although, in some situations it might be beneficial to place it on the primary database. For example, when the result set returned from the secondary database is used to determine what data

should be returned from the primary database.

3.3.5 Number of Connections

The number of connections is the amount of database connections established in order to execute queries for the users of a given system. The reason for not considering it is that a connection pool is implemented, which manages a fixed amount of available connections. The connection pool is simply a cache of connections that is maintained in the application server's memory. Thereby, when a new connection is established and put into the pool, the connection can be reused when new requests are executed, without having to establish a new, assuming the connection string is equal. If all the connections in the pool are used, then a new one is made and put into the pool. Initially in the life span of an application using connection pooling, there is a performance overhead as new connections are added to the pool. However, assuming a distributed application executing thousands or millions of queries to the database, the overhead of creating and maintaining a connection pool is less than creating new connections for every query. Furthermore, it is possible to initialize a set amount of connections such that new connections do not need to be added continuously. [22]

3.3.6 Returned Result

Returned result should be understood as the result returned by a piece of functionality or query. For example, if the functionality is placed on the database, a scenario could be that data retrieved from the database is traversed; however, only a single value is send back to the application, thus saving network utilization. However, a preliminary analysis of the impact on performance by this parameter indicated an almost insignificant influence, except as explained for network utilization in Section 4.5.

Chapter 4

Experimental Design

To develop the metric, experiments are performed on the factors included in the metric, in order to support the validity of the metric. The chapter contains a presentation of the design strategy for the experiments, as well as a discussion of the factors and levels chosen for the experiments.

4.1 Choosing the Experimental Design

When designing an experimental design, there are primarily two options available. One option is to perform a full factorial design, where all possible factors and levels are included. The other option is to perform a fractional factorial design, where only a subset of the factors and levels are included, thus the design is not validated against all possible combinations of levels as with the full factorial design.

4.1.1 Fractional Factorial Design

Making a fractional factorial design has the benefit of analysing all factors and all levels, but with a limited number of combinations. Assume an experimental design where experiments are performed on all combinations of the factors: *number of queries*, *hardware*, and *computations*. These experiments only include a fraction of the factors: *type of queries*, *data volume*, *WU*, and *locality*. For this experimental design, eight experiments are performed as listed in Table 4.1, however very limited information is gained from these experiments. Even though all combinations are made on the three first factors, nothing is definitely concluded, since the other factors come into play as well. For example, if the experiment with

1 *number of queries*, 1 *computation*, and *hardware Scenario1* is faster than the experiment with 1 *number of queries*, 1 *computations*, and *hardware Scenario2*, then it is not certain that Scenario1 is faster than Scenario2. It could instead be because Query 1 is faster than Query 2 or that a SP written in T-SQL is faster than placing the functionality on the application server. Therefore, with a fractional factorial design, it is possible to examine many factors, but only get an indication of how the factors influence performance and scalability.

Experiment Number	Number of queries	Hardware	Computations	Type of queries	Data volume	Workload unit	Locality
1	1	1	1	Q1	17	63	T-SQL
2	1	1	6	Q1	1	1	SQLCLR
3	1	2	1	Q2	17	125	App
4	1	2	6	Q3	17	63	SQLCLR
5	6	1	1	Q2	1	125	App
6	6	1	6	Q3	17	1	T-SQL
7	6	2	1	Q2	17	1	SQLCLR
8	6	2	6	Q1	1	125	T-SQL

Table 4.1: Example of a Fractional Factorial Design

4.1.2 Full Factorial Design

Making a full factorial design provides information on all combinations of factors and their associated levels. Thus, the drawbacks of performing a fractional factorial design are avoided. Instead, more experiments have to be performed compared to the fractional design, unless the amount of included factors is reduced. The drawback of performing a full factorial design is that it usually becomes extremely comprehensive and time consuming. For example, assume the excluded parameters listed in Section 3.3 should be included as well. For each factor the number of levels is multiplied with each other, e.g. the *number of queries* times *computations* etc. Therefore, by including additional factors the number of experiments increases accordingly. In a more formal logic, it can be described as:

$$n = \prod_{i=1}^{k} n_i$$

$n =$ amount of experiments

$k =$ amount of factors

The ith factor has n_i levels

4.1.3 Experimental Design

Having considered the benefits and drawbacks, we have chosen the full factorial design with fewer factors and fewer levels. The experimental design with thorough experiments on a limited set of factors and levels gives more usable results than a fractional design containing all factors with a limited amount of combinations. This is because in order to develop the metric, it is vital to know how much each of the factors affects the measurements. If this information is unknown, the metric will be nothing more than a guess.

For the full factorial design, the following seven factors with the number of levels in parenthesis are chosen: *number of queries*(2), *hardware*(2), *computations*(2), *type of queries*(3), *data volume*(2), *WU*(3), and *locality*(3). Performing a full factorial design on these factors accumulates to 2*2*2*3*2*3*3 = 432 experiments for the experimental design. The experiments are throughout the thesis referred to as the abbreviations c0 to c431. One example of an experiment is c0 with 17 GB *data volume*, *hardware Scenario1*, *Query* 1, 1 *number of queries*, 1 *computation*, 1 *WU*, and *locality* on the application server.

4.2 Preliminary Experiments

Before running the full suite of experiments, it is preferable to perform a subset of the experiments. The purpose of these experiments is to determine whether some factors or levels should be removed, and determine how many replications that have to be performed in order to get accurate results. Furthermore, the preliminary experiments are used to uncover bugs and errors that might occur and potentially render some results unusable. Thus, by fixing these bugs and errors, the overall comparison is more accurate for all experiment configurations.

Since a single execution of the preliminary experiments would not be very useful, considering there might be one or more outliers among the results, three repli-

cations are performed. The results from the preliminary experiments are subsequently used to determine the final factors and levels, as well as the amount of replications for the final experiments. How the number of replications is calculated is described in Section 4.4.

The implementation and setup of the preliminary experiments are not described; instead refer to the description of the final implementation in Chapter 5, since the difference between the preliminary experiments and the final does not differ much. The preliminary experiments were tested on the *returned result* parameter and all factors and levels described in this chapter, except for *hardware* and *data volume*, since it was not possible at the time due to software issues.

The preliminary experiments indicated that all factors except one have an impact on the overall performance. Thus, all factors are included except the *returned result* parameter as explained in Section 3.3.6.

4.3 Factors

In our approach, seven different factors are considered, which are specified with specific levels for each experiment. Several other parameters have been evaluated, which were described in Chapter 3. The chosen factors are described below with their corresponding levels.

4.3.1 Computations

2 levels: 1 and 6 computations.

Number of Computations

The number of computations an application is required to perform can differ from one execution to the other. In order to specify the levels for computations, the observed numbers in the Compiere solution are used, as listed in Table C.2 in Appendix C. The number of computations concerning loops in the Compiere solution range from zero to six. However, as the queries only returns 10 rows, the difference between 0 and 1 loop is limited. The loop can be flattened to an almost identical computation without any looping. Additionally, functions containing 1 loop is the most frequently observed amount in the Compiere solution. Therefore, the levels for computations are specified to 1 and 6.

Complexity of Computations

The performance issues concerning creation and destruction of objects are disregarded, as the statistics of the functions in the Compiere solution in Appendix C.3 indicate the functions only contain a limited amount. However, the number of method invocations associated with each query is specified according to the average values identified by the Compiere solution. Therefore, each of the functions consists of 16 method invocations outside the loop(s) and each loop contains 7 additional method invocations.

The reason for distinguishing between method invocations inside and outside loops is the number of times a method is invoked. Furthermore, loops are generally considered the cornerstone of algorithms concerning execution paths [6]. However, the loops in the functions are not nested as only 15 of the functions executing `SELECT` statements have nested loops in the Compiere solution. The reason for counting methods in the specification is the CPU utilization of invoking a method is significantly larger than e.g. evaluating a logical boolean expression. Hence, the complexity of computations is limited to method invocations and loops.

4.3.2 Data Volume

2 levels: 1 and 17 GB data.

As the data volume differs among SME, the experiments on performance and scalability require different volumes of data. Since the benchmarks from TPC are business related, their data makes sense to utilize in the experiments. The database schema used for outlining our metric is therefore the same as the one in the TPC-H benchmark, which is illustrated in Appendix D. The database schema consists of eighth tables, ranging from the small `REGION` table to the large `LINEITEM` table that contains 102 million rows for 17 GB data volume. The numbers above the table names are the amount of rows in the table factored by a scale factor (SF). The SF is used to simulate various volumes of data and is achieved by scaling the data up to a given amount. The SF used is 1 and 17 GB of data, which is inspired by the *"Exact Globe 2003"* ERP solution [27]. The specification for this solution states that small enterprises use databases with less than 2 GB of data, small/medium enterprises use databases with data between two and 8 GB of data, medium/large enterprises use databases with data between 8 and 25 GB of data, and large enterprises use databases with more than 25 GB of data [27]. Considering the metric development is addressed at SME, specifying the data volume to 1 and 17 GB is representative.

4.3.3 Hardware

2 levels: Scenario1 and Scenario2.

To examine different hardware scenarios, the computer laboratory at Aalborg University is utilized. The laboratory contains a number of machines, however only one machine has significant computation capabilities. In addition to this machine, the machine situated in our group room is used, albeit not as good as the one in the laboratory. The configurations are listed in Appendix A.

Since the configurations of the machines available differ, it is not possible to examine a system configuration where both application server and database have equal memory and processing capacity. As only one machine has significant computation capabilities, at least one of the machines in the system configuration is lightweight. Hence, the available machines provide three possible system configurations. However, as the lightweight machines are limited in processing capacity and are insufficient for SME, the configuration containing two lightweight machines is left out. The two configurations used as levels for the *hardware* factor are therefore:

	Database	**Application**
Scenario1	heavy–duty	lightweight
Scenario2	lightweight	heavy–duty

Table 4.2: Hardware Configurations

4.3.4 Number of Queries

2 levels: 1 and 6 queries

The experimental design specifies two levels for the *number of queries* executed in a function, which are either 1 or 6 queries. The two levels are chosen after examining the functionality containing database interactions in the Compiere solution. The statistics in Table C.2 in Appendix C illustrate that a function with only 1 query is the most common in the Compiere solution. Furthermore, the Table illustrates the highest number of queries executed by any of the functions is 6. Hence, *number of queries* is set to 1 and 6 according to the numbers observed in the Compiere solution.

4.3.5 Type of Queries

3 levels: Query 1, Query 2, and Query 3

To make the metric cover a number of scenarios, it is important to examine different queries that might be relevant in a real distributed system. Since data is used from the TPC-H benchmark, it makes sense to partly utilize the queries from this benchmark, which is designed specifically for the database schema. Thereby, the queries examine a large percentage of the available data, while exhibiting a high degree of complexity.

The 22 queries covered in the TPC-H benchmark are all `SELECT` queries, which is the most commonly used query in real systems. One example of such a real system is the Compiere solution that contains 639 functions containing `SELECT` queries and 173 functions containing `INSERT`, `UPDATE`, or `DELETE` queries. Thus, to comply with these statistics, the experimental design covers two different `SELECT` queries and a `SELECT` query, where the result is used to insert data into the database and subsequently delete it again.

Query 1 resembles query Q6 from the TPC-H benchmark and Query 2 resembles Q14, which is illustrated in Listing B.1 and B.2 respectively in Appendix B. Query 3 also uses query Q14 as in Query 2, however additionally it inserts the data retrieved in the query into a table and subsequently deletes it. A more detailed description of Q6 and Q14 is provided in Section 5.1.3.

4.3.6 Workload Unit

3 levels: 1, 63, and 125 *WU*

The *WU* factor is meant as a representative unit for the users that access the system. Since the metric is directed at SME, the highest level is set to 125. The number is derived from the specification for SME by the European Commission. Small enterprises are specified as having less than 50 employees and medium as having less than 250 [28]. Choosing 125 *WU* is therefore approximately the mid value. Furthermore, the worst–case scenario is assumed, which means that all users use the system concurrently. Having assumed the worst case scenario and in order to examine how well the system scales, the levels 1 and 63 *WU* are also considered.

Although 125 seem reasonable for a SME, it is still a relatively small amount compared to larger scale applications, where the amount of users can be tens of thousands. However, testing for this large amount of users is not viable because of the limited amount of resources available as mentioned earlier, as such we limit

this factor to our target group of SME.

4.3.7 Locality

3 levels: Application Server, SQLCLR, and T-SQL

The *locality* factor has three levels; T-SQL, SQLCLR, and functionality on the application server. Although the purpose of the metric is to determine whether a certain piece of functionality should be placed on either the database or application server, it is still important to know which implementation technique is best on the database – T-SQL or SQLCLR. This is because T-SQL and SQLCLR performs best in different areas, as mentioned in Section 1.3. As such, T-SQL might perform better than the application server, whereas SQLCLR does not.

4.4 Replication

Performing replications is an important part of any experiment in order to reduce deviations and errors. It is of course not possible to remove all experimental errors, but it is possible to minimize the effect they might imply, thereby improving the accuracy of the experiments. Based on the results from the preliminary experiments, which were obtained by running the experiments three times, the final experiments should be replicated 78 times. This is a compromise between the required time for executing all the experiments and a satisfactory accuracy. 78 replications give an inaccuracy of maximum 20% with a certainty of 80% (80/20). In contrast to this 514 replications had to be made if the accuracy should be 90/10, and 2917 replications if the accuracy should be 95/5. The calculations used to define the amount of replications are based on the formula used in [7].

Replication formula:

$$n = (\frac{100zs}{r\overline{x}})^2$$

$n =$ Amount of replications required.

$z =$ The normal variate of the desired confidence level. Since a 80% quantile is used, this value is 1,282 according to Table M.1 in Appendix M.

$s =$ Standard deviation in response time.

$r =$ Accuracy of response time. Setting the level to 20, means the mean response time of the result will deviate with maximum 20%.

$\overline{x} =$ Mean of the response time.

Putting the values for c115 from the preliminary experiments into the formula gives the following calculation:

$$n = (\frac{100 \times 1{,}282 \times 1{,}67174583}{20 \times 1{,}2134729})^2 = 77,98237$$

The above calculation has been done for all experiments and the highest replication value was that for c115 with 77,98, thus 78 replications are performed. All the results from calculating the amount of replications can be found in Appendix J.

4.5 Experiment Measures

The different functionality available in the system have different possible outcome; correct, incorrect, or refused. In order to evaluate the performance of these outcomes the following measures are suggested: response time, throughput, reliability, availability, and utilization [29]. However, as we are only concerned about the correct outcomes of the performed experiments, the availability and reliability measures are disregarded. The reason is that correct execution of functionality is of primary concern of an implementation, whereas incorrect and refused represents exceptions. Therefore, during the execution of the functionality, the following measures are recorded for further analysis.

Response Time

The analysed functionality is provided by the application server, even though the underlying functionality is situated on either the application server or the database. Hence, the response time is the time between the invocations of the function on the application server until the result of the function is available for use in the application. The measurement is performed individually for each of the functions representing the experiments.

Throughput

Throughput measures the rate of function executions per unit of time the system is able to service and is measured to determine how well the system scales as load on the application server and database are varied according to the *WU* factor. This effectively means:

$$Throughput = \frac{workload\ units}{response\ time}$$

CPU Utilization

There are three CPUs present in the system. As described in the hardware configuration in Appendix A, one of the machines has 1 CPU and the other 2 CPUs. Both the CPU on the application server as well as the CPU on the database is measured concerning utilization. The measured CPU utilizations are an average of the percentage of CPU time used during the execution of a function in the application server and database respectively.

Network Utilization

The various functions transfer different amount of data across the network as the underlying functionality is placed on either the application server or the database. Hence, the total amount of transferred data is measured for each execution of the functions.

Part III

System Development

Chapter 5

Experiment Implementation

In this chapter, the experiments specified in the experimental design are implemented, this is done in order to be able to produce sample data that can be used to define the metric. An application is implemented that include functionality corresponding to each of the 216 scenarios specified in the experimental design. The reason only 216 scenarios are implemented is because the *hardware* factor cannot be implemented directly into an application, but only varied externally by switching between the two hardware configurations respectively. This chapter contains a description of the setup used for the different experiments, as well as a detailed description of the implementation of the application and the various performance optimizations applied to both the setup and application.

5.1 Implementation

The application is implemented according to the factors and their corresponding levels as specified in Chapter 4. One of the factors is *locality*, where the application server in this factor is used as a common base for the implementation, i.e. the functionality is first implemented for the application server. The functionality provided by this implementation is then translated to SQLCLR and T-SQL respectively. This approach ensures the functionality provided by the different implementations is equivalent. The reason for using the application server as the common base is the statistics from Compiere solution used to outline the *computations* factor are gathered from the application server tier. Furthermore, one of the ideas of integrating the .NET framework on the database is to enable the programmer to move code from the application server to the database [3].

As the Compiere solution is implemented in Java, the values obtained from the

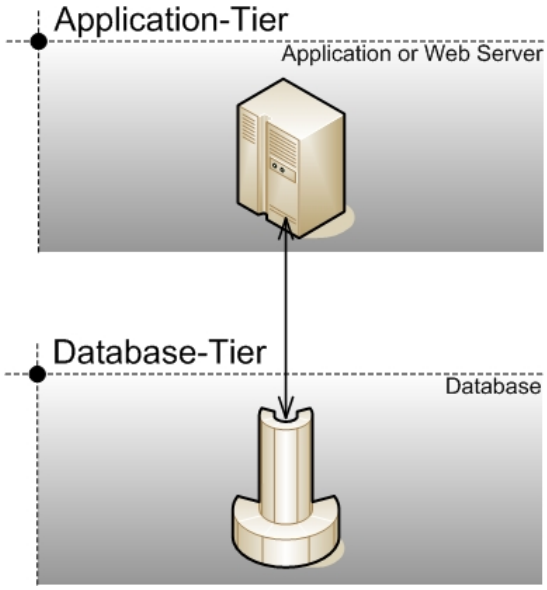


Figure 5.1: System Architecture

statistics as presented in Appendix C need to be converted in order to be valid in the C# implementation. Therefore, statements accessing attributes using implicit `get` and `set` methods are counted as explicit method invocations. Furthermore, the `using` keyword in C# is able to implicitly call the `dispose` method. Hence, the different functionalities implemented for the application server and SQLCLR seem to contain fewer method invocations than defined in the *computations* factor; however, this is not the case. The following sections contain the necessary steps involved in developing the application and to comply with the stated factors in Section 4.3.

5.1.1 System Architecture

First the system architecture has to be determined. The system is, as already mentioned, structured as a three–tier system, however physically it is only a two–tier system as illustrated in Figure 5.1. The reason this approach is chosen is from a financial viewpoint, since choosing a true three–tier approach would require 125 different clients according to the *WU* factor. Instead, the application server is structured to simulate 1, 63, and 125 different clients accessing the application by using threads.

It can be argued whether this approach is optimal, since using threads to simulate clients is not pure concurrency, considering only one `Thread` can truly run

at a time, unless the simulation is performed on a machine with multiple processors. However, context switching between different threads is so fast that it is hardly noticeable. Another argumentation against using threads is the fact that it introduce an overhead each time threads have to be context switched [30]. The overhead is because each time a `Thread` has used its time slice on the CPU, the state of the `Thread` has to be saved; thereby the thread scheduler can go back to the exact state of that `Thread` when it is granted a time slice again. Using true three–tier system architecture would also introduce an overhead, since the application server would have to handle the incoming connections. We do however, acknowledge the most realistic approach would be to structure the system as a true three–tier system architecture.

5.1.2 Setup

The setup is specified according to the *hardware* and *data volume* factors. The hardware scenarios used are as specified in Section 4.3.3, with the configurations listed in Appendix A. In order to generate different data volumes, a tool named DBGEN developed by TPC is utilized [31]. DBGEN is a database population generator that generates random data corresponding to the schema for the TPC-H benchmark, which is also the same schema used for the database as listed in Appendix D. Since the data is randomly generated, the same data generated for 1GB and 17GB are used in both hardware scenarios to ensure a fair comparison.

Optimizations

The setup is a combination of several components, such as hardware, operating system, database, and the implemented experiments. Each of these components is optimized according to existing guidelines. The guidelines are primarily from two sources, namely a series of articles named *"How to Perform a SQL Server Audit"* [32] and a document named *"Improving .NET Application Performance and Scalability"* [26]. One of the guideline mentioned in [32] states that most SQL Server configuration settings should not be changed, since doing so will likely result in decreased performance. For example, in our case where the database runs dedicated on a server, the maximum and minimum memory allocation settings is left at default, since it allows SQL Server to dynamically allocate memory for best performance.

5.1.3 Query Selection

The three selected queries for the experiments are modifications of the TPC-H queries Q6 and Q14 and comply with the *type of queries* factor in the experimental design. Furthermore, they are also designed to be comparable with the *computations* factor, concerning the amount of loops and method invocations.

TPC-H Queries

The TPC-H benchmark includes 22 decision support queries, which cover a broad spectrum of useful real world business scenarios. Among the 22 queries stated in the TPC-H benchmark, only the following two queries are used in the developed experiments:

- **Forecasting Revenue Change Query (Q6)**
  This query forecasts how much could be saved over a given year if certain discounts +/- 0.01 are removed. This is done by running through the `LINEITEM` table and adding the discounts with a shipping date in the current year, the correct discount, and a quantity less than the stated quantity.
- **Promotion Effect Query (Q14)**
  This query is used to monitor how various promotions such as campaigns or advertisement is responded to on the market. This is done by performing a join on the `LINEITEM` and `PART` tables and retrieving only those parts that are shipped in a given year and month and returns the percentage of the revenue for them.

Our Queries

The Q6 and Q14 queries are used as basis and modified according to fulfil the *computations* factor presented in Section 4.3.1. That is, if the *computations* factor needs 4 columns and only 2 is provided by the original TPC-H query, additional columns are added in order to make the computations comparable. Furthermore, in order to make the queries comparable both are modified to return at most 10 rows.

Query 1, illustrated in Listing B.1, is almost identical to the TPC-H query Q14. The difference is that in order to fulfil the *computations* factor, our query additionally selects the columns `L_SHIPMODE`, `L_PARTKEY`, and `L_SUPPKEY`.

Query 2 is illustrated in Listing B.2 and is somewhat similar to the TPC-H query Q6. The difference between the two queries is that our query selects at most

10 rows and additionally retrieves data from the four columns L_SHIPMODE, L_PARTKEY, L_SUPPKEY, and L_TAX instead of getting the REVENUE aggregate. The reason for this substitution is to get a simpler query that does not utilize many resources on the database.

Query 3 is illustrated in Listing B.3. Query 3 is actually a combination of multiple statements and uses the same SELECT statement as in Query 2. In addition, it inserts the four columns of data retrieved in Query 2 into a separate table and subsequently deletes it.

The reason for choosing these three queries is that they are rather simple and have a reasonable execution time making them suitable for running a large number of replications. Furthermore, the TPC-H queries are representative for any industry that manage, sell, or distribute products. The forecasting (Q6) and promotion effect (Q14) are common tasks in these businesses [31].

Query Optimization

The three queries are optimized using the MSSQL2005 Database Engine Tuning Advisor (DETA) tool. By using the queries as input to the DETA, several optimizations are suggested in order to achieve better performance. For the three queries, the DETA suggests 2 indexes and 2 statistic to be created as presented in Appendix F.1. The statistics are associated with the LINEITEM table, whereas an index is associated with both the PART and the LINEITEM tables respectively. The expected performance gain from the suggested optimizations is estimated to be approximately 99%. All of the optimizations suggested by DETA are performed. The reason for not using optimizations such as table partitioning is that partitioning is not supported in the standard version of SQL Server, which is used in the system.

- **Statistics**
  Statistics are used by the query optimizer to help create the most optimal query execution plan for a query and whether indexes should be used for the query. The statistics contain information such as cardinality and selectivity [16].

- **Partitioning**
  Partitioning should be understood as the process of splitting either a database or table into several smaller databases or tables. The partitioning can be done following different approaches, such as partitioning on a range of keys for a table or for a list of values. The benefit of partitioning can be in-

creased performance, scalability, and manageability, since queries do not need to search and retrieve data from e.g. a large table [16].

5.1.4 Order of Execution

A consideration to take into account when implementing the experiments is the order in which they should be executed. This should not be neglected, since mechanisms such as caching on the database can affect the outcome. Instead of executing each experiment 78 times in a row, to comply with the replications, and then proceed to the next experiment, a random method has been implemented that generates a random sequence of numbers from 0–107 corresponding to the number of experiments. This method is executed 78 times in order to generate the random execution order for the 78 replications. The execution order of the experiments in the first replication is presented in Appendix E.1. The sequence of numbers is therefore not random across the 78 replications, i.e. any number can at most appear two times in a row, which happens when a number is the last in a replication and the first in the succeeding replication.

The reason for generating the numbers this way is the random pattern of random functions. Preliminary testing of the .NET random function revealed a non–desirable effect. The last numbers in a sequence usually appeared frequently because those numbers were the only ones left. Hence, in order to reduce the non–random pattern, the method described previously is used.

5.1.5 System Initialization

In order to provide a common base before the execution of the experiments, an initialization step is developed. The initialization procedure contains two steps that ensure the preconditions are the same.

First, the database is restarted in order to reset any caching that might be present. Furthermore, any non–vital processes are stopped, thereby the database has the maximum amount of memory available and the effect by other processes that might require processing during execution is minimized.

Second, the 48 methods calling the SP on the database are initialized; thereby the overhead of loading the data to main memory, establishing the connection pool as well as compiling the SP and queries is avoided. Hence, the variance in the measurements is smaller and fewer replications are required.

5.1.6 Performance Monitoring

In order to determine how well each of the experiments performs, it is necessary to continuously collect information concerning different performance aspects of the experiments, which are related to the measures specified in Section 4.5. According to [32], some of the key aspects to monitor are disk, memory and processor utilization. To collect information on these aspects, the .NET Framework `PerformanceCounter` class is utilized, which can be used to read a number of Windows NT performance counters. The performance counters implemented in the application are:

- *"Available KBytes"*
  Measures the amount of free memory
- *"% Processor Time"*
  Measures the utilization of one or more CPUs in percentage – can exceed 100% for multiple CPUs
- *"Avg. Disk Queue Length"*
  Measures how long the I/O queue is for the disk
- *"Bytes Total/sec"*
  Measures both data transmitted and received over the network

Listing 5.1: Sample Performance Counter

```
1 ...
2 PerformanceCounter CPUdb = new System.Diagnostics.
     PerformanceCounter("Process", "% Processor Time", "
     sqlservr", "D631A-CELSIUS");
3 ...
4 dbCPU += CPUdb.NextValue();
5 ...
```

A sample performance counter is illustrated in Listing 5.1. In this case, the counter is specified for collecting information about how much processor time the `sqlservr` process, which is the process handling the SQL Server, consumes on the remote database situated on the Celsius machine. To retrieve the information, the `NextValue()` is called as illustrated in line 4. Furthermore, in order to reduce the overhead that performance counters introduce, this information is only collected once every 1/10th of a second.

In addition to the information collected from the performance counter, another aspect has to be considered, namely the time it takes for executing each of the

experiments. The response time is calculated using a special high–resolution counter called `QueryPerformanceCounter`, which supports a timing resolution of approximately 10 milliseconds. This is in contrast to the standard `DateTime.Now()` timer method that only supports a resolution of 1 second [33].

5.2 Code Snippets

The section presents code snippets for each of the three localities; application server, SQLCLR, and T-SQL. The code snippets illustrate the implementation of the same piece of functionality for each of the three. The functionality has been picked because it gives a good overview of the general implementation. The code functionality is specified for *Query 1*, 1 *computations*, and 1 *number of queries*. In order to reduce cluttering the code with the query used in the implementation, the query has been left out and replaced by a `QUERY1`, however refer to Query 1 in Appendix B for an outline of the query.

5.2.1 Application Server Code

To simulate that all code is executed on the application server, no code is placed on the database, as neither SP nor general functionality. Instead, the functionality is restricted solely to the application server, using ad–hoc string based SQL statements to retrieve data from the database, which is illustrated in Listing 5.2.

Listing 5.2: Application Server Code

```
1  public void Query1Number1Comp1()
2  {
3      int i = 0;
4      StringBuilder sb = new StringBuilder();
5      using (SqlDataReader sdr = db.ExecuteReader(CommandType.
           Text, QUERY1))
6      {
7          while (sdr.Read())
8          {
9              sb.Append(sdr.GetString(0)).Append(sdr.GetInt32
                   (2)).Append(sdr.GetDecimal(3));
10             i += sdr.GetInt32(2);
11         }
12     }
13     sb.Append("1"); sb.Append("2"); sb.Append("3"); sb.Append
           ("4"); sb.Append("5");
14     sb.Replace('1', '2');
15 }
```

To simulate one computation, a `StringBuilder` is implemented, which the data retrieved from the database is appended to together with a number of static appends, in order to reach the set number of method calls as specified in Section 4.3.1. Another point to consider regarding computations is loops, which is simulated by a `while` loop that iterates through the data retrieved from the database. The connectivity to the database is handled through an external connectivity class, which manages all communication with the database, such as opening and closing the connections to the database.

5.2.2 SQLCLR Code

As opposed to placing all code on the application server, only the connectivity between the application server and database is handled on the application server. The actual code containing the functionality is solely placed on the database, however compared to the code described previously for the application server, there is not much difference as illustrated in Listing 5.3.

Listing 5.3: SQLCLR Code

```
1  [Microsoft.SqlServer.Server.SqlProcedure]
2  public static void ClrQuery1Number1Comp1()
3  {
4      using (SqlConnection conn = new SqlConnection("context
           connection=true;"))
5      using (SqlCommand cmd = conn.CreateCommand())
6      {
7          int i = 0;
8          StringBuilder sb = new StringBuilder();
9          cmd.CommandText = QUERY1;
10         cmd.CommandTimeout = 360;
11         conn.Open();
12
13         using (SqlDataReader sdr = cmd.ExecuteReader())
14         {
15             while (sdr.Read())
16             {
17                 sb.Append(sdr.GetString(0)).Append(sdr.
                       GetInt32(2)).Append(sdr.GetDecimal(3));
18                 i += sdr.GetInt32(1);
19             }
20         }
21         sb.Append("1"); sb.Append("2"); sb.Append("3"); sb.
               Append("4"); sb.Append("5");
22         sb.Replace('1', '2');
23         SqlMetaData[] metadataQuery1 = new SqlMetaData[] {
24                new SqlMetaData("s_name", SqlDbType.Text)};
25         SqlDataRecord record = new SqlDataRecord(
               metadataQuery1);
```

```
26            record.SetSqlString(0, sb.ToString());
27            SqlContext.Pipe.Send(record);
28        }
29    }
```

In line 1, the class is specified as a SP, however lines 2–22 are almost equal to placing functionality on the application server, except the connectivity internal on the database is handled explicitly in the SP. It is however noteworthy to recall from Section 2.2.4 that in order to use the client's existing connection, thus not having to create a new connection, the `SqlConnection` has to be specified as "`context connection=true`" as illustrated in line 4. In lines 23–27, is the code necessary to send results back to the application server, where the `SqlDataRecord` class represents a single row of data and its corresponding meta data, which is specified by the `SqlMetaData` object in line 23. The `SqlDataRecord` is then sent back to the application server using the `SqlPipe` as specified in line 27.

5.2.3 T-SQL Code

As with SQLCLR, only the connectivity between the application server and database is handled on the application server. The T-SQL code is illustrated in Listing 5.4. Line 1 in Listing 5.4 contains the standard keywords used to create a T-SQL SP on the database, as well as an `OUTPUT` variable declaration, which specify it should be returned upon successful execution. The statement in line 4, is an optimization as it stops the message containing the number of rows affected to be returned as part of the result. Hence, reducing the amount of data returned to the application server. In lines 5–6, are various declarations of required variables, however only one is of particular interest. In line 6, an in–memory temporary table is created, which is populated in line 8 with the values returned by Query 1. Instead of the in–memory table, a `CURSOR` could have been used. However, `CURSOR` should generally be avoided as they are slow and use more resources compared to e.g. in–memory table [32]. Hence, the usage of an in–memory table is an optimization of the implemented T-SQL code. The data obtained by Query 1 is extracted from the temporary table inside the `WHILE` loop in lines 10–14. Each row in the temporary table is extracted and either appended to the `@sb` variable or added to an integer variable as seen in line 13. At the end of the SP all the 1's in `@sb` are replaced with 2's.

Listing 5.4: T-SQL Code

```
1 CREATE PROCEDURE [dbo].[TsqlQuery1Number1Comp1] (@sb VARCHAR
      (8000) OUTPUT)
2 AS
```

```
3  BEGIN
4    SET NOCOUNT ON;
5    DECLARE @i INT, @count INT, @countlimit INT, @names VARCHAR
         (25), @namen VARCHAR(25), @partkeyp INT, @addresss
         VARCHAR(40)
6    DECLARE @tempTable TABLE(ROW_VALUE INT IDENTITY (1,1),
         S_NAME VARCHAR(25), N_NAME VARCHAR(25), P_PARTKEY INT
         NOT NULL, S_ADDRESS VARCHAR(40))
7    SELECT @i = 0, @count = 1, @sb = ''
8    INSERT INTO @tempTable QUERY1
9    SELECT @countlimit= COUNT(row_value) FROM @temptable
10   WHILE @count <= @countlimit
11     BEGIN
12       SELECT @names = s_name, @namen = n_name, @partkeyp =
             p_partkey, @addresss = s_address FROM @tempTable
             WHERE @count = row_value
13       SELECT @sb = @sb + @names + @namen + @addresss, @i = @i
              + @partkeyp, @count = @count + 1
14     END
15   SELECT @sb = @sb + '1' + '2' + '3' + '4' + '5'
16   SET @sb = REPLACE(@sb,'1','2')
17 END
```

Chapter 6

Experiment Results

This chapter contains a presentation of the results of the experiments. As described previously, the experimental design includes 432 different experiments, which have been replicated 78 times to a total of 33.696 experiments. To help interpret and present the large amount of results collected from the experiments, a number of statistical methods have been employed. These methods as well as a discussion of the accuracy of the experiments are presented in this chapter. Furthermore, the summarized data collected from using the statistical methods are used to graphically illustrate and discuss interesting findings.

6.1 Statistical Methods

When working with large sets of data that should be analysed, it is favourable to use statistical methods to summarize the data and present it as graphs or charts. The methods used are the mean value and the confidence interval, which are described below.

6.1.1 Mean Value

The reasons for using the mean value of the replications are to reduce the sum of squared deviation as much as possible and because the skewness of the sample data is limited. If the determinants in Figure 6.1 are not fulfilled, it is recommendable to use mean instead of e.g. mode or median [7].

- **Mode**
  The most frequent element in a data set.

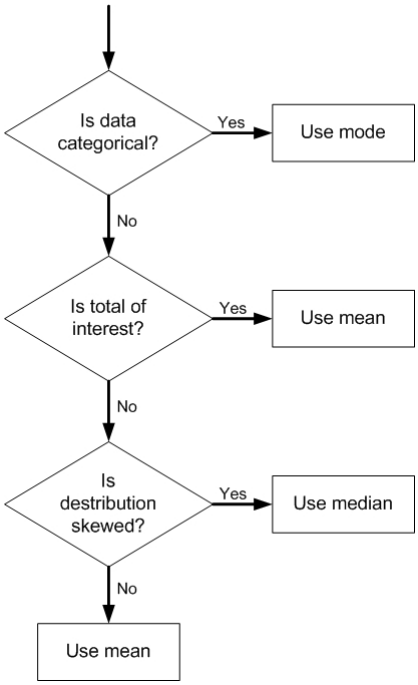


Figure 6.1: Deciding between the Median, Mode, and Mean

- **Median**
  The element located exactly between two exact halves of a sorted data set.

In order to compare the different alternatives, the mean value of each of the measurements is calculated by taking the sum of all the replications divided by the number of replications:

$$Mean\ \overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

n = Number of replications
x_i = Measured value of the i'th replication

6.1.2 Standard Deviation

The mean value is an average value; however, it disregards the variance in the replications. The variance should be understood as a measurement of the dispersion of a replication, thereby indicating how far from the mean value the typical

measured value is. In order to calculate confidence intervals of the mean value, standard deviation must be determined. The variance is calculated by:

$$Variance \;\; s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$$

n = Number of replications
x_i = Measured value of the i'th replication
$\overline{x}$ = Mean value

$$Standard \;\; Deviation \;\; s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2}$$

6.1.3 Confidence Interval

The calculated mean value is as explained an average of the 78 replications. Hence, the obtained values for each of the 78 replications are likely different from the calculated mean value and even varies between samples. As the values differ from the mean, it is not a perfect estimate. In fact, it is not possible to obtain a perfect estimate from any finite size samples [7]. Therefore, it is required to calculate an interval, which with a certain probability contains the mean of the sample. The probability of the mean being within a certain interval is given by:

$$Probability\{c_1 \leq \overline{x} \leq c_2\} = 1 - \alpha$$

$\overline{x}$ = Mean value
α = Significance level
$1 - \alpha$ = Confidence coefficient

The value of $1 - \alpha$ can be converted to percent. Hence, the value of $100(1 - \alpha)$ is called the confidence level and is usually near 100%. In our case, α is specified to 0,2, which means with 80% confidence the mean is between interval. The confidence interval within the range of c_1 and c_2 is calculated by

$$Confidence \;\; Interval(\overline{x} - \frac{z_{1-\alpha/2^S}}{\sqrt{n}}, \; \overline{x} + \frac{z_{1-\alpha/2^S}}{\sqrt{n}})$$

n = Number of replications
$\overline{x}$ = Mean value
$z_{1-\alpha/2^S} = (1 - \alpha/2) - quantile$
S = Standard Deviation

In order to calculate the bounds of the confidence interval a value for the quantile needs to be obtained. The 100-quantiles are referred to as percentiles. The quantile value is obtainable from tables such as the one in Appendix M.

When the confidence intervals are calculated for the different experiments it is possible to determine, which experiment is the better one and thereby decide whether the functionality should be placed on the application server or the database. If the calculated confidence intervals do not overlap, one of the compared experiments can be said to be better than the other. However, if the confidence intervals of two experiments overlap it is not possible to determine which of the experiments is the best without performing additional replications to reduce the uncertainty.

6.2 Replication Accuracy

Before examining the data gathered from the experiments, it is advisable to analyse the accuracy of the data, since data not within the expected bounds can lead to misleading conclusions. As specified in Section 4.4, the satisfactory amount of replications was calculated to be 78. This amount of replications should ensure with a certainty of 80% that the measured mean value of the response time does not deviate with more than 20% from the real mean value. However, whether this is the case cannot be determined exactly, since the real mean value is unknown. Thus, it can only be assumed the measured mean values for 78 replications are correct with the stated accuracy.

Although the real mean value is unknown, it is still possible to determine whether the results follow the tendency from the replication calculations in Section 4.4. That is, whether the results that deviate the most from the mean are the same experiments that required the most replications. By doing this, it is possible to detect unexpected errors since experiments with low replication levels should be the ones with lowest deviation, according to the formula for replications in Section 4.4.

The replication calculations are listed in Appendix J and the deviations in the result are listed in Appendix I. When examining the values in these two tables, some interesting experiments are c115, c150, and c83. c115 was the one that required the most replications with 77,98, but the result is much more accurate and only two experiments deviate with more than 10% from the mean and none deviate with more than 20%. c150 also required a lot of replications with 35,75, which is also the tendency for the result where 73 of the replications deviate with more than 50%. When examining the other spectrum of deviations, c83 only

required 0,0004 replications and the result also shows that none deviates with more than 10%. Generally speaking, it seems the tendency from the replication calculations is not far off for the results, even though a few experiments deviates with more or less than expected.

6.3 Sample Data

The data collected during the execution of the experiments is as described in Section 4.5 divided into four groups: response time, CPU utilization, network utilization, and throughput. The data for the four groups are obtained by the various performance counters as described in Section 5.1.6 and are collected for each of the 33.696 experiments. Presenting all the data from these experiments would be too vast; instead, in the following a description of the data that displays the most interesting deviations is given. The displayed data is the mean value for each of the 78 replications, calculated using the equation in Section 6.1.1. Furthermore, the confidence interval is illustrated for each of the values by using the confidence interval equation in Section 6.1.3. The interval is represented by a high and low value and the interval between is the confidence interval.

The data represented use the following static configuration:

- 6 *Number of queries*
- 6 *Computations*
- *Hardware Scenario1*
- 1GB *Data volume*

The factors *WU*, *locality*, and *type of queries* are varied throughout the collected data.

6.3.1 Response Time

The response time provides an overall measure of how fast each of the experiments executed and is the prime measure used to determine how well each of the experiments performs.

1 Workload Unit

Figure 6.2 illustrates the response time for 1 *WU*. The tendency is clear; the application server has the highest response time for all three queries and the SQLCLR has the lowest. The results are not surprising, since simulating only 1 *WU* puts limited load on the database and as such SQLCLR and T-SQL can utilize the database more effectively.

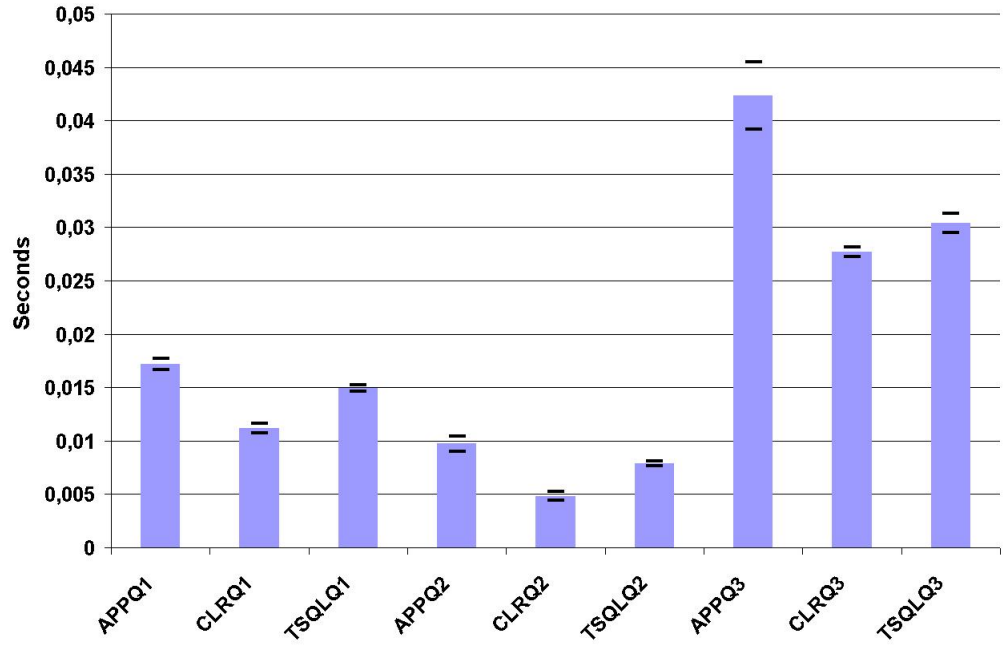


Figure 6.2: Response Time - 1 Workload Unit

An interesting point to consider is the large confidence interval illustrated by the two black lines for Query 3. However, this is likely because the execution time of the experiments for 1 *WU* is as low as approximately 5ms. The slightest variance in the setup, such as another process briefly utilizing the CPU, brief network congestion, or the performance counter measuring the elapsed time, affects the results much more than if the execution time is longer.

63 Workload Units

Figure 6.3 illustrates the response time for 63 *WU*. As opposed to the tendency for 1 *WU*, the figure indicates the application server is noticeable faster than both SQLCLR and T-SQL for Query 1, marginally faster than SQLCLR for Query 2,

but not the fastest for Query 3. The reason the application server is faster for Query 1 and 2 is because the load on the database is increased as the number of *WU* is increased, which can also be read from the graphs illustrating the CPU utilization in Section 6.3.3.

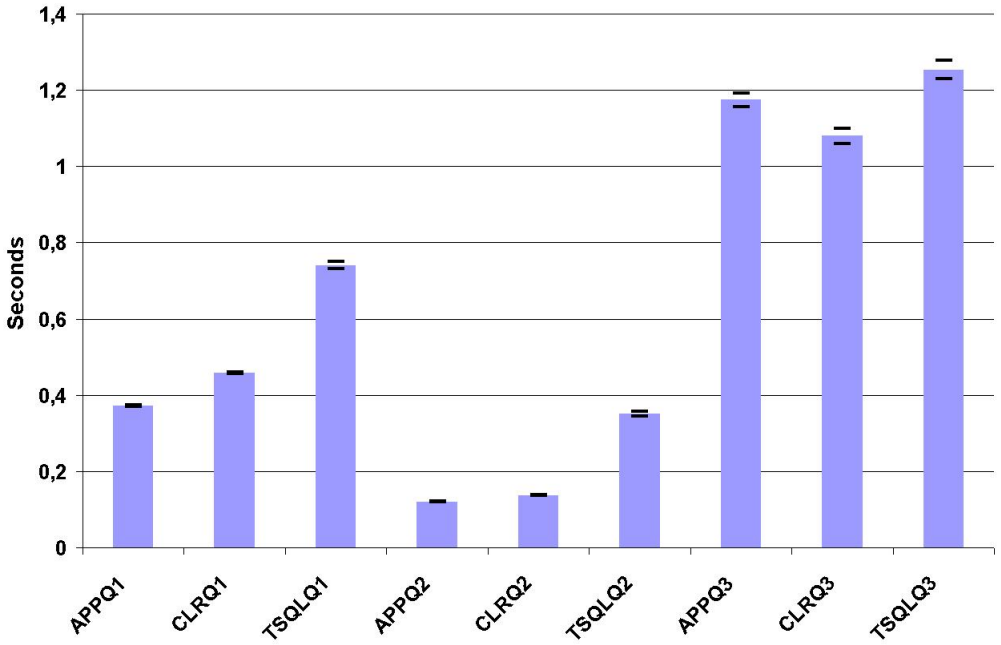


Figure 6.3: Response Time - 63 Workload Units

However, even though the load on the database is increased, the results for Query 3 displays an interesting finding, namely the SQLCLR is the fastest and the difference between the application server and T-SQL is marginal if considering the upper confidence interval value of 1,19 for application server and lower confidence interval value 1,23 for T-SQL. Recall that Query 3 uses the data from the `SELECT` statement to first insert it in the database and subsequently delete it. For both SQLCLR and T-SQL, this does not imply additional round trips to the database, since the code executing the statements reside on the database. However, the application server requires additional round trips to the database for both inserting and deleting, thereby increasing the overall response time. These additional round trips can also be read from the network utilization in Section 6.3.4.

Another issue to consider compared to 1 *WU*, is the decrease in confidence interval for primarily Query 1 and 2 if considering the ratio between the confidence interval and mean value. However, this can be contributed to the increased execution time compared to 1 *WU*.

125 Workload Units

Since the ratio for the graph for 125 *WU* is almost identical to that for 63 *WU*, the figure is placed in Appendix K. Not much can be concluded from the figure that has not already been noted for 63 *WU*. It does however indicate the tendency for 63 and 125 *WU* might span beyond 125 *WU*, i.e. the application server performs best for Query 1 and 2 and SQLCLR for Query 3. However, further experiments have to be performed to make a conclusion on this subject.

6.3.2 Throughput

To get an idea of how well the different experiments scale, it makes sense to look at the throughput for different loads on the system by varying the amount of *WU*. The throughput is, as mentioned in Section 4.5, the rate of function executions per unit of time the system is able to service.

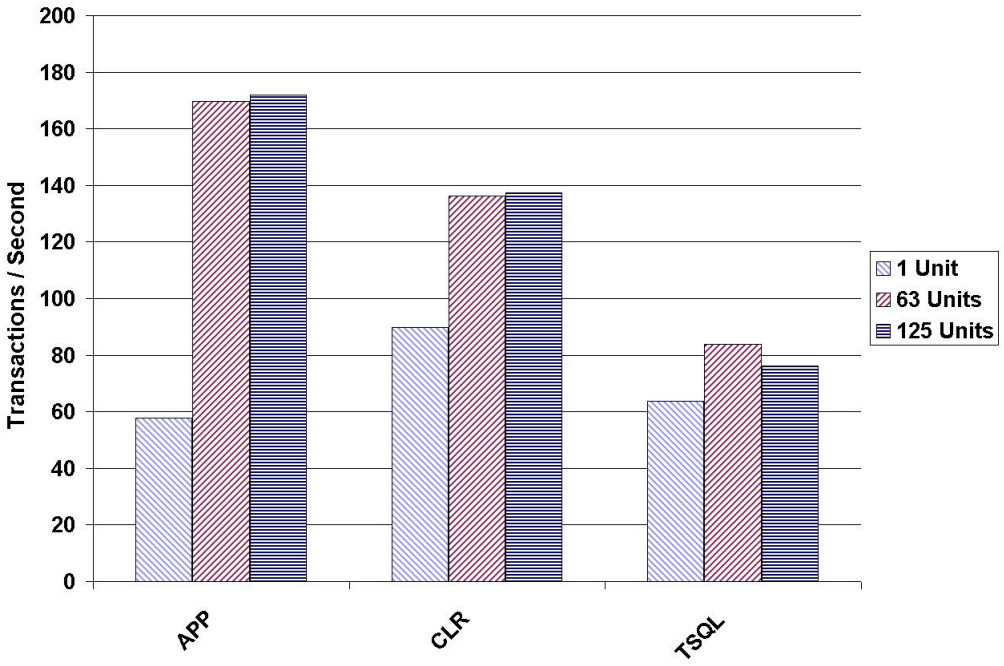


Figure 6.4: Throughput - Query 1

Query 1

Figure 6.4 shows a somewhat mixed tendency, since both SQLCLR and the application server delivers an increase in throughput for all three *WU* levels, whereas the throughput for T-SQL declines for 125 *WU*. Generally speaking, the application server delivers the best throughput, except for 1 *WU*.

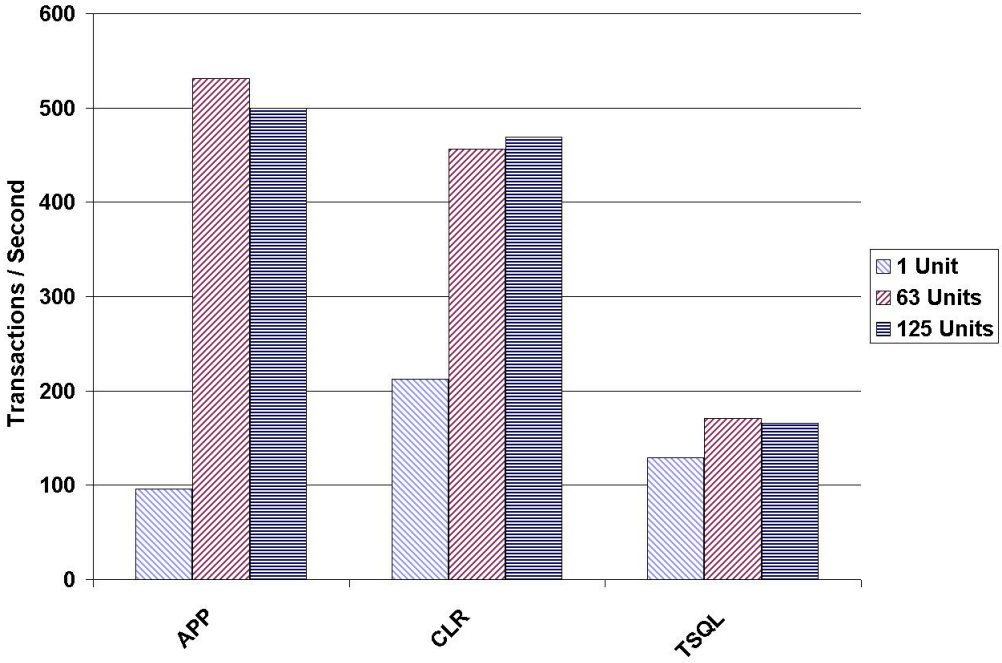


Figure 6.5: Throughput - Query 2

Query 2

When examining Query 2 as illustrated in Figure 6.5, it shows the best throughput for both T-SQL and application server is for 63 *WU*, which could indicate the experiments only scales up to a certain level. However, for SQLCLR, the best throughput is still for 125 *WU*, with 474,2 transactions/second compared to 458,1 for 63 *WU*. Furthermore, compared to the throughput for Query 1, the SQLCLR is closer to the performance of the application server.

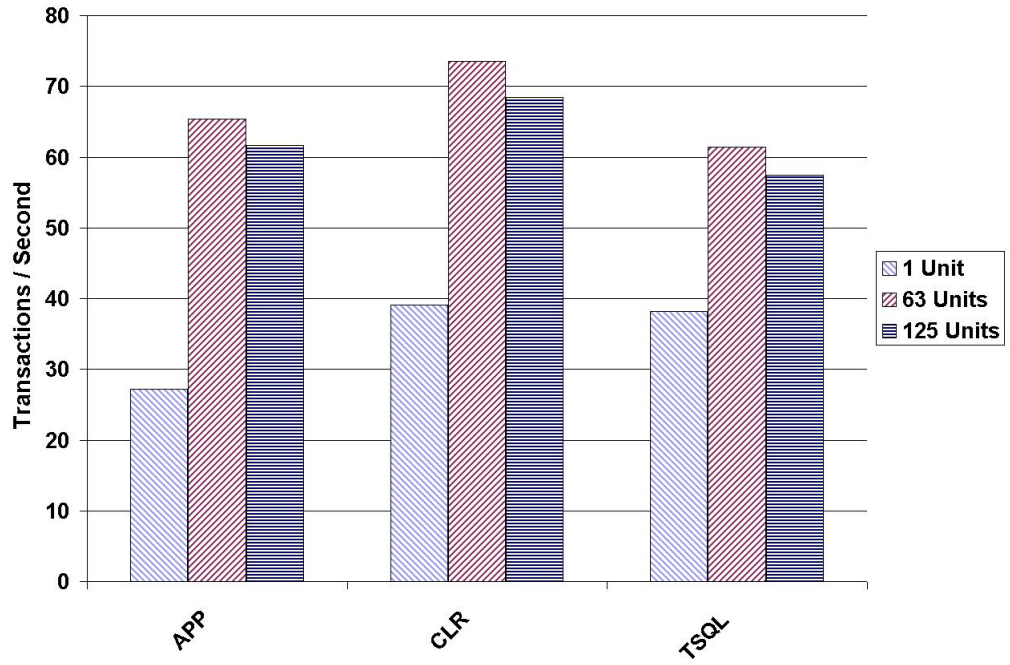


Figure 6.6: Throughput - Query 3

Query 3

Figure 6.6 shows the throughput for Query 3. Common for all three placements – the best throughput is for 63 *WU*, albeit only with a slight advantage over 125 *WU*, considering the different y–axis scales for the three queries. The best overall throughput is for the SQLCLR; however, compared to Query 1 and 2, the T-SQL is suddenly not the worst performing of the three if considering all three *WU*. This can be contributed to what was discussed for response time concerning the insert and delete operations in Query 3.

As all three figures show, the tendency seems to be the throughput for 1 *WU* is much worse than that for both 63 and 125 *WU*. The throughput for 63 and 125 *WU* is almost equal but the fact that 63 *WU* generally produce the best throughput could indicate the maximum load has been reached for the system for 125. Furthermore, there is a mixed tendency between SQLCLR and the application server concerning the best throughput; however, the SQLCLR appears to perform the best.

Finally, it is interesting to observe the rather large difference between the throughput for the three queries. For Query 1, the maximum transactions/second is 172,

for Query 2 it is 531, and for Query 3 it is 73. This is not surprising, since Query 2 is also the simplest query compared to Query 1, which contains both aggregates and joins. Query 3 uses the same `SELECT` statement as Query 2, however additionally it also inserts and deletes data, which lengthen the execution time and thereby reduces the throughput. Furthermore, insert and delete transactions are continuously logged on the database in order to enable roll–back if an error occur, which also imply additional overhead.

6.3.3 CPU Utilization

As described in Section 4.5, CPU utilization is the percentage of CPU time used during the execution of the functions on the application server and database respectively. As the Celsius machine has two CPUs, the utilization for the database is divided by two in order to make it comparable. The confidence intervals for the graphs are illustrated individually for the application server and database in Appendix K. Furthermore, the figure for 1 *WU* is illustrated in Appendix K, since the results are inconclusive.

63 Workload Units

When examining Query 1 and 3 in both Figure 6.7 and Figure 6.8, the tendency is the overall application CPU utilization for an experiment is almost identical for the three placements. Not surprising, there is also a connection between how much functionality is placed on the application server and database respectively and the CPU utilization.

What is surprising however, is that the CPU utilization for Query 2 is rather small both for SQLCLR and application server. For 63 *WU* the combined CPU utilization is 5,13% for the application server and 6,86% for the SQLCLR. The reason for this low utilization is probably that Query 2 is a very simple query with no joins or aggregates. Therefore, the execution time of the query is very short and the performance counters might miss high CPU utilization readings considering the interval between successive reads is 1/10th of a second. The CPU utilization when using T-SQL is not quiet as low, which can be because the response time for T-SQL is higher and the counters therefore measure more values, thus achieving higher accuracy.

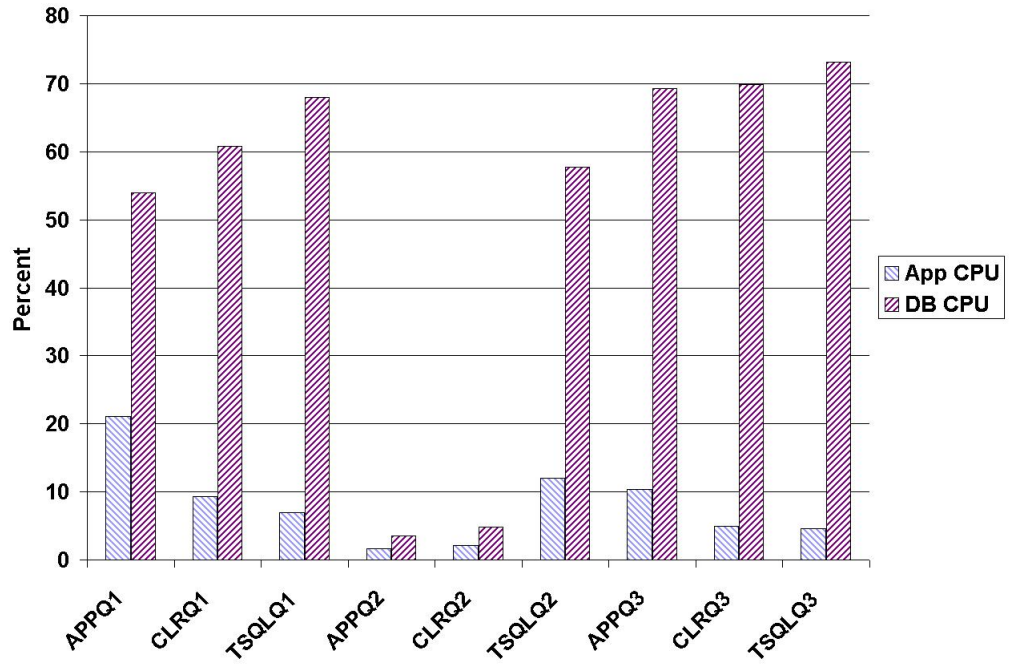


Figure 6.7: CPU Utilization - 63 Workload Units

125 Workload Units

With 125 *WU*, the measured CPU utilization for Query 2 for application server and SQLCLR gives a more accurate result. The application server utilizes 50,08% overall and SQLCLR 61,63%. This is still lower than for the other queries, but as mentioned earlier; this is because Query 2 is simpler than both Query 1 and 3.

6.3.4 Network Utilization

The network utilization measures how much data is transferred across the network on the application server, both inbound and outbound. However, since it is not possible to fit the setup into a closed network, there might be small deviations because of this, since other machines might be accessing the network. Furthermore, it should be noted the amount of bytes transferred is not solely tied to the queries, since monitoring performance counters remotely accounts for additional network traffic. The figure for 1 *WU* is placed in Appendix K, since the results are inconclusive because all values lie in the same confidence interval.

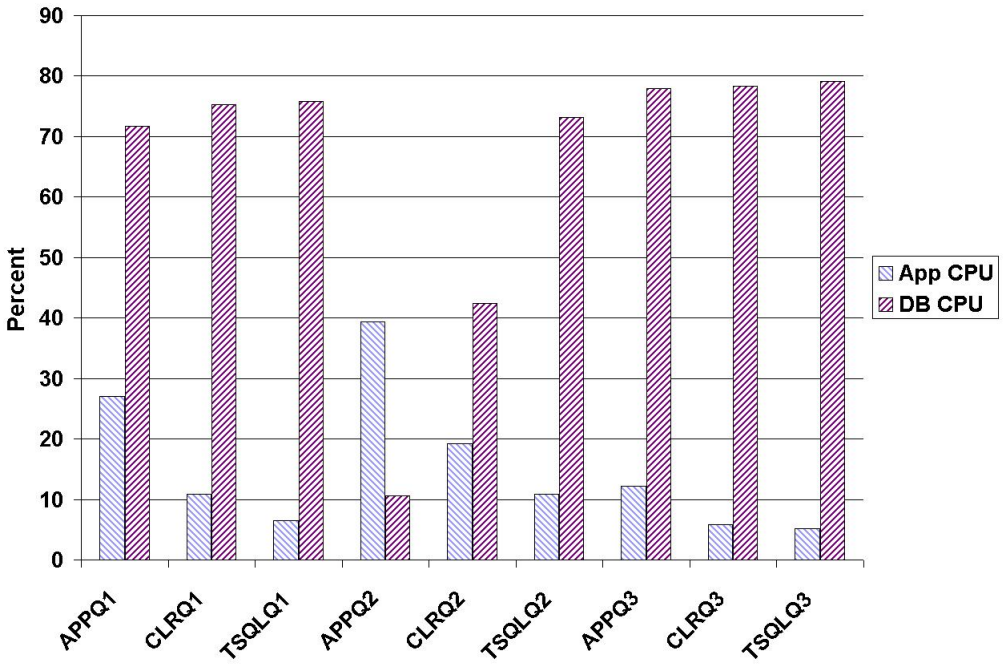


Figure 6.8: CPU Utilization - 125 Workload Units

125 Workload Units

Figure 6.9 illustrates the network utilization for 125 *WU*. The figure for 63 *WU* has been excluded since it is similar to that for 125. Compared to 1 *WU*, the figure for 125 draws a completely different picture, where the confidence interval seems to be somewhat settled. SQLCLR clearly uses the least amount of bandwidth and the applications server the most.

The results of this figure are not surprising, since for all three queries, the SQL statements used for the application server are pure ad–hoc SQL strings, compared to the parameterized approach for T-SQL and SQLCLR. In effect, this results in more characters having to be transferred across the network, for Query 1 this is 364 chars for the application server and 22 for T-SQL – a factor 17. Obviously as the number of *WU* increase, the required bandwidth is increased as well.

However, this cannot alone account for the large difference for the application server for Query 3. Recall that Query 3 also inserts and deletes data in the database, for T-SQL and SQLCLR this does not require additional round trips to the database as was discussed earlier. In contrast, the application server not only has to make additional round trips, but it also has to transmit the batch of `INSERT`

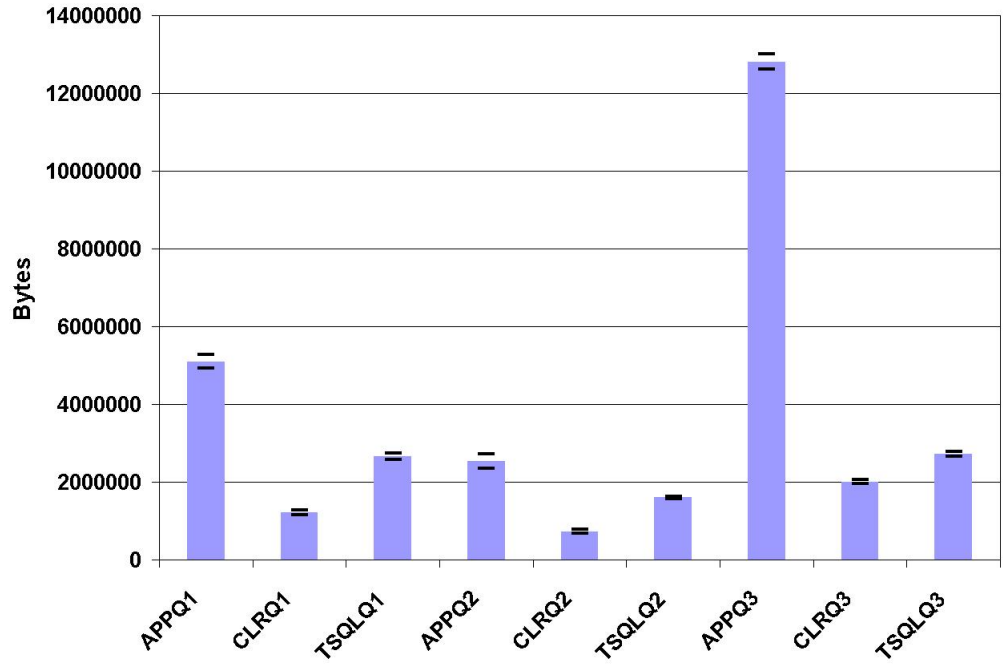


Figure 6.9: Network Utilization - 125 Workload Units

statements where each statement is approximately 73 chars, which grows along with the number of *WU*.

6.3.5 Memory Consumption

Analysing the data concerning memory consumption, the results are quite one–sided as illustrated in Figure K.11 in Appendix K. For all the experiments, the difference is only 0,18%, where the amount of free memory is approximately 2,6 GB of 5.3 GB for Scenario1 and 640 MB of 3 GB for Scenario2. This could indicate the memory settles once it has reached a certain level and thus does not cause any bottlenecks that might affect the performance of the experiments, since the memory limit has not been reached.

6.3.6 Disk Queue Length

The length of the disk queue on the database is an estimate of the number of outstanding requests for the disk. The number of outstanding requests contains both requests that are not yet serviced as well as the requests that are currently

serviced. As illustrated in Figure K.9 in Appendix K, the length of the disk queue even for 125 *WU* is very short. Generally speaking, a disk queue length below 2 during the execution of an application is satisfactory [32]. Therefore, the disk utilized by the database is not a bottleneck for the system and does not have a significant influence on the results gained from the experiments.

6.3.7 Hardware

As already stated, the experiments include two different hardware configurations. One configuration with a heavy–duty database and a lightweight application server (Scenario1), and another with a lightweight database and a heavy–duty application server (Scenario2). Even though no resources have been added, Scenario1 still performs twice as fast overall as Scenario2, with a mean response time of 0,452 seconds compared to 0,908 seconds. When examining the specific experiments, the same tendency seems to be the case, where Scenario2 is generally slower than Scenario1.

However, it is interesting to examine the three different queries in this context. For Query 1, Scenario2 only requires about 50% more time than Scenario1, while for Query 2 it requires twice as much time and for Query 3 three–time as much. An explanation for Query 2 is because of the increased CPU utilization, which e.g. uses an aggregate as opposed to Query 1 that is straight table retrieval. Query 3 on the other hand can be explained because it performs a number of inserts and a delete in the database, which demands a lot of writing on the disk. As such, the performance of the disk is very important for Query 3. Therefore, Query 2 and 3 benefit the most from having a heavy–duty database.

6.4 Summary

The results presented in this chapter generally indicate an advantage of using the application server or SQLCLR rather than T-SQL. However, this does not mean that T-SQL is never preferable, most noticeable are the experiments c276, c68, c106, c212, and c213, which are listed in Appendix I. These experiments are a combination of the following factors and levels; 1 *computations*, 1 or 6 *number of queries*, *Query 3*, *hardware Scenario1*, 1 or 17 *data volume*, and 1 or 63 *WU*. These experiments show a general pattern, namely the *number of queries*, *data volume*, and *WU* factors have the least impact on T-SQL, except for 125 *WU*. Furthermore, by examining Table 6.1, it is possible to get an insight into how T-SQL overall compares to SQLCLR and the application server.

	Response Time
AppQ1	0,29075649
CLRQ1	0,38132076
TSQLQ1	0,47114546
AppQ2	0,12895275
CLRQ2	0,1477825
TSQLQ2	0,24352864
AppQ3	1,47130295
CLRQ3	1,45850985
TSQLQ3	1,5267751

Table 6.1: Placement by Query

In Table 6.1 is listed the mean response time, which is for all combinations of factors except *locality* and *type of queries*. The table shows that T-SQL is the worst performing of the three, although there is only a small difference for Query 3, which is the query that T-SQL performs best on. A reason for this is as mentioned in [3] that T-SQL is good at data access that contains little or no procedural logic at all. As such, the reason SQLCLR performs better than T-SQL is likely because there are computations in all the experiments, which was also noted in [3] to be in SQLCLR's favour.

In general, across all three queries however, the application server is the best performing followed by SQLCLR. The reason for this is likely because of the increased load introduced by executing T-SQL and SQLCLR code on the database. This is also noted in [16], where it is mentioned that performance and scalability might be affected by the increased load on the database. This is also the conclusion reached when examining the figures in this chapter, although some interesting observations were noted as well. For example, the network utilization of the application server, where in best cases it transferred more than twice as much data across the network and in worst cases more than six times.

Chapter 7

The Metric

In this chapter, the definition of the metric is presented using the data from the conducted experiments, as was presented in the previous chapter. Additionally, to ease the use of the metric, a tool is implemented that can be used by developers to input levels for the different factors specified in the metric. The tool is presented along with a sample usage of the metric.

7.1 Metric Development

To develop an effective software metric some conditions can improve the quality of the metric as stated in [6]. The applicable of these conditions have been used to define the metric in this thesis:

- **Simple and computable**
  It is easy for the developer to use the metric and it does not demand inordinate effort or time. To achieve this condition, a tool is implemented where the developer only needs to input the levels for each factor. Thereby, the developer does not waste any time or effort on mathematical formulas and calculations.
- **Intuitively persuasive**
  The metric satisfy the developer's intuitive notions by outputting an easy understandable measure for the best *locality* and how certain the measurement is.
- **Consistent and objective**
  The metric result is unambiguous and always indicates a best *locality* using the data from the experiments.

- **Consistent in the use of units and dimensions**
  The factors considered can be put in any combination not leading to bizarre combinations that are impossible in real situations.

The metric has been developed with these conditions in mind and all conditions have been met. To use the metric, the developer has to input the specified levels for the different factors, where some of these levels can be determined precisely, while others are only estimates. *Number of queries* and *computations* is likely not known at design time and the developer therefore has to make a qualified estimate. *Type of queries* can in many cases be determined at design time, since the design reveals what kind of data that needs to be retrieved from the database. *Hardware* is in most cases known at design time, but can of course be upgraded at a later stage. *Data volume* and *WU* are likely also known at design time but in most cases they increase over time, as such, an estimate has to be made on what level can be expected. When these levels are determined, they can be specified for the metric and the best *locality* can be calculated with a given certainty.

7.1.1 The Metric

The most optimal approach would be to develop a metric that has the benefit of 100% accuracy for the levels experimented on, while still retaining a certain level of accuracy on the in–between. Thus, in order to estimate a response time for a scenario that has not been conducted in the experiments, it is necessary to specify a response time between two scenarios. This is possible if only one factor deviates from the levels; however, it is somewhat more complex if more factors deviate. Therefore, a response time is calculated for each factor individually.

For the factor examined, the level specified by the developer is used, but for the other factors, the levels that are closest to the ones chosen by the developer are considered. For example, if 100 *WU* is chosen, the metric determines the closest value, which in this case is 125, considering the levels in the metric are specified for 1, 63, and 125 *WU*. When calculating the value in–between for the *WU* factor, the following formula is used:

$$Response\,time \;=\; res63 + (\frac{res125 - res63}{int} \times (dev - int2))$$

$res63 =$ Response time for the scenario with 63 *WU* and the levels for the other factors closest to the developers choice.
$res125 =$ Response time for the scenario with 125 *WU* and the levels for the other factors closest to the developers choice.

$dev =$ The developer's choice for *WU*
$int =$ Interval between $res63$ and $res125$, in this case $125 - 63 = 62$
$int2 =$ The lowest level for the factor. In most cases 1, but in this case 63

The same formula is used for the other factors, by replacing $res63$ and $res125$ with the corresponding high/low levels. However, the factors *type of queries* and *hardware* cannot deviate from the experimented levels, since it is not possible to draw a line between the response time of one level of these factors and the other. As such, the developer has to specify a level that corresponds to a level for the two factors.

Furthermore, since not all factors from the experiments affect response time equally, the metric puts a weighted score on each factor. The weighted score is calculated by assuming that functionality on the database is slower than on the application server, as was the general conclusion from the results presented in Chapter 6. The difference between the two is calculated in percent and if the database is the fastest, a negative percent value is given. For each factor, the difference in percent between the levels is calculated and this value determines how large the weighted score will be. A table with these values is shown in Appendix H.

Finally, the values for the different factors are added together and the values for T-SQL, SQLCLR, and application server can be compared. The location with the lowest value indicates location where the lowest response time can be expected.

7.1.2 Sample Usage

The process explained in the previous section has been implemented into a tool. The tool is straightforward and only requires the developer to input the levels corresponding to the system, except for the *type of queries* and *hardware* factors as was mentioned earlier. These two factors can only be chosen from a drop–down menu and as such cannot be generic.

Figure 7.1 illustrates a screen shot from the tool. The sample is specified for the scenario with 2 *number of queries*, 4 *computations*, *Query 3*, *Scenario1*, 12,4 GB *data volume*, and 85 *WU*. As the result illustrates, the best *locality* is SQLCLR, which is 1,58% faster than T-SQL and 10,19% faster than the application server. These percentages are indications of how certain it is that one *locality* is better than another, as such they are not accurate results that specifies SQLCLR is exactly 1,58% faster than T-SQL.

Therefore, this particular result specifies that SQLCLR is almost the same as T-SQL and slightly better than the application server. The calculations made to

determine the percentages for this scenario are listed in Appendix L.

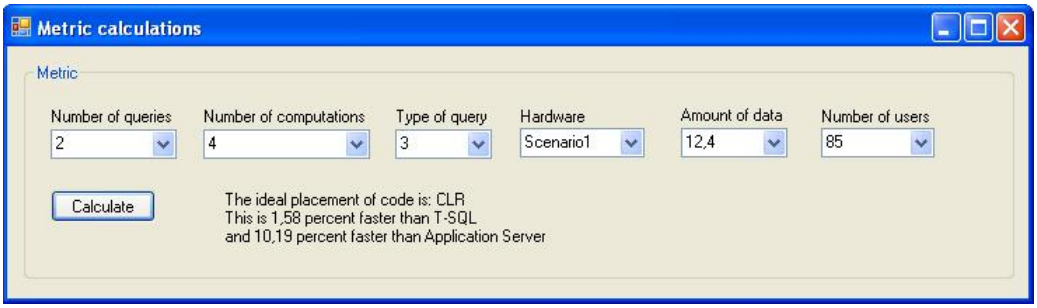


Figure 7.1: Metric Tool Screenshot

7.2 Metric Accuracy

The metric output has been examined with the results from the experiments and the accuracy of the metric is satisfactory. The metric's output corresponds to the performed experiments. The two factors *type of queries* and *hardware* cannot be varied in the metric however *number of queries*, *computations*, *data volume*, and *WU* can. When variations are made on these factors, the metric's output values varies accordingly. The accuracy of these variations is difficult to determine precisely, because it has not been possible to validate the output with a sample system.

The metric is therefore not 100 % verified, since it would demand experiments on all possible scenarios. However, an approach to verifying it can be by developing an application for each of the three *locality*. The metric should then be used on the design documents and the results compared to the performance of the three implementations. This would still only be a validation of one specific scenario. As such, testing all possible scenarios would demand an infinite amount of time.

Part IV

Discussion

Chapter 8

Discussion

This chapter contains a general discussion of the core parts of the thesis. This includes the existing guidelines for locality as was outlined in Section 1.3. Furthermore, is a discussion of the validity of the included factors and their levels. Finally, there is a discussion of the developed metric.

8.1 Comparing existing Guidelines

As described in Section 1.3, Microsoft has published existing guidelines for deciding placement of functionality between the application server and database. The published article contains guidelines for whether the functionality should be implemented in either SQLCLR or T-SQL. There is however, one statement in particular in the article, which contradict best practice concerning databases. For example, the article assumes utilization of `CURSOR` in T-SQL, which is discouraged in databases [26, 32]. Whether this is due to special cases or because the authors are unaware of the drawbacks of `CURSOR` is unknown. Hence, the outlined guidelines might deviate if optimizations are done on both the code for T-SQL and SQLCLR.

Most of the provided guidelines in [3], concerning code placement, are primarily concerned about security, reduction in network utilization, and performance. In this thesis however, the focus is only on performance and scalability and the results obtained from the experiments is used as basis for the developed metric. With this metric, it is possible to determine whether the code should be placed on the application server or database.

As stated in Chapter 6, the majority of the experiments benefits from being placed on the application server. However, in several situations such as the experiments

containing both `SELECT`, `INSERT`, and `DELETE` statements, the database outperforms the application server. Hence, the results indicate that functionality utilizing data obtained from the database to insert or delete data benefits from being placed on the database. Of course other characteristics of the functionality have an influence as well, e.g. the amount of computations needed to execute the functionality as these computations use CPU time that could otherwise have been used processing queries.

As mentioned, the article [3] provides guidelines concerning T-SQL vs. SQLCLR on the database. These guidelines generally correspond to the results obtained from the experiments performed in this thesis. For example, the results indicate an advantage for the SQLCLR when the retrieved data needs additional processing. However, if the functionality primarily contains declarative statements an implementation in T-SQL performs better than the equivalent SQLCLR.

8.2 Experimental Design

During the preliminary search of related work, we did not come across any analysis of the impact of factors like those in our metric. The related work only contained guidelines, on which the developer needs to rely. Therefore, a full factorial design was used in order to provide insight about the influence of each of the included factors. By using this approach, the impact of each of the factors was determined. This knowledge would not have been obtained by other methods such as the fractional factorial design. A drawback however, of the full factorial design is the extensive amount of experiments required to be performed when including multiple factors. Hence, the number of levels included in each factor needed to be considered carefully in order to cover performance and scalability sufficiently.

8.2.1 Factors

The factors and their corresponding levels included in the experimental design have different impact on the response time. Overall, the effect of the factors and their interactions explain 98% of the result obtained when measuring response time, which is determined by the effect screening performed by the JMP application. The effect screening provides a measure of the impact of each of the factors as well as their interactions. The last 2% can be contributed to excluded parameters and the accuracy of the performance counters. Since, the model explains 98% of the variation, the model appears to be appropriate according to the guidelines in [7].

Data Volume

As explained in Section 4.2, the *data volume* factor does not pose any significant impact on the result. The reason for this is the optimization done on the database with indexes and statistics improved the performance of the queries with 99%. The optimizations reduce the amount of data required to be traversed for both levels of *data volume*. Therefore, as the data traversed for both levels is reduced significantly, the difference concerning response time becomes insignificant. If the *data volume* factor should have an impact on the result, the difference between the levels must therefore be larger. Alternatively, the queries could be changed such that a larger part of the tables is traversed. Thereby, a difference in response time is revealed. However, changing the *data volume* implies going beyond the SME target group.

Hardware

Even though a part of the hardware used in the experiments is from the computer lab at Aalborg University, the computation capability is limited. Comparing the Celsius machine to other applicable machines for SME, the capability of Celsius is limited. Despite the limited capability of the hardware, the *hardware* factor significantly influences the result as explained in Section 6.3.7. However, it is not possible to predict the influence of hardware configurations that differ from those in the experiments, as the machine consists of multiple components. Each of the components in the machine affects the processing capability and even a small change can induce a significant impact on the response time for the experiments. Therefore, additional experiments need to be performed utilizing different hardware configurations in order to determine the impact of the factor and be able to provide the metric with levels that correspond to the need of the developer.

Workload Unit

The choice of levels in Section 4.3.6 for the *WU* factor is influenced by the *hardware* factor as well as the SME target group. The response time of the experiments is varying due to the levels. The choice of 1 *WU* is debatable as the response time of the experiments with 1 *WU* is very low. The low response time in conjunction with the relatively low accuracy of the measurements obtained by the performance counters degrades the worth of these experiments. Furthermore, a SME application is most likely not supposed to support only 1 *WU*, as such the results are primarily useful when calculating the values for the metric between 1 and 63 *WU*. Another consideration concerning *WU* is the upper level of 125 *WU*. Whether this

level is sufficient depends on the usage of the application. If only the employees of a SME is expected to use the application, then 125 *WU* is appropriate. However, if the application e.g. is the official website of an enterprise, then 125 *WU* is insufficient. The results however, indicate that at least for experiments containing both `SELECT`, `INSERT`, and `DELETE` statements, additional *WU* would degrade performance, as the throughput for 63 *WU* is higher than 125 *WU*. Hence, when considering the available hardware, the levels appear to be appropriate.

Type of Queries

The factor *type of queries* exhibited a significant impact on the results. As described previously, the complexity of the queries such as aggregates and joins affects the response time of the overall functionality. The reason for utilizing fast queries is due to a preliminary execution of all the experiments. The preliminaries revealed that when the complexity of the queries increased and thereby the execution time, placing the functionality on the application server is the better choice. In addition, by using complex queries, the overall execution time of the experiments increased to weeks or even months. However, if the hardware configuration for the experiments is improved, it could be interesting to examine whether functionality containing complex queries would benefit from being located on the database.

A property of the queries that could easily be changed is the number of rows returned by the queries. Currently, the number is set to 10 for all the queries, as it is difficult to compare the queries if they return different number of rows. This is because the number of results returned alters the *computations* factor, as the code in the loops is executed according to the number of rows. Furthermore, the preliminaries did not indicate any change concerning placing the functionality on the application server or the database when altering the number of returned results.

Number of Queries

As mentioned in Section 4.3.1, the levels for the *number of queries* factor are determined from the number of queries observed in the Compiere solution. The advantage of using the Compiere solution is that it contains a significant range of functionality and represents the SME target group [12]. Furthermore, as the response time of the used queries is low, the influence of each query is limited. Therefore, the levels are considered appropriate for the development of our metric.

Computations

The definition of the *computations* factor is debatable. A computation consists of a certain amount of method invocations and loops according to the statistics obtained from the Compiere solution as stated in Section 4.3.1. As stated in the definition, the approach is inspired by a similar one used in SPE [9]. However, the use of method invocations as a measure of the complexity is debatable. In our definition, method invocations are given the same weight even though the execution time or utilization of resources differs significantly. That is, a string append gives the same weight as an implicit traversing of a large collection of objects or a remote connection to another server. These methods are obviously not equal concerning resource consumption. However, when examining the implementation of the experiments performed in this thesis, all of them contain comparable method invocations, as primarily data extraction from result sets and string appends are used. Thereby, the situation is avoided where unequal methods are provided the same weight. If a particular piece of functionality is determined to contain methods that require additional resources compared to the ones used in the experiments, it is possible to increase the value for *computations* accordingly. Hence, the metric is able to consider these methods.

8.3 The Defined Metric

As stated in the previous chapter, the metric tool is only known to be 100% accurate in the cases where the experiments described in this thesis and the input made by the developer is identical. Beside these cases, the metric can only give an informed guess based on the performed experiments. Hence, the farther away the input values are from the performed experiments, likely the more inaccurate the result will be. The only way to reduce this inaccuracy is to perform additional experiments, either to determine whether the result of the metric is in fact accurate or to include the results of the additional experiments in the metric and thereby improve it.

Chapter 9

Conclusion

The goal of this thesis was to define a metric that can help determine when it is beneficial to place functionality on either the application server or database. The metric was defined by performing a number of experiments that included various measures to help determine the response time and utilization of hardware resources, such as CPU and disk. The response time measure provided the most useful information and was the source for defining the metric, which was implemented into a tool. The tool allows the developer to easily determine where a certain piece of functionality should be placed.

In general, the results gathered from the experiments displayed a noticeable advantage to using the application server for functionality containing pure `SELECT` queries. However, especially the SQLCLR performed better than the application server for functionality containing both `SELECT`, `INSERT`, and `DELETE` queries. This was contributed to the additional round trips to the database required for the application server, for the `INSERT` and `DELETE` queries.

The additional round trips combined with the ad–hoc SQL approach resulted in a doubling of the network utilization compared to T-SQL and SQLCLR. This was also noted in [3] as being one of the strengths of moving functionality away from the application server. While network utilization is not included in our metric, it is still an important consideration to keep in mind for large–scale applications that require huge amounts of bandwidth. This is especially important if the network acts as a bottleneck for the application, since noticeable improvements can be achieved by deploying functionality on the database. As such, if bandwidth is a consideration, our experiments displayed a clear advantage of placing functionality on the database. This also leads us to conclude the same is the case if e.g. only a single value is required to be retrieved from a large result set.

When examining the scalability, the functionality containing pure `SELECT` queries

shows a mixed tendency concerning throughput when increasing *WU* from 63 to 125. For SQLCLR, the throughput increases, whereas for T-SQL it degrades, the application server on the other hand shows both degrading and increasing scalability. However, examining the throughput for functionality containing both `SELECT`, `INSERT`, and `DELETE` queries, it degrades for all three *locality* when the *WU* is increased from 63 to 125. Therefore, the experiments containing only `SELECT` queries scales better than the experiments containing `SELECT`, `INSERT`, and `DELETE` queries.

Thus, evaluating the introduction of the SQLCLR in MSSQL2005, it should definitely be considered when designing systems that utilize MSSQL2005. First and foremost, it introduces improved security and reliability, which was not possible using XP [3]. Furthermore, the possibility of writing code in a .NET supported language both on the application server and database, which in our case allowed for an easy implementation of the functionality in SQLCLR and will likely appeal to many developers. In addition and just as important is how well the SQLCLR performs and scales and as we discovered it certainly was an alternative to implementing all the functionality on the application server. As such, we believe it will become more popular as more developers become aware of its possibilities.

While the developed metric has limitations, such as network utilization, it is still applicable for the goal stated for the metric. Namely, that it should be able to help the developer determine where functionality should be placed when considering performance. This is also the case within the boundaries stated for the metric. The accuracy of the metric however, is difficult to predict except for the specific scenarios that were conducted experiments on. These specific scenarios however, are satisfactory accurate and in order to reduce the uncertainty for the in–between levels, additional experiments would have to be performed.

Besides defining a metric, we have implemented a tool, which is able to perform the calculations required in the metric for the developer. The tool developed for the metric is straightforward to use and does not require a high learning curve for the developer. As was specified in the problem statement, the metric could be used as early as the design phase. This is also the case with the metric, as long as the developer is able to estimate the expected levels of the factors.

As such, the new possibilities concerning the hosting of the SQLCLR in MSSQL2005 presented a number of interesting findings concerning the performance and scalability criteria. Furthermore, the evaluation resulted in the development of a metric, which was implemented in a tool. By providing the tool with the desired levels of the factors, it calculates the value of the metric for the developer. Hence, the tool is helpful for a developer working with distributed systems.

Chapter 10

Future Work

In this thesis, we examined the new addition of a hosted CLR in MSSQL2005 and how well it performed compared to placing functionality on the application server and database respectively. However, other solutions exist beside the standard approaches examined in this thesis, such as ORM tools like NHibernate for .NET and Hibernate and TopLink [34] for Java. Besides performing the standard create, read, update, and delete operations these tools also support caching of both queries and data, which is one of the parameters not varied in this thesis. Furthermore and just as interesting, they map tables from the database to classes in the application, thus retaining the object–oriented syntax and semantics while working with a relational database. Some of these solutions were analysed and tested in our previous project [1] and demonstrated substantial performance gain compared to using the ad–hoc approach in e.g. C# or Java. In this thesis, the results were somewhat mixed with best performance between the application server and SQLCLR, as such it could be interesting to examine how the results would be if the ORM tools were utilized on the application server.

Another important future work would be to perform experiments on additional levels of the factors used in the metric. This should be done to ensure the metric is not only accurate for the levels experimented on in this thesis but also for the in–between levels. Furthermore, as was noted in Chapter 6, the experiments for 1 *WU* had a low response time, causing the results to be highly inaccurate with large confidence intervals. A solution to this could be to use longer running queries, such that deviations in CPU and network utilization would have less effect. However, on the other hand, longer running queries would also equal more utilization of the database and thus the overall results will likely shift more in favour of the application server. Aside from experimenting on additional levels, the amount of included factors could also be increased. Six factors beside the

locality are included in the metric, but with more time and resources it could be interesting to make experiments on more factors. Thereby, the metric will be able to cover scenarios closer to the systems, which the metric will be used on.

As stated earlier in this thesis, the performed replications have a certainty of 80% that the measured mean values for the response time does not deviate with more than 20% from the real mean value. It has not been possible with the time and reasources available to make this measure more accurate; however, it could be very useful for the validity of the metric to make the accuracy higher in the future.

Furthermore, it could be interesting to focus on additional criteria beside performance and scalability, considering they are not the only concerns for the developer when choosing *locality*. Therefore, by defining a metric that includes e.g. the criteria specified in the ISO 9126 standard, the developer does not have to examine additional tools, guidelines, and metrics.

A critical point to consider is to use the metric on an actual application. One thing is to define a metric using results gathered from a simulation of a real system, another is to try to apply the metric. A possibility could be to port parts of the Compiere solution to the .NET platform, considering it is used to determine a number of the levels for the different factors. Hopefully, this could give an indication of how accurate the metric is when used on a real system.

In this thesis, we have only implemented and evaluated functionality in C#. However, it could be interesting to analyse other .NET languages such as Visual Basic .NET, in order to determine any differences in performance and scalability. If there are any differences between the .NET languages, it would likely affect the decision of placing the functionality on the application server or the database.

As an additional follow up, we are also aware of the support for both JVM and CLR in Oracle [35] and DB2 [36], where the CLR is hosted as an external process, outside the actual database engine. In effect, some of the benefits of integrating the CLR as in MSSQL2005, such as the internal models for memory, threads, and security management are not as apparent and other limitations such as limited support for other functionality than SP. However, it could still be interesting to examine how these implementations of the CLR perform compared to that in MSSQL2005 and in general, what the performance benefits are on the Java platform.

Part V

Appendix

Appendix A

Setup Specification

- SCENICO 14
  - AMD 1.8GHz Sempron 3000+
  - Single processor
  - 1.2GB RAM
  - 80GB IDE HDD
  - Windows Server 2003
  - Microsoft SQL Server 2005 v. 9.0.2047
- Celsius 1
  - Intel(R) Pentium(R) 4 CPU 3.00GHz
  - Dual processor
  - 3.2GB RAM
  - 141GB, SATA max UDMA/133 disk drive
  - Windows Server 2003
  - Microsoft SQL Server 2005 v. 9.0.2047

Appendix B

Experiment Queries

Listing B.1: Query 1

```
SELECT TOP 10
  L_SHIPMODE,
  L_PARTKEY,
  L_SUPPKEY,
  100.00 * SUM (
    CASE WHEN P_TYPE LIKE 'PROMO%%'
    THEN L_EXTENDEDPRICE*(1-L_DISCOUNT)
    ELSE 0 END
    ) / SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS PROMO_REVENUE
FROM
  LINEITEM,
  PART
WHERE
  L_PARTKEY = P_PARTKEY AND
  L_SHIPDATE >= '1995-09-01' AND
  L_SHIPDATE < dateadd(mm, 1, '1995-09-01')
GROUP BY
  L_PARTKEY,
  L_SUPPKEY,
  L_SHIPMODE
```

Listing B.2: Query 2

```
SELECT TOP 10
  L_SHIPMODE,
  L_PARTKEY,
  L_SUPPKEY,
  L_TAX
FROM
  LINEITEM
WHERE
  L_SHIPDATE >= '1994-01-01' AND
  L_SHIPDATE < dateadd (yy, 1, '1994-01-01') AND
```

```
11    L_DISCOUNT BETWEEN .06 - 0.01 AND
12    .06 + 0.01 AND
13    L_QUANTITY < 24
```

Listing B.3: Query 3

```
1  SELECT TOP 10
2    L_SHIPMODE,
3    L_PARTKEY,
4    L_SUPPKEY,
5    L_TAX
6  FROM
7    LINEITEM
8  WHERE
9    L_SHIPDATE >= '1994-01-01' AND
10   L_SHIPDATE < dateadd (yy, 1, '1994-01-01') AND
11   L_DISCOUNT BETWEEN .06 - 0.01 AND .06 + 0.01 AND
12   L_QUANTITY < 24
13
14
15 INSERT INTO [dbo].[TEMPTABLE] VALUES (L_SHIPMODE, L_PARTKEY,
       L_SUPPKEY, L_TAX)
16
17 DELETE FROM [dbo].[TEMPTABLE] WHERE SUPPKEY > 0
```

Appendix C

Compiere Statistics

The Compiere ERP application is developed in Java [37]. The project contains 2021 files. Among these files 347 are of special interest since they contains functions utilizing select queries. The statistics of the analysed functions are presented in the following tables.

Compiere functions	Number of functions
With select	654
With insert, update, or delete	173
Selects without nested loops	639
Selects with nested loops	15

Table C.1: Compiere Functions

Functions containing	0	1	2	3	4	5	6
Number of selects		598	30	6	3	1	1
Number of loops	241	359	27	8	0	3	1

Table C.2: Function Details

Method invocations	Max	Min	Average	Total
Outside loops	230	4	15.55	9939
Inside loops	110	1	6.55	3004

Table C.3: Function Computations

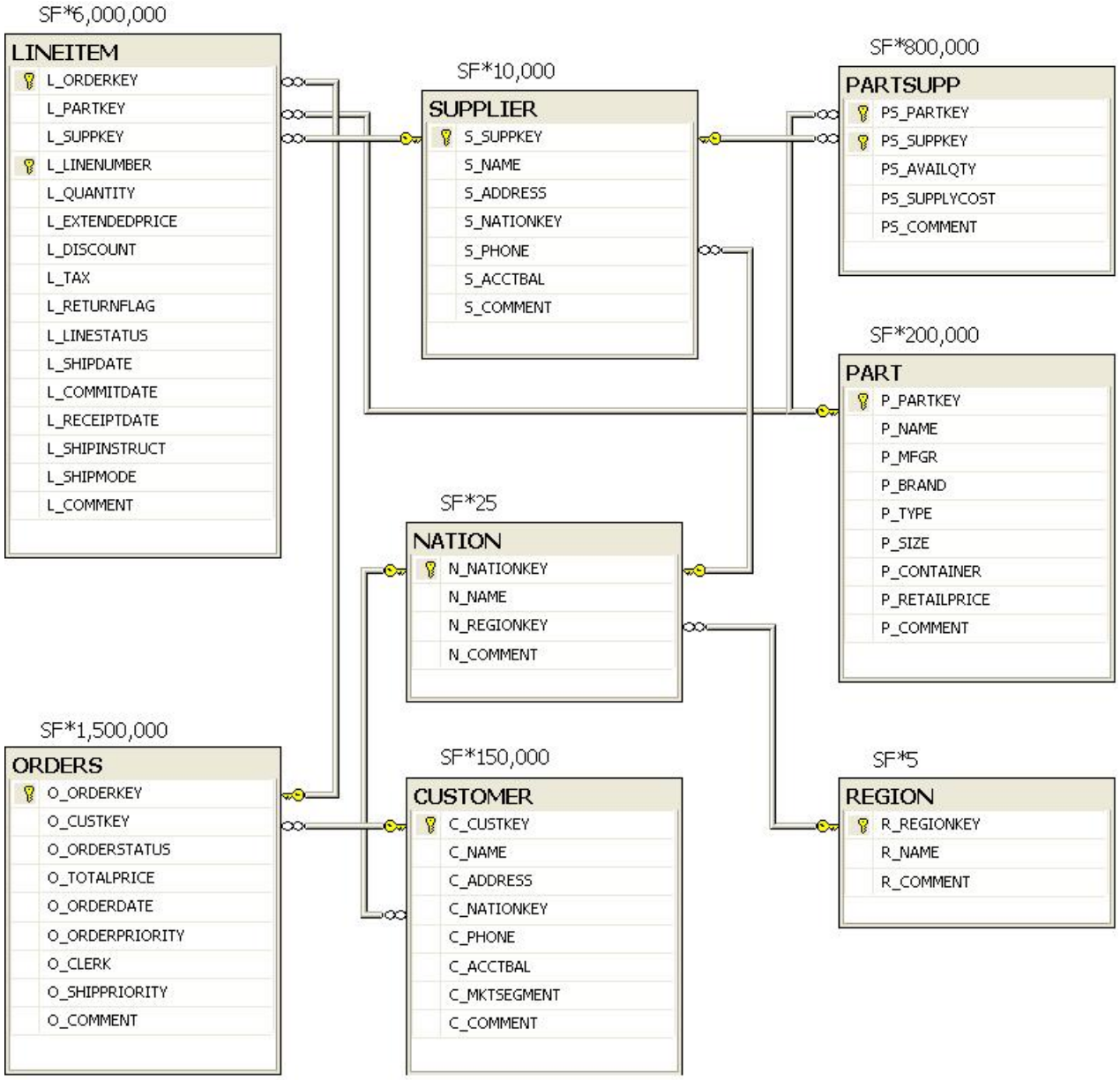
Returned Value	Number of Functions
void	144
Single value	295
Array	200

Table C.4: Returned Values by Functions

Appendix D

TPC-H Database Schema

TPC-H Database Schema



Appendix E

Experiments Execution Order

104,	3,	32,	14,	22,	44,	1,	7,	73,	4,	46,	86
38,	17,	39,	9,	77,	43,	2,	11,	96,	101,	67,	37
93,	95,	51,	62,	58,	106,	34,	42,	30,	92,	24,	79
45,	107,	97,	105,	54,	59,	19,	21,	49,	87,	10,	100
82,	0,	99,	68,	35,	61,	13,	28,	89,	56,	52,	5
31,	90,	65,	47,	81,	98,	69,	83,	76,	16,	36,	40
27,	103,	66,	84,	33,	94,	15,	78,	6,	41,	57,	55
23,	91,	63,	88,	25,	72,	85,	64,	29,	70,	74,	12
20,	102,	53,	26,	18,	48,	75,	50,	80,	71,	60,	8

Table E.1: Experiments Execution Order

Appendix F

Query Optimizations

Listing F.1: Query Optimizations

```
<Database>
  <Name>Test</Name>
  <Schema>
    <Name>dbo</Name>
    <Table>
      <Name>PART</Name>
      <Recommendation>
        <Create>
          <Index Clustered="true" IndexSizeInMB="28.695313">
            <Name>_dta_index_PART_c_6_2137058649__K1</Name>
            <Column Type="KeyColumn" SortOrder="Ascending">
              <Name>[P_PARTKEY]</Name>
            </Column>
            <FileGroup>[PRIMARY]</FileGroup>
          </Index>
        </Create>
      </Recommendation>
    </Table>
    <Table>
      <Name>LINEITEM</Name>
      <Recommendation>
        <Create>
          <Index IndexSizeInMB="282.062500">
            <Name>
                _dta_index_LINEITEM_6_2089058478__K2_K3_K15_K11_6_7
                </Name>
            <Column Type="KeyColumn" SortOrder="Ascending">
              <Name>[L_PARTKEY]</Name>
            </Column>
            <Column Type="KeyColumn" SortOrder="Ascending">
              <Name>[L_SUPPKEY]</Name>
            </Column>
            <Column Type="KeyColumn" SortOrder="Ascending">
              <Name>[L_SHIPMODE]</Name>
```

```
33              </Column>
34              <Column Type="KeyColumn" SortOrder="Ascending">
35                <Name>[L_SHIPDATE]</Name>
36              </Column>
37              <Column Type="IncludedColumn">
38                <Name>[L_EXTENDEDPRICE]</Name>
39              </Column>
40              <Column Type="IncludedColumn">
41                <Name>[L_DISCOUNT]</Name>
42              </Column>
43              <FileGroup>[PRIMARY]</FileGroup>
44            </Index>
45          </Create>
46          <Create>
47            <Statistics>
48              <Name>_dta_stat_2089058478_11_7_5</Name>
49              <Column>
50                <Name>[L_SHIPDATE]</Name>
51              </Column>
52              <Column>
53                <Name>[L_DISCOUNT]</Name>
54              </Column>
55              <Column>
56                <Name>[L_QUANTITY]</Name>
57              </Column>
58            </Statistics>
59          </Create>
60          <Create>
61            <Statistics>
62              <Name>_dta_stat_2089058478_2_11_3_15</Name>
63              <Column>
64                <Name>[L_PARTKEY]</Name>
65              </Column>
66              <Column>
67                <Name>[L_SHIPDATE]</Name>
68              </Column>
69              <Column>
70                <Name>[L_SUPPKEY]</Name>
71              </Column>
72              <Column>
73                <Name>[L_SHIPMODE]</Name>
74              </Column>
75            </Statistics>
76          </Create>
77        </Recommendation>
78      </Table>
79    </Schema>
80  </Database>
```

Appendix G

Software Performance Engineering

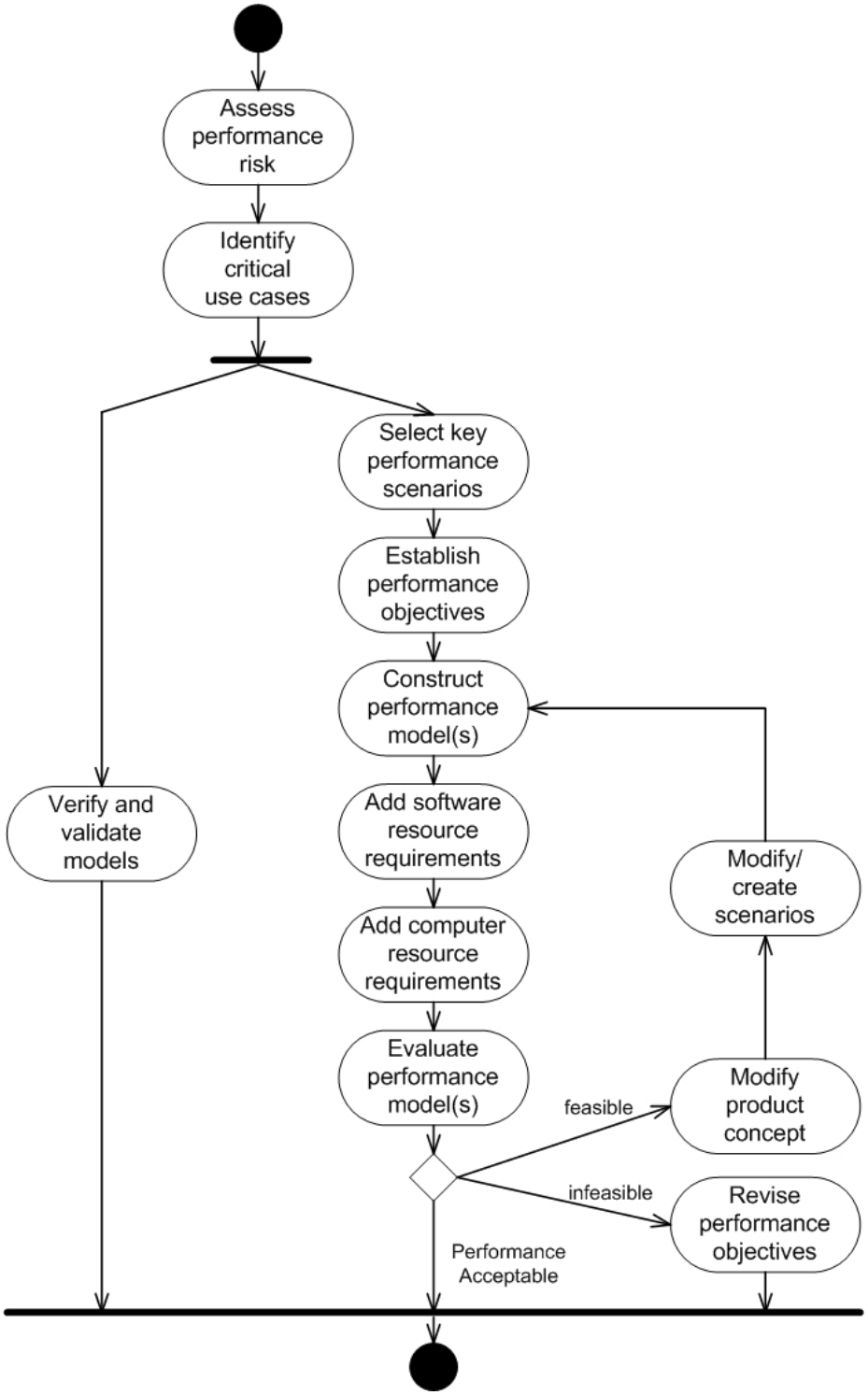


Figure G.1: The SPE Process

Appendix H

Weighted Score Calculations

	App	DB	P	D	W
Query1	0,28996	0,37876	30,62493	33,95197	0,42219
Query2	0,12684	0,14723	16,07890		
Query3	1,05129	1,01631	-3,32704		
Data1	0,48864	0,51317	5,01886	0,07320	0,00091
Data17	0,49008	0,51504	5,09207		
Number1	0,3813	0,40359	5,82226	1,25629	0,01562
Number6	0,59733	0,62461	4,56596		
Comp1	0,35299	0,38628	9,43072	6,84337	0,08509
Comp6	0,62573	0,64192	2,58735		
Scenario1	0,34367	0,34686	0,92767	6,36170	0,07910
Scenario2	0,63505	0,68135	7,28938		
Unit1	0,01582	0,01183	-25,18373	31,93031	0,39705
Units63	0,47785	0,51009	6,74658		
Units125	0,97442	1,02039	4,71728		
				Total 80,41686	

Table H.1: Calculating Weighted Score

App = Response time for code on the application server
DB = Response time for code on the database
P = The percentage that code on the database performs poorer than on the application server
D = Difference between best and worse
W = Weighted score

Appendix I

Experiment Deviations

ID	Response Time Mean	10%	20%	50%
0	0,128528593333333	3	1	0
1	0,164670266410256	5	1	0
2	0,726469002307692	0	0	0
3	0,726183122820512	0	0	0
4	0,112264178717948	5	3	0
5	0,161708398461538	6	0	0
6	0,247534579358974	71	7	4
7	0,254660048076923	3	0	0
8	0,211148230000000	3	0	0
9	0,861841812692307	0	0	0
10	0,242185457564102	4	0	0
11	0,906030307820512	0	0	0
12	0,112465181923076	14	4	1
13	0,232690177564102	3	0	0
14	0,135007614871794	9	2	0
15	0,263581122051282	2	0	0
16	0,266716701666666	25	6	1
17	0,896204351794871	2	0	0
18	0,469279256410256	9	2	0
19	1,623953180384615	26	0	0
20	0,123948725512820	26	6	2
21	0,280611949615384	31	12	0
22	0,342288407307692	11	6	0
23	0,767955639230769	34	9	0
24	0,003578351794871	43	35	2
25	0,004556935256410	75	41	9
26	0,016411887179487	22	3	1
27	0,017200853846153	27	3	2
28	0,003166926410256	72	53	49
29	0,002895247051282	52	46	9
30	0,009497393461538	32	9	3
31	0,009740712948717	42	21	2
32	0,003821299487179	35	31	4
33	0,010098250128205	8	4	1
34	0,004185151153846	43	13	3
35	0,011160066794871	15	3	2
36	0,002510661410256	63	52	8
37	0,003788807307692	41	22	4
38	0,004287567179487	71	62	4
39	0,004820090256410	74	22	3
40	0,004406200384615	72	51	6
41	0,010324235128205	48	4	3
42	0,007192032307692	41	6	4
43	0,014936188205128	7	4	2
44	0,002632371025641	68	55	9
45	0,003933532435897	53	30	5
46	0,006170762948717	71	21	5
47	0,007854640769230	9	1	1
48	1,333372838717948	44	16	0
49	2,279144960641025	24	1	0
50	1,439375962051282	34	7	0
51	2,389565580769230	36	2	0
52	1,399710241025641	41	16	0
53	1,512333773333333	35	12	0
54	2,067537136025641	34	5	0
55	2,169421905000000	30	3	0
56	1,354351862820512	39	19	0
57	1,472001996282051	35	10	0
58	2,092162168076923	26	6	0

Continuing Table I.1

ID	Response Time Mean	10%	20%	50%
59	2,587985623589743	30	0	0
60	0,021342420512820	48	26	2
61	0,035803171410256	30	5	2
62	0,026072389871794	30	8	2
63	0,042330551282051	31	7	2
64	0,016500098589743	38	18	3
65	0,016698079230769	41	12	1
66	0,026559603333333	18	2	1
67	0,027722975897435	19	2	1
68	0,015931630384615	42	15	4
69	0,019174116923076	57	27	5
70	0,026863806794871	17	1	0
71	0,030417986666666	14	3	1
72	0,069557655769230	5	2	2
73	0,084402261025641	9	1	1
74	0,375130129615384	3	1	0
75	0,372595407051282	0	0	0
76	0,056348455512820	4	2	1
77	0,082021816025641	10	1	0
78	0,100537723589743	8	2	0
79	0,120480006025641	6	1	0
80	0,110397325128205	8	3	0
81	0,432966673717948	1	0	0
82	0,126509976410256	7	0	0
83	0,459021736410256	2	0	0
84	0,060158165256410	38	8	0
85	0,120646482051282	9	5	0
86	0,070413916538461	22	7	0
87	0,137512771923076	5	2	0
88	0,126141518974358	19	3	0
89	0,434464861538461	2	0	0
90	0,243030505128205	2	1	1
91	0,740433097435897	25	0	0
92	0,063553704487179	19	6	3
93	0,126194254102564	26	5	0
94	0,165878040000000	4	1	0
95	0,351020791538461	28	7	0
96	0,665892309230769	48	14	0
97	1,104354141923076	27	7	0
98	0,728151232435897	43	16	0
99	1,174441641025641	28	3	0
100	0,695166639487179	44	15	0
101	0,745360409615384	40	17	0
102	1,026757024615384	33	6	0
103	1,079648082820512	31	8	1
104	0,668858923717948	46	17	0
105	0,715550723205128	43	14	0
106	1,023905122307692	34	8	0
107	1,253759313717948	26	7	1
108	0,157065697179487	1	0	0
109	0,169102873205128	6	1	0
110	0,896146347564102	0	0	0
111	0,924051481410256	1	1	1
112	0,111449986794871	2	1	0
113	0,160149834487179	5	1	0
114	0,235000494615384	52	3	3
115	0,258073414615384	2	0	0
116	0,238835034743589	0	0	0
117	1,035277413846153	1	0	0
118	0,272721440256410	3	0	0
119	1,078972843076923	0	0	0
120	0,110121345128205	7	5	0

Continuing Table I.1

ID	Response Time Mean	10%	20%	50%
121	0,236341726923076	5	1	0
122	0,139093881410256	15	3	0
123	0,267892080384615	3	0	0
124	0,292320535769230	11	3	0
125	1,045563393205128	0	0	0
126	0,495761676794871	5	3	0
127	1,870670036923076	8	2	0
128	0,130549897051282	40	10	2
129	0,287595077948717	37	16	0
130	0,346617474358974	15	4	1
131	0,843186724615384	29	8	0
132	0,003615643717948	34	26	5
133	0,004584259743589	71	36	3
134	0,031178432564102	78	76	9
135	0,018098540384615	26	4	2
136	0,003160436538461	67	54	44
137	0,003576837179487	63	42	26
138	0,008969153461538	41	7	1
139	0,009139200897435	42	8	0
140	0,003827767820512	38	21	5
141	0,011381635384615	6	2	0
142	0,005095196794871	72	33	4
143	0,012542748974358	2	2	1
144	0,003411675128205	63	50	5
145	0,004358485897435	76	31	4
146	0,004925321666666	64	58	5
147	0,005659106794871	76	69	6
148	0,004565270512820	73	58	3
149	0,011452680000000	5	3	1
150	0,007450767564102	37	9	4
151	0,016040571666666	5	2	0
152	0,003957633333333	73	66	18
153	0,004626368461538	70	65	5
154	0,005946798205128	76	6	2
155	0,008512792564102	34	4	2
156	1,230451095128205	35	15	0
157	2,175194888333333	22	2	0
158	1,333013001666666	45	11	0
159	2,265796863333333	23	5	0
160	1,263528805641025	39	15	0
161	1,400313859487179	32	18	0
162	1,991914614871794	39	10	0
163	2,102002994230769	32	5	0
164	1,264174374615384	39	16	0
165	1,349932345128205	34	18	0
166	1,978104551025641	25	4	0
167	2,472934956282051	30	2	0
168	0,019400435384615	40	20	4
169	0,032677401538461	19	4	1
170	0,024904225128205	22	5	1
171	0,038235696410256	13	1	1
172	0,015149515256410	46	18	1
173	0,016521444615384	38	8	3
174	0,025274645128205	11	1	1
175	0,027274340512820	16	3	2
176	0,015964860384615	47	15	2
177	0,015484592435897	30	7	1
178	0,025736593076923	8	1	0
179	0,030611933333333	25	3	1
180	0,082920068333333	4	1	0
181	0,087258991282051	7	2	0
182	0,455560001410256	1	0	0

Continuing Table I.1

ID	Response Time Mean	10%	20%	50%
183	0,456873171794871	1	0	0
184	0,057998039615384	13	6	2
185	0,084082256153846	17	5	0
186	0,121147133333333	74	33	2
187	0,125619542948717	25	3	1
188	0,125966020641025	7	3	0
189	0,520209533974358	2	0	0
190	0,140637513461538	4	1	0
191	0,549568414358974	3	0	0
192	0,061908071666666	50	9	5
193	0,120524991025641	6	6	0
194	0,071818367435897	28	7	0
195	0,137995135128205	11	4	0
196	0,139902296282051	17	4	0
197	0,519426558974358	2	0	0
198	0,249628249487179	6	1	0
199	0,872684402820512	18	0	0
200	0,064836265897435	15	3	1
201	0,126397045000000	26	4	0
202	0,184002866282051	36	1	1
203	0,390485225512820	38	10	0
204	0,622391024358974	45	16	0
205	1,057746846794871	23	5	0
206	0,680183637179487	29	16	0
207	1,116514583333333	27	1	0
208	0,619383036153846	37	18	0
209	0,704835143846153	40	16	1
210	0,955224670512820	39	9	0
211	1,016990939358974	28	6	0
212	0,606498419230769	41	19	0
213	0,673414007564102	42	14	0
214	0,978408776153846	29	11	2
215	1,186799403974358	36	6	0
216	0,175923128717948	2	2	0
217	0,176266595256410	5	2	0
218	1,039198345769230	0	0	0
219	1,038002356538461	0	0	0
220	0,074565052948717	6	4	2
221	0,109666900000000	8	3	1
222	0,684656662179487	46	28	1
223	0,514309883589743	74	67	10
224	0,337696769358974	1	0	0
225	1,289027083717948	0	0	0
226	0,401681893846153	1	0	0
227	1,364985352564102	0	0	0
228	0,209250486282051	4	2	0
229	0,525435234615384	1	1	0
230	0,259469292307692	0	0	0
231	0,571035963717948	0	0	0
232	0,326806476282051	1	0	0
233	1,183421740000000	1	0	0
234	0,593135860641025	1	1	1
235	1,750886563333333	0	0	0
236	0,215784848205128	2	0	0
237	0,563532012307692	1	1	1
238	0,439009176923076	1	0	0
239	1,104674335384615	1	1	1
240	0,002741365897435	17	8	3
241	0,003560421282051	76	16	5
242	0,015123040128205	8	2	0
243	0,015385563205128	8	2	0
244	0,001879830256410	76	41	8

Continuing Table I.1

ID	Response Time Mean	10%	20%	50%
245	0,002885216282051	77	69	11
246	0,008592281538461	16	8	0
247	0,009142173461538	39	6	0
248	0,003415012435897	7	6	2
249	0,010487177435897	2	0	0
250	0,004660329871794	45	10	4
251	0,035388552692307	78	78	78
252	0,002532230897435	51	8	6
253	0,004984812564102	9	7	2
254	0,003230538589743	5	5	3
255	0,005730003205128	6	5	2
256	0,003573515384615	4	4	2
257	0,010629840256410	3	2	1
258	0,006909803076923	63	3	1
259	0,015801148974358	13	3	0
260	0,002641915384615	37	10	5
261	0,005217606282051	11	5	1
262	0,004986995512820	11	8	3
263	0,010215429230769	18	4	2
264	2,633370022948717	59	18	0
265	4,951291611025641	34	0	0
266	2,944799752307692	60	15	0
267	5,281589438076923	18	0	0
268	2,729550232692307	62	23	0
269	3,034162190384615	60	9	0
270	4,909851318076923	30	1	1
271	5,191368979230769	25	0	0
272	2,728203162692307	62	18	0
273	3,050790269358974	57	14	0
274	5,130804139487179	23	0	0
275	5,831501357307692	18	0	0
276	0,029755384487179	43	27	1
277	0,052381802820512	15	1	0
278	0,033372051410256	40	6	0
279	0,057303127692307	8	0	0
280	0,024494532564102	55	11	0
281	0,029283568205128	42	24	1
282	0,045851861794871	17	0	0
283	0,048873736282051	14	1	1
284	0,024476243461538	52	17	1
285	0,026609728589743	47	11	0
286	0,047767050897435	17	1	1
287	0,052425107948717	16	2	2
288	0,089715293846153	2	2	1
289	0,101625715512820	61	2	1
290	0,521647960512820	0	0	0
291	0,523616377564102	1	0	0
292	0,041214036666666	4	1	1
293	0,059381191923076	13	5	0
294	0,186413221666666	0	0	0
295	0,186756255641025	1	0	0
296	0,173422947948717	1	1	0
297	0,654644602307692	0	0	0
298	0,205107418461538	5	1	0
299	0,690898581153846	0	0	0
300	0,104850344615384	4	2	0
301	0,264489422692307	0	0	0
302	0,131146114230769	0	0	0
303	0,287948052307692	4	0	0
304	0,165684587435897	0	0	0
305	0,597484345384615	0	0	0
306	0,298286270256410	1	0	0

Continuing Table I.1

ID	Response Time Mean	10%	20%	50%
307	0,895567991282051	1	1	1
308	0,109672554615384	4	2	1
309	0,280224660512820	1	0	0
310	0,223391128846153	1	1	0
311	0,545151852051282	0	0	0
312	1,323885615512820	64	21	0
313	2,493461230769230	24	0	0
314	1,481438560384615	58	13	0
315	2,663378012948717	19	0	0
316	1,389342538461538	62	20	0
317	1,533347146025641	59	14	0
318	2,457273873717948	28	0	0
319	2,607861999615384	21	0	0
320	1,370648481410256	63	19	0
321	1,534683969487179	58	14	0
322	2,560467996153846	29	0	0
323	2,938348672307692	17	0	0
324	0,201429137435897	0	0	0
325	0,202446509487179	1	0	0
326	1,206676769358974	0	0	0
327	1,204570601282051	0	0	0
328	0,074978060256410	6	5	3
329	0,109667694358974	6	3	2
330	0,694255040384615	52	34	2
331	0,570272378205128	65	53	6
332	0,368411636410256	1	0	0
333	1,449545875000000	0	0	0
334	0,437449951666666	0	0	0
335	1,541025594871794	0	0	0
336	0,211675095769230	2	0	0
337	0,527753269102564	0	0	0
338	0,263919587051282	1	1	0
339	0,584041023076923	1	0	0
340	0,355656031794871	0	0	0
341	1,344069300000000	0	0	0
342	0,621926657564102	1	1	0
343	1,898020085000000	0	0	0
344	0,220047970641025	3	1	0
345	0,567956031410256	0	0	0
346	0,453701152564102	2	1	1
347	1,086355488333333	0	0	0
348	0,002866201923076	6	5	1
349	0,003583792948717	33	13	3
350	0,016879071153846	9	3	1
351	0,016854019743589	15	3	0
352	0,002127891410256	77	68	11
353	0,003120210000000	74	73	13
354	0,009750832820512	54	4	2
355	0,009410733076923	54	9	1
356	0,003602583974358	4	4	3
357	0,011939249615384	2	2	1
358	0,004829013461538	22	9	3
359	0,012846883974358	2	1	0
360	0,002405382692307	9	6	2
361	0,005158875769230	13	5	2
362	0,003354922435897	7	6	4
363	0,006032754743589	10	8	3
364	0,004033182820512	38	7	6
365	0,011475583846153	2	1	1
366	0,006806322692307	10	8	1
367	0,017872578205128	28	2	1
368	0,003296492051282	77	71	3

Continuing Table I.1

ID	Response Time Mean	10%	20%	50%
369	0,005432721282051	11	5	4
370	0,005512959871794	41	7	3
371	0,010541199358974	12	3	0
372	2,772453694743589	40	3	0
373	5,108759932820512	4	0	0
374	3,080500273461538	32	2	0
375	5,419166970769230	3	0	0
376	2,926471899230769	37	3	0
377	3,186674219102564	33	2	0
378	5,064032630769230	7	0	0
379	5,395675251538461	4	0	0
380	2,846526055512820	37	3	0
381	3,174966833461538	32	1	0
382	5,258800605000000	2	0	0
383	5,974342876923076	0	0	0
384	0,029379227051282	27	5	2
385	0,058653698717948	37	8	4
386	0,038071549230769	33	5	2
387	0,063151822435897	24	7	5
388	0,026443506538461	33	7	1
389	0,029048637692307	30	9	1
390	0,052143112948717	31	6	2
391	0,051241568461538	6	3	0
392	0,026781636538461	31	9	3
393	0,028803920641025	27	2	1
394	0,051699396282051	24	7	4
395	0,054872437179487	13	5	2
396	0,103186759743589	6	0	0
397	0,103735701923076	5	1	0
398	0,607829788589743	0	0	0
399	0,607707038076923	0	0	0
400	0,042855312179487	17	6	1
401	0,060877482820512	16	5	2
402	0,194012309487179	3	3	0
403	0,199719593717948	1	1	1
404	0,187790606666666	1	0	0
405	0,738127335897435	0	0	0
406	0,221947637179487	1	0	0
407	0,780062690256410	0	0	0
408	0,107762120256410	2	1	1
409	0,267655709615384	1	0	0
410	0,135112592307692	4	4	0
411	0,296653198333333	2	1	1
412	0,181888172564102	3	1	0
413	0,678851794615384	1	0	0
414	0,315439818589743	0	0	0
415	0,968205619102564	0	0	0
416	0,112314265897435	3	2	1
417	0,287922076538461	2	0	0
418	0,237821742435897	1	1	1
419	0,555213277307692	0	0	0
420	1,389093475897435	41	3	0
421	2,573134705897435	4	0	0
422	1,559207850769230	32	1	0
423	2,748584986666666	3	1	1
424	1,495245159487179	38	4	0
425	1,617523952435897	29	2	0
426	2,537707405512820	10	0	0
427	2,703173536410256	5	0	0
428	1,442392223974358	37	4	0
429	1,606729535897435	25	2	0
430	2,657386690384615	9	2	0

Continuing Table I.1

ID	Response Time Mean	10%	20%	50%
431	3,006148163461538	2	0	0

Table I.1: Deviations in the Results of the Experiments

Appendix J

Replication Calculations

C	Response time	Std deviation	95% - 5%	90% - 10%	80% - 20%
0	0,13356549	0,00440274	1,66966632930571	0,294028453558379	0,0446450808941234
1	0,16763872	0,00399587	0,873064165193119	0,153746710850733	0,0233447962605889
2	0,74551049	0,01046617	0,302858792024168	0,053333471905404	0,00809811818810406
3	0,75367689	0,01316569	0,468909451431239	0,0825750142069203	0,0125381341311901
4	0,12886824	0,00895517	7,42041182608859	1,3067354690553	0,198413827019673
5	0,16627579	0,00393121	0,858947424804604	0,151260751067711	0,0229673298138237
6	0,23581656	0,00596019	0,981620780118559	0,17286354458561	0,0262474833243892
7	0,28119626	0,00160147	0,0498414500456543	0,00877708571036754	0,00133270673913269
8	0,21158276	0,0014682	0,0739915316574693	0,0130299181625757	0,00197845393318882
9	0,88329513	0,00802509	0,12684118343251	0,0223367485811874	0,00339159674939694
10	0,25504051	0,00568261	0,762868179627819	0,134341183737074	0,0203982741900287
11	0,93020556	0,00615657	0,0673119102280464	0,0118536359768618	0,00179984804420098
12	0,1198873	0,00321877	1,10765793967238	0,195058704459206	0,0296175813404913
13	0,23666816	0,01090247	3,26093593016362	0,574251233237519	0,0871939176333792
14	0,14520952	0,01112796	9,02429047852201	1,58917870432519	0,24129981622928
15	0,26777621	0,00616735	0,815127122847735	0,143543990303782	0,0217956220951427
16	0,22384074	0,00320768	0,315555635668088	0,0555693876905936	0,00843761812388977
17	0,86483897	0,01937495	0,771228043622058	0,135813356852768	0,020621807957067
18	0,44861409	0,00418066	0,13344921178026	0,023500423216087	0,00356828831642318
19	1,82886615	0,03410554	0,534389788746964	0,0941061099603031	0,014289007886697
20	0,12722474	0,0043638	1,80783448289337	0,318359883028656	0,0483395486363567
21	0,24487675	0,00892398	2,0407706761657	0,359379976375162	0,0545680117785323
22	0,31973999	0,00417367	0,261827003426157	0,0461077686996477	0,00700097231588028
23	0,76657573	0,07696185	15,4886298443702	2,72754969118412	0,414149294505196
24	0,00247005	0,000055145	0,765901512279781	0,134875354003989	0,0204793822409301
25	0,00327854	0,00093115	123,951120166633	21,8278080714359	3,31431956770526
26	0,02234111	0,00769059	182,088265108835	32,0657667112099	4,86884426125724
27	0,01605428	0,00177952	18,8797489264179	3,32472619407343	0,504824169528029
28	0,00152328	0,00041526	114,196515720015	20,1100209841767	3,05349194187027
29	0,0019279	0,00069536	199,904484409351	35,2032052025076	5,34523079306789
30	0,00823485	0,00180026	73,4397538336903	12,9327500174284	1,96369998795687
31	0,01026387	0,00210946	64,9071347062193	11,4301547007365	1,73554693456022
32	0,00285772	0,00002304	0,099884416057669	0,0175896584081147	0,00267080179835167
33	0,00976828	0,00021312	0,731449503373546	0,128808350841212	0,0195581725970706
34	0,00494169	0,00147338	136,599993551871	24,0552762879531	3,6525368303953
35	0,01110644	0,00049667	3,07296944002345	0,541150310348517	0,082167895960425
36	0,00189884	0,00012474	6,63142511787755	1,16779480910328	0,177317171481846

Continuing Table J.1

C	Response time	Std deviation	95% - 5%	90% - 10%	80% - 20%
37	0,00353779	0,00077525	73,7889930466646	12,994251087379	1,97303826869056
38	0,0032669	0,00107955	167,797689205431	29,549194471843	4,48672963989073
39	0,00391754	0,000028087	0,0789870626167811	0,0139096318016956	0,00211202905527941
40	0,0030678	0,0001068	1,86234678354079	0,327959506125767	0,0497971488942124
41	0,00944003	0,00027093	1,2657256438949	0,222894447334488	0,0338441416525132
42	0,0078158	0,00137821	47,7810899436492	8,41425603360341	1,27761492718767
43	0,01487573	0,00142114	14,0245613515272	2,46972704285105	0,375001678511352
44	0,00266384	0,00098808	211,417084521125	37,2305755506349	5,65306533118519
45	0,00604546	0,00279793	329,14489214592	57,9624574896633	8,80098021851622
46	0,00508994	0,00013302	1,04949577652156	0,184816340109703	0,0280623876869633
47	0,00864421	0,00235834	114,375854167851	20,1416025077155	3,05828725897678
48	0,70819559	0,20403556	127,54910824429	22,4614142308383	3,41052589705605
49	1,64739951	0,04105621	0,954403407618723	0,168070561816786	0,0255197200727424
50	0,69024983	0,11892592	45,6155123169398	8,03289753731823	1,21970971185811
51	1,73946307	0,06910785	2,42547315525427	0,427126132011852	0,0648545420855912
52	0,67978928	0,16512896	90,6713333231346	15,9672333634251	2,42445388035863
53	0,92755522	0,03898613	2,71464372227943	0,478049105748985	0,0725866518673142
54	1,36520889	0,07687789	4,87278028044791	0,858097228914973	0,130292901031518
55	1,51145914	0,04394957	1,29923757526945	0,228795902724028	0,0347402146348062
56	0,69851373	0,07397417	17,2338552933888	3,03488411536159	0,460814744950208
57	0,77416506	0,04232254	4,59249242906357	0,80873850253519	0,122798305507215
58	1,50016916	0,0741993	3,7591640960113	0,661989276792807	0,100516002637782
59	1,91036691	0,02898457	0,353730317305009	0,062291953996075	0,00945836803054713
60	0,01440909	0,00218746	35,4143493002807	6,23647142891582	0,946941590408205
61	0,02743216	0,00200492	8,20816768372395	1,44545937606395	0,219477570937642
62	0,01933337	0,0004953	1,00854124803801	0,177604242419959	0,0269673076670597
63	0,03252276	0,00171551	4,27546990115185	0,752910794933339	0,114321464263274
64	0,00944086	0,00083534	12,0302755556038	2,11853305981574	0,321676622407462
65	0,01051037	0,00020365	0,576905863563027	0,101593196158381	0,0154258419751302
66	0,02414897	0,00502224	66,4614535713791	11,7038704511461	1,77710774839011
67	0,02110734	0,00028922	0,288510594623731	0,0508067525130597	0,00771446282991417
68	0,00743903	0,00081509	18,4480575077294	3,24870528017971	0,49328120554153
69	0,00910181	0,00073681	10,0699572359962	1,77332075369226	0,269259820067379
70	0,02059955	0,00014327	0,0743305231282469	0,0130896146038184	0,0019875182003239
71	0,02373458	0,0002636	0,189539549445957	0,0333779388333959	0,00506808358599959
72	0,08033181	0,0093363	20,7561416477543	3,65515919163658	0,554996891683864
73	0,0894325	0,00620423	7,39532339642595	1,30231739339165	0,197742989408507
74	0,38232941	0,00355961	0,133199181802504	0,0234563929051125	0,00356160278387876
75	0,38342266	0,01579628	2,60811195032448	0,459288847119482	0,0697381069255826
76	0,06531547	0,0024769	2,20982034534103	0,389149644679396	0,0590882180155449
77	0,08650457	0,00519001	5,53134083443978	0,974069826473013	0,14790210155895
78	0,12554026	0,00412304	1,65745285032646	0,291877658352292	0,0443185056093155
79	0,14414607	0,00177357	0,232628644233251	0,0409659339214965	0,00622023961182857
80	0,1100787	0,00113796	0,164217540438457	0,0289187298173272	0,00439100031451048
81	0,44197805	0,00252695	0,0502300983602669	0,0088455267281433	0,00134309877683549
82	0,14025861	0,01507407	17,7490097590577	3,12560288247436	0,4745894209972
83	0,46914203	0,00148192	0,015332499782802	0,00270005516742742	0,000409974544672628
84	0,06354774	0,00302842	3,4898288239414	0,614559301027308	0,0933142672981223
85	0,1167424	0,0023788	0,638014808655865	0,112354489183738	0,017059829406714
86	0,08009957	0,0113328	30,7599916778616	5,41683845793975	0,82248907620467
87	0,13578243	0,00621888	3,22336387337488	0,567634789244667	0,0861892812666691
88	0,11905191	0,00526386	3,00405624313781	0,529014687587008	0,0803252312341672
89	0,43680356	0,00659526	0,3503189558086	0,0616912128014347	0,00936715189514742
90	0,22858827	0,00297713	0,260651273108772	0,0459007224407906	0,0069695345524134
91	0,76027977	0,12006959	38,3258080387939	6,74917968412753	1,02479086401419
92	0,06346589	0,00200198	1,52901185877664	0,269259183357589	0,0408841317124378
93	0,12088259	0,00474199	2,36464466514489	0,416414227144855	0,0632280537184635
94	0,16103443	0,00545752	1,76492122157397	0,310802852236679	0,0471920942082424
95	0,42569284	0,03719694	11,7325893773076	2,06611048584761	0,313716814344121
96	0,33187021	0,05563304	43,1818433826216	7,60432812756786	1,15463602345995
97	0,73363067	0,03900093	4,34277273177411	0,764762831990514	0,116121069528587
98	0,37199113	0,03977578	17,5688920754284	3,09388413541305	0,469773267964135

Continuing Table J.1

C	Response time	Std deviation	95% - 5%	90% - 10%	80% - 20%
99	0,85853249	0,02466973	1,26878330818696	0,223432902406329	0,0339259003052872
100	0,32612823	0,05378884	41,8003293979437	7,36104333865256	1,11769582617637
101	0,38400858	0,04079799	17,3447407284664	3,05441105462231	0,4637797021588
102	0,62052002	0,04738686	8,96141113111669	1,57810564322678	0,239618490145015
103	0,66287876	0,0553193	10,7018110517511	1,8845902912403	0,286154911153853
104	0,34279031	0,03352681	14,6993972553674	2,58856573159333	0,393046064379596
105	0,40697668	0,03189904	9,44035467705463	1,66244766275619	0,252424925165483
106	0,64619628	0,08526227	26,7520463948902	4,71103878239099	0,715320932344524
107	0,83422744	0,078202	13,5032687924869	2,37792735671266	0,361062876452899
108	0,152511	0,0046496	1,4282393089756	0,251513124483985	0,0381895821735217
109	0,17708638	0,01965114	18,9224476781128	3,33224544969318	0,505965888199622
110	0,88621831	0,01489952	0,434345425275887	0,076488284792286	0,0116139292666325
111	0,88338583	0,00350843	0,0242380071135792	0,00426831614428384	0,000648098227355434
112	0,11585439	0,00724698	6,01258556392507	1,05881708471863	0,160770067751139
113	0,162516	0,0073329	3,12846578780094	0,550923224270749	0,0836518085795917
114	0,21852936	0,01294155	5,38921035422073	0,94904063077755	0,144101685466509
115	1,21347285	1,67174583	2916,43635462918	513,584814044631	77,9823697044455
116	0,23189595	0,002854	0,232752172768111	0,0409876873122415	0,00622354263191991
117	1,0027714	0,01513555	0,350077879978275	0,061648759316965	0,00936070578686962
118	0,28594305	0,0204949	7,89413684784593	1,39015850581088	0,21108072431928
119	1,055826	0,02296037	0,726683323615045	0,127969025977809	0,0194307300792816
120	0,11511975	0,0173905	35,066837102173	6,17527448650352	0,937649488193587
121	0,22896271	0,00806902	1,90846502062012	0,336080933557864	0,0510303009252275
122	0,13564322	0,00516454	2,22761043304315	0,392282481392882	0,0595639058165349
123	0,26398001	0,01113872	2,73590256169183	0,481792790081095	0,0731550903562644
124	0,30433887	0,0304365	15,3690387774199	2,70648968904022	0,410951558068582
125	1,01600799	0,02983581	1,32511494800269	0,233352911363037	0,0354321477346869
126	0,48063213	0,01737986	2,00927361810761	0,35383334043365	0,0537258143404878
127	1,56613788	0,0728545	3,32525465544743	0,585577768963918	0,08891373113318
128	0,12400245	0,00423824	1,79507539546298	0,31611300610473	0,0479983843687022
129	0,25040669	0,01396336	4,77815406695187	0,841433540648651	0,127762697911238
130	0,32731041	0,01358573	2,64739324456386	0,466206288045405	0,0707884464623882
131	0,7622913	0,09752173	25,1497111483999	4,42886734110455	0,672476212148956
132	0,0026738	0,00013202	3,74622409614682	0,65971054116616	0,100170000966323
133	0,00459882	0,00058563	24,9187322446513	4,38819192669282	0,666300084822376
134	0,01747215	0,00069815	2,45344835062999	0,432052567485781	0,0656025687054352
135	0,02518579	0,01276791	394,912248311347	69,5440972990884	10,5595285491992
136	0,00205333	0,00075862	209,750596909036	36,9371068695839	5,60850524576879
137	0,00298939	0,00141433	343,960432385687	60,5714760156235	9,19713182131555
138	0,02625539	0,02762766	1701,46645807616	299,628460290994	45,4953821168777
139	0,01038354	0,00166051	39,2974386194101	6,92028395330651	1,05077122015448
140	0,003436	0,0007658	76,3302259112242	13,4417625189648	2,04098810083231
141	0,01043131	0,00016035	0,36310510803641	0,0639428558396389	0,00970903984636012
142	0,00496041	0,00058444	21,3312767140294	3,75644055016668	0,570375384445095
143	0,0117325	0,0000946866	0,10008478318722	0,0176249430851857	0,00267615939977797
144	0,00302431	0,00204034	699,398361764492	123,164140715702	18,7011595611305
145	0,00323561	0,00019866	5,79269494599293	1,02009431885351	0,154890429255032
146	0,00944208	0,00844494	1229,22085290991	216,465948985809	32,8680428248957
147	0,00479707	0,00114956	88,2436187829856	15,5397125232474	2,35953940605294
148	0,00400246	0,00112097	120,533040847185	21,2258838673016	3,22292380494664
149	0,01063468	0,00071611	6,96759274084266	1,22699397038596	0,186305931965208
150	0,01546686	0,01442747	1337,05165197626	235,454966455321	35,7513223536844
151	0,01551352	0,00017303	0,191158779839251	0,0336630855121895	0,00511138006423958
152	0,00194215	0,0000371556	0,562412744418477	0,0990409560284133	0,015038311565455
153	0,00420463	0,00131046	149,267102704008	26,2859558253471	3,99124169786844
154	0,00486226	0,0000725631	0,342237373246964	0,0602680450815165	0,00915105907415437
155	0,00741603	0,00031279	2,73359957461172	0,481387233762539	0,0730935109600192
156	0,40826874	0,04474832	18,4600701604336	3,25082070985313	0,493602410947854
157	1,67387745	0,01472178	0,118862665046278	0,0209317304757038	0,00317825975354539
158	0,48188462	0,05555474	20,4234093465914	3,59656499096145	0,546099988008696
159	1,789991	0,04821254	1,11478332447531	0,196313485631853	0,0298081064623947
160	0,40870911	0,03547485	11,5766961165778	2,03865766274875	0,309548396311147

Continuing Table J.1

C	Response time	Std deviation	95% - 5%	90% - 10%	80% - 20%
161	0,56234298	0,01510774	1,10909457832335	0,195311696708886	0,0296559955120311
162	1,49222578	0,05385088	2,00119157595251	0,352410091779264	0,0535097092304601
163	1,4720856	0,0578822	2,37572443319954	0,418365375712514	0,0635243147931499
164	0,43531222	0,04481762	16,2880088451459	2,86832043518234	0,435523828762911
165	0,49031157	0,02960006	5,60032245845752	0,986217499261214	0,149746595953085
166	1,49597719	0,01761383	0,213024616240726	0,037513662091499	0,00569604899947975
167	1,89377821	0,1249442	6,68876757263379	1,17789282269994	0,178850447015003
168	0,0123593	0,0021635	47,0866924420404	8,29197254499898	1,2590474852456
169	0,02801427	0,0056474	62,4468610365554	10,9969001936983	1,66976186416864
170	0,01715525	0,00086183	3,87812003081187	0,682937432084138	0,103696756324189
171	0,03131385	0,00073207	0,839855209085304	0,147898609444311	0,0224568245108925
172	0,01501541	0,01206869	992,699967387349	174,814590876806	26,5437288694868
173	0,00866758	0,00184051	69,287163005212	12,2014782428662	1,85266417786498
174	0,0288342	0,01319686	321,882629154121	56,6835720504985	8,60679512113926
175	0,0188104	0,00042362	0,779343619382319	0,1372425107793	0,0208388097196107
176	0,00727858	0,00104549	31,704330081813	5,58313657132693	0,847739668910442
177	0,00692453	0,00106545	36,3795798605159	6,40644865368938	0,97275081689066
178	0,018656	0,00127311	7,15594389291817	1,26016263229083	0,191342239084373
179	0,02322679	0,00237402	16,0532324394602	2,82697628019512	0,42924616032108
180	0,08287422	0,00181807	0,739527668731807	0,13023091679118	0,0197741740457227
181	0,0885105	0,00233314	1,06773663653928	0,188028557885465	0,0285501286545903
182	0,43975421	0,00577216	0,264745656763513	0,0466217439246332	0,00707901396531711
183	0,45174941	0,01539066	1,7835752380712	0,314087826921179	0,0476908825355276
184	0,06227887	0,00280378	3,11442984827252	0,548451493529497	0,0832765026608746
185	0,09519925	0,01288141	28,1340261785815	4,95440559857194	0,75227358459247
186	0,12426915	0,00985127	9,65672631170614	1,70055073521675	0,258210469835494
187	0,14044457	0,00904139	6,36843373688001	1,12148199336479	0,17028506496271
188	0,12537506	0,01052487	10,8288741585514	1,90696612055485	0,289552441895288
189	0,50251768	0,00512273	0,159688034241406	0,0281210831982833	0,00426988619306689
190	0,15048812	0,01397176	13,245548111261	2,33254271187624	0,354171702773781
191	0,52856019	0,00462768	0,117790394058949	0,020742903418066	0,00314958838123016
192	0,06337714	0,00275251	2,89844237820097	0,510416072500474	0,0775012301383273
193	0,12282334	0,01667492	28,3229082606273	4,98766775731231	0,757324088207699
194	0,07065954	0,00148317	0,677036974765294	0,119226297612925	0,0181032401361762
195	0,14425642	0,0109087	8,78714576144425	1,54741748939974	0,234958821689399
196	0,13649109	0,00858835	6,08391306702592	1,07137786711258	0,162677288427522
197	0,50231272	0,01176453	0,84289396691195	0,14843373508518	0,0225380776251268
198	0,23930425	0,00498984	0,668104736915988	0,117653329388346	0,0178644017081931
199	0,85141324	0,04228309	3,78987398952779	0,667397292958139	0,101337152143063
200	0,06905095	0,0006767	0,147579206559363	0,0259887170989823	0,00394610917132951
201	0,12410264	0,00046646	0,0217089545545635	0,00382294966605794	0,000580474083476521
202	0,16781258	0,00752292	3,08813297063295	0,543820610022974	0,0825733524512336
203	0,36406089	0,03188865	11,7895261166323	2,07613705290504	0,315239241483236
204	0,20294181	0,0066665	1,65815392031143	0,292001116866067	0,0443372514662822
205	0,73696266	0,03943195	4,3992402004162	0,77470676042087	0,117630948874636
206	0,30969886	0,01226973	2,41192018806632	0,42473945276136	0,0644921503275248
207	0,80604481	0,00808193	0,154484151282164	0,0272046787454002	0,00413073996270869
208	0,18862369	0,00436449	0,822709792015125	0,144879299311337	0,0219983745088613
209	0,26886294	0,03312274	23,3218371563665	4,10697849796027	0,623600828602992
210	0,60318849	0,0533581	12,0244924987307	2,11751466276276	0,321521989689905
211	0,69555982	0,01737319	0,958654907415489	0,168819251473279	0,0256334006022089
212	0,21292473	0,05866241	116,637909039478	20,5399506591363	3,11877217201489
213	0,24434046	0,01243714	3,9812770994475	0,701103405028661	0,106455065330727
214	0,65077712	0,02456224	2,1889887216929	0,385481193097864	0,058531202816398
215	0,87520658	0,00300816	0,018153185683437	0,00319677831431061	0,000485396650275685
MAX			2916,43635462918	513,584814044631	77,9823697044455
AVERAGE			66,052148963456	11,6317918574836	1,76616338363271

Table J.1: Calculating the Amount of Replications

Appendix K

Result Graphs

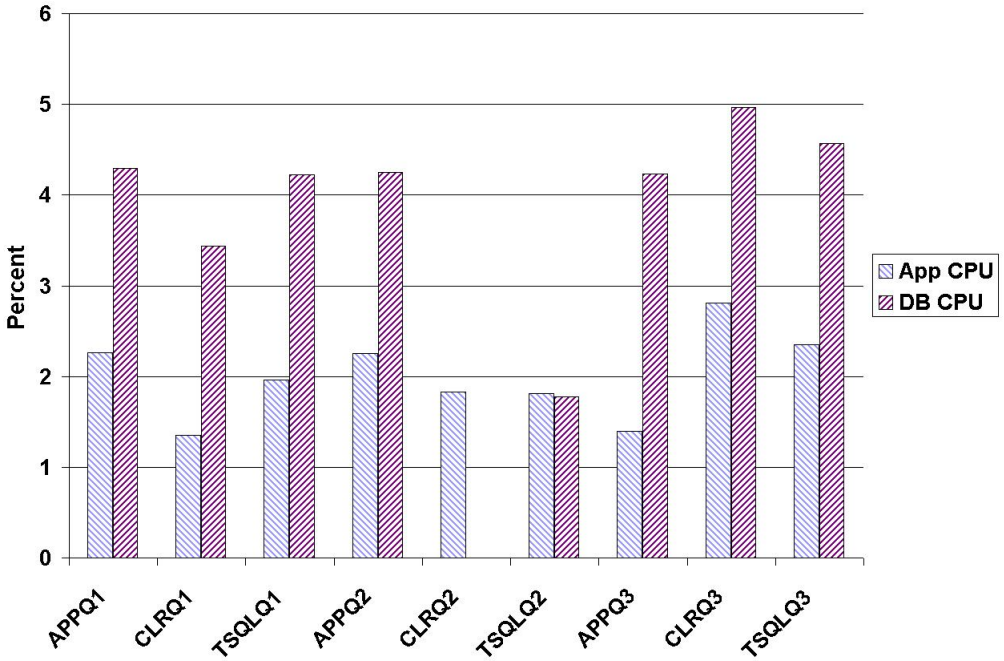


Figure K.1: CPU Utilization - 1 Workload Unit

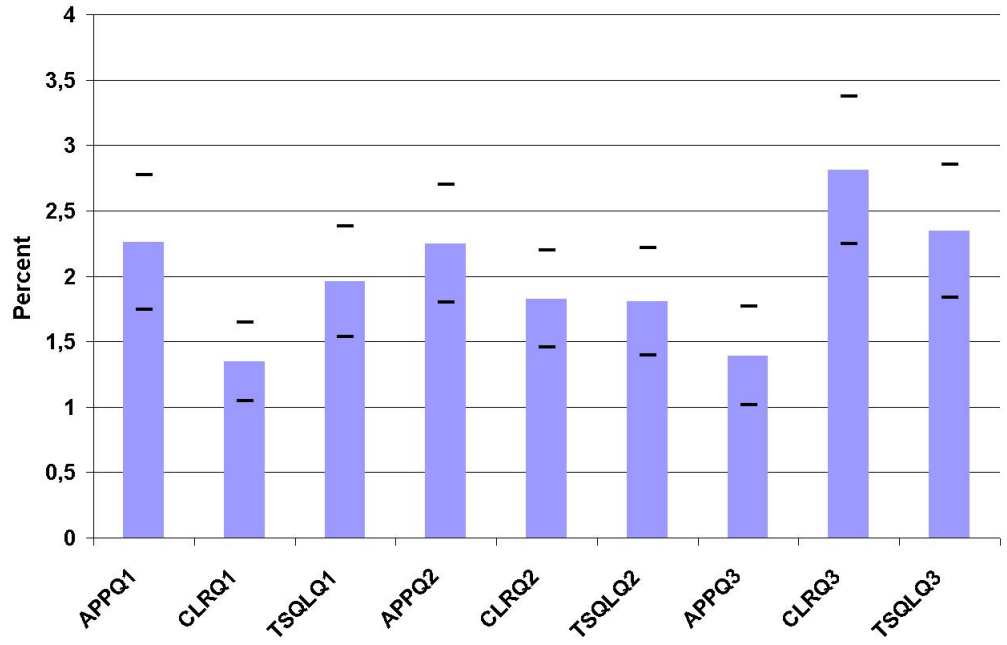


Figure K.2: Application CPU Utilization - 1 Workload Unit

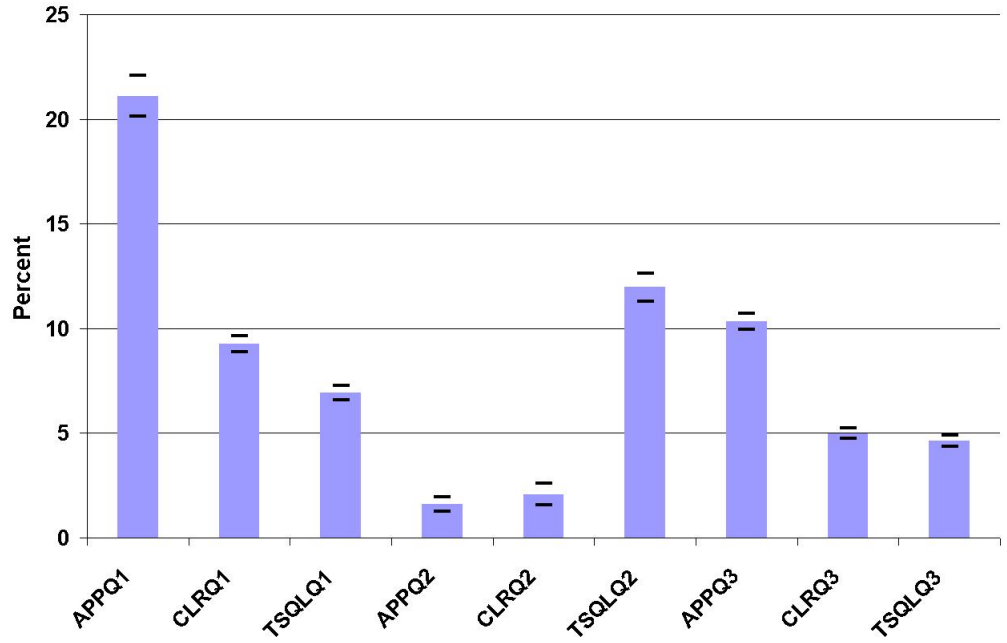


Figure K.3: Application CPU Utilization - 63 Workload Units

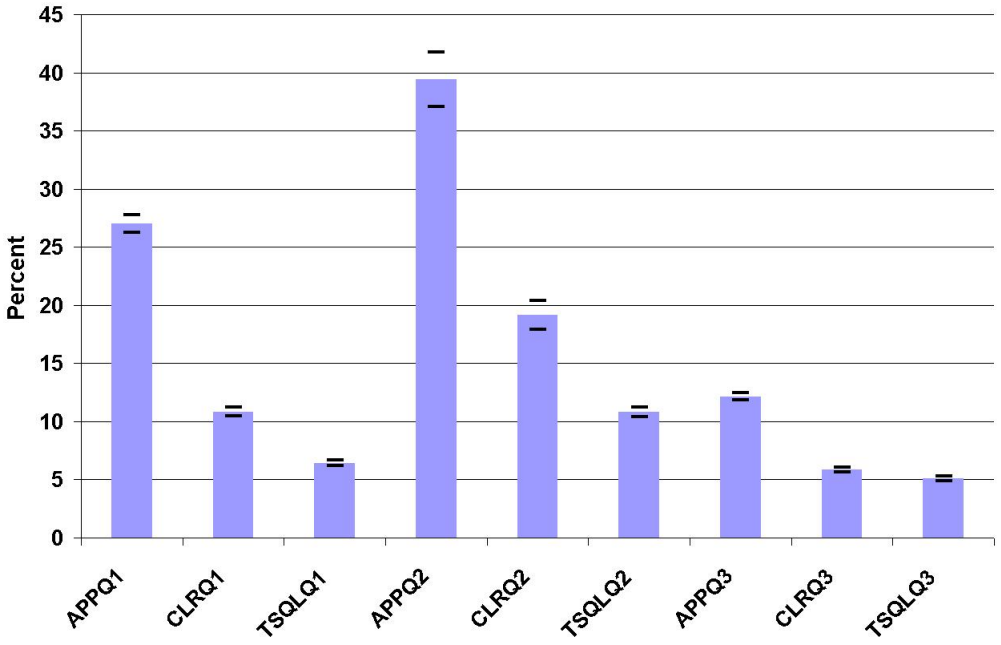


Figure K.4: Application CPU Utilization - 125 Workload Units

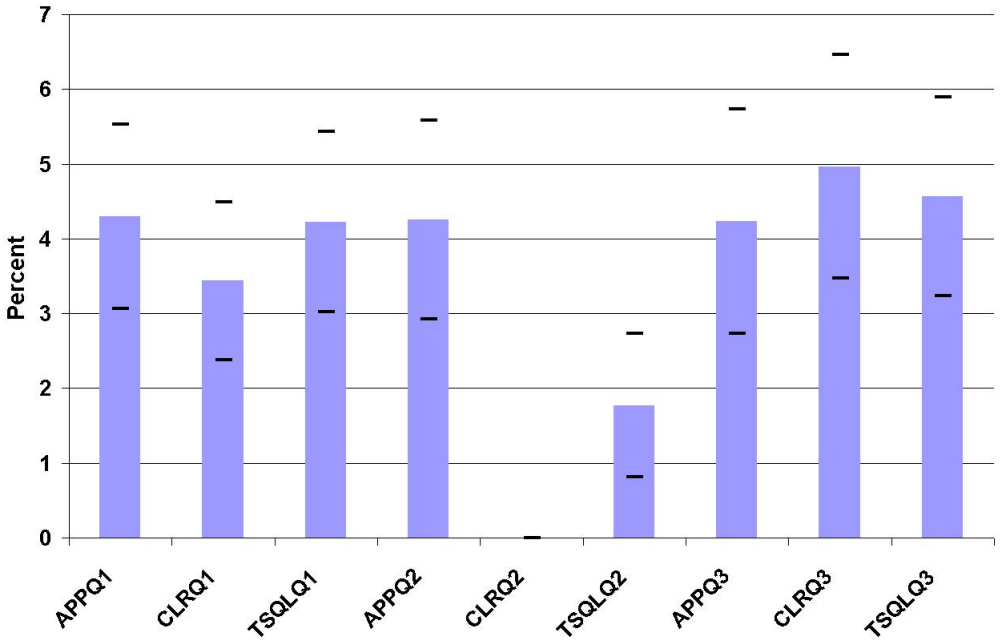


Figure K.5: Database CPU Utilization - 1 Workload Unit

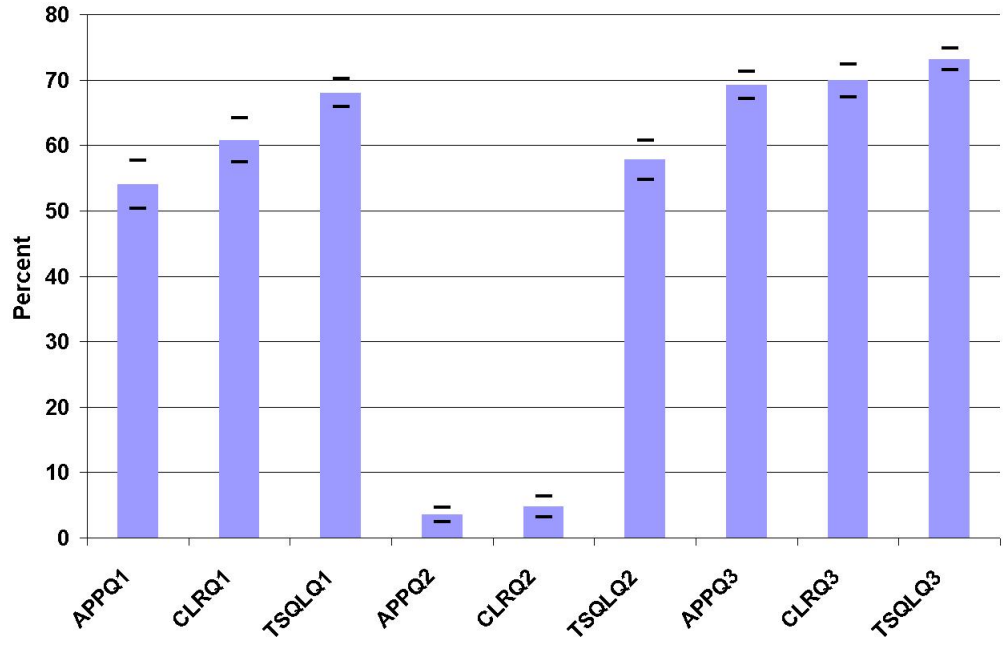


Figure K.6: Database CPU Utilization - 63 Workload Units

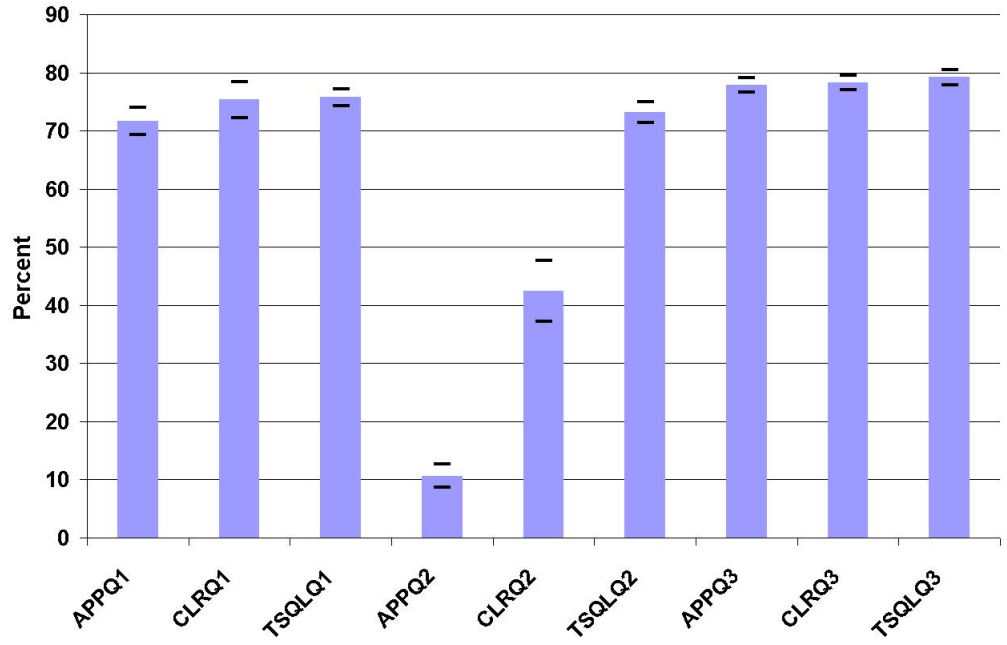


Figure K.7: Database CPU Utilization - 125 Workload Units

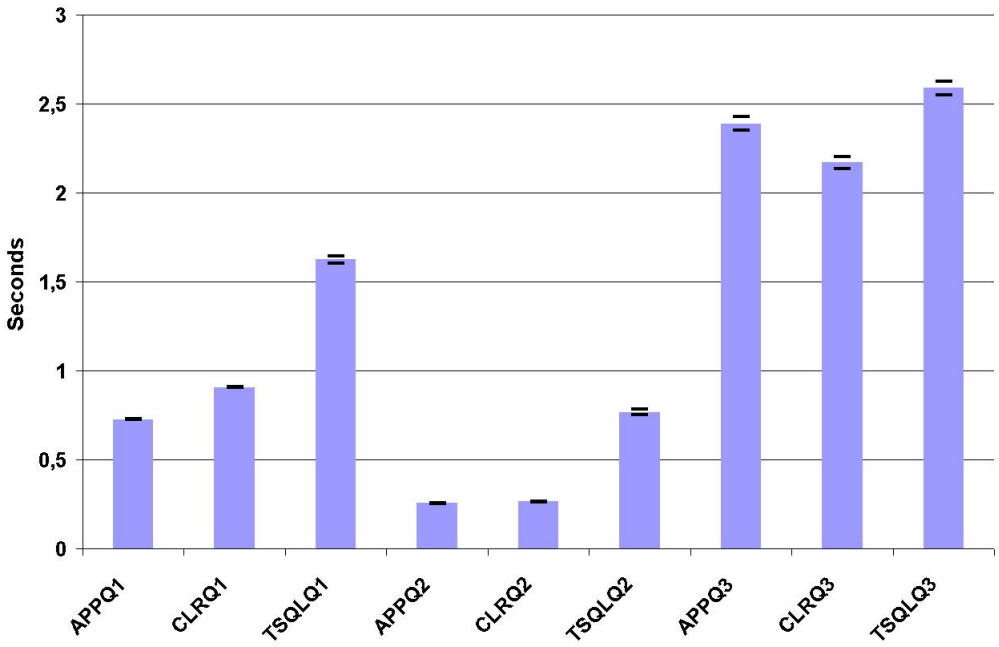


Figure K.8: Response Time - 125 Workload Units

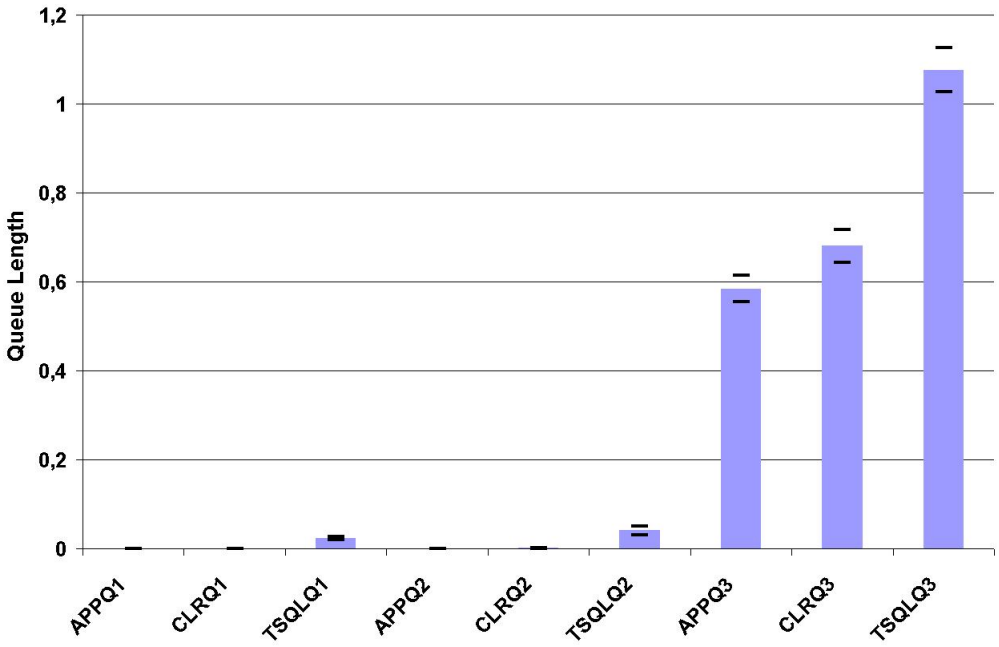


Figure K.9: Disk Queue Length - 125 Workload Units

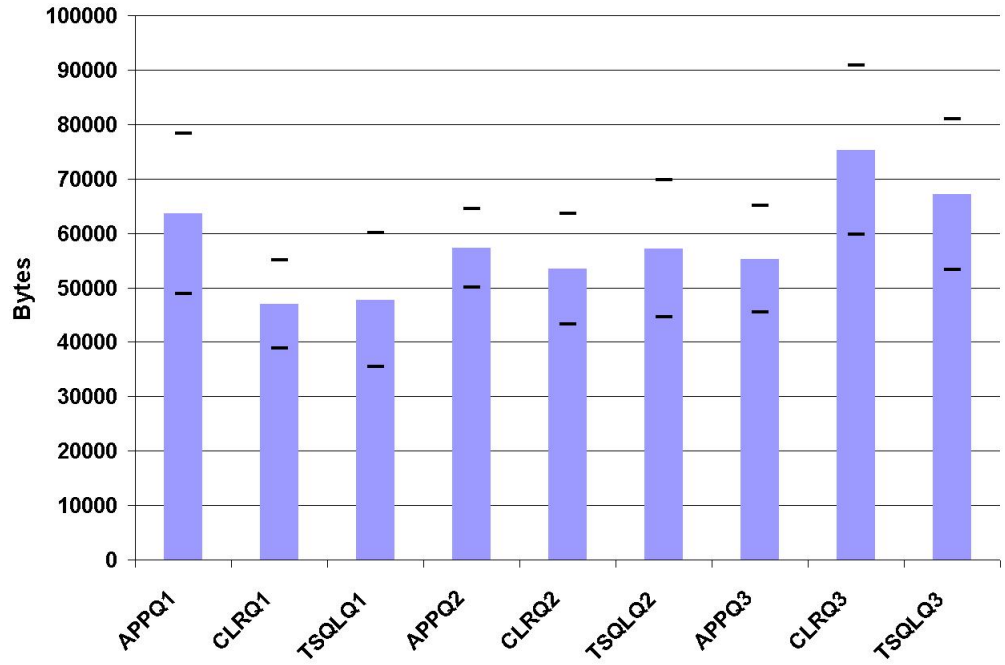


Figure K.10: Network Utilization - 1 Workload Unit

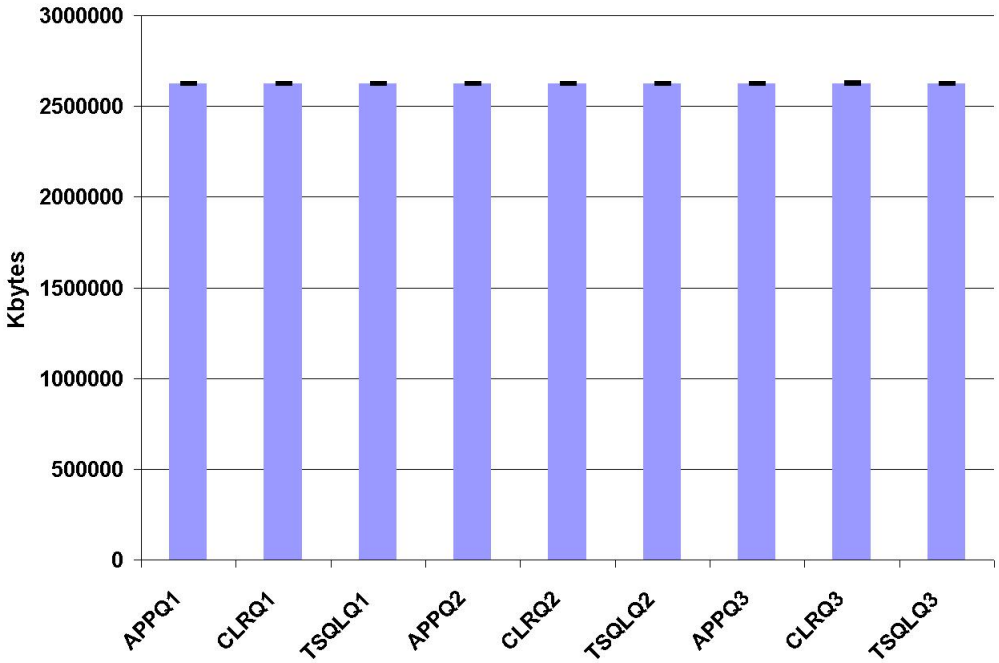


Figure K.11: Memory Utilization - 1 Workload Unit

Appendix L

Sample Metric Calculations

The example scenario considered has 2 *number of queries*, 4 *computations*, *Query 3*, *Scenario1*, 12,4 GB *data volume*, and 85 *WU*.

Preliminaries

$cloQue =$ Closest *number of queries* $= 1$
$cloCom =$ Closest *computations* $= 6$
$cloDat =$ Closest *data volume* $= 17$
$cloUse =$ Closest *WU* $= 63$

Application Server

$resQue1 =$ Response time for 1 *number of queries* and closest values for the other factors $=$ 1,058
$resQue6 =$ Response time for 6 *number of queries* and closest values for the other factors $=$ 1,117
$resCom1 =$ Response time for 1 *computations* and closest values for the other factors $=$ 0,622
$resCom6 =$ Response time for 6 *computations* and closest values for the other factors $=$ 1,058
$resDat1 =$ Response time for 1 GB *data volume* and closest values for the other factors $=$ 1,1004
$resDat17 =$ Response time for 17 GB *data volume* and closest values for the other factors $=$ 1,058
$resUse63 =$ Response time for 63 *WU* and closest values for the other factors $=$

1,058
$resUse125$ = Response time for 125 *WU* and closest values for the other factors = 2,175
$resQue$ = The calculated response time for 2 *number of queries* = $resQue1 + (\frac{resQue6 - resQue1}{5} \times (2-1))$ = 1,058
$resCom$ = The calculated response time for 4 *computations* = $resCom1 + (\frac{resCom6 - resCom1}{5} \times (4-1))$ = 0,884
$resDat$ = The calculated response time for 12,4 GB *data volume* = $resDat1 + (\frac{resDat17 - resDat1}{16} \times (12,4-1))$ = 1,071
$resUse$ = The calculated response time for 85 *WU* = $resUse63 + (\frac{resUse125 - resUse63}{62} \times (85-63))$ = 1,454
$resTqu$ = Response time for *Query 3* = 1,058
$resHar$ = Response time for *Scenario1* = 1,058

The final value for the application is calculated by multiplying all factors with their corresponding weights and then adding the factors together:
Final response time for the application = $resQue \times 0,01562 + resCom \times 0,08509 + resDat \times 0,00091 + resUse \times 0,39705 + resTqu \times 0,42219 + resHar \times 0,07910$ = 85,314

SQLCLR

$resQue1$ = Response time for 1 *number of queries* and closest values for the other factors = 0,955
$resQue6$ = Response time for 6 *number of queries* and closest values for the other factors = 1,017
$resCom1$ = Response time for 1 *computations* and closest values for the other factors = 0,619
$resCom6$ = Response time for 6 *computations* and closest values for the other factors = 0,955
$resDat1$ = Response time for 1 GB *data volume* and closest values for the other factors = 1,027
$resDat17$ = Response time for 17 GB *data volume* and closest values for the other factors = 0,955
$resUse63$ = Response time for 63 *WU* and closest values for the other factors = 0,955
$resUse125$ = Response time for 125 *WU* and closest values for the other factors = 1,992
$resQue$ = The calculated response time for 2 *number of queries* = $resQue1 + (\frac{resQue6 - resQue1}{5} \times (2-1))$ = 1,058

$resCom$ = The calculated response time for 4 *computations* = $resCom1 + (\frac{resCom6-resCom1}{5} \times (4-1))$ = 0,821
$resDat$ = The calculated response time for 12,4 GB *data volume* = $resDat1 + (\frac{resDat17-resDat1}{16} \times (12,4-1))$ = 0,976
$resUse$ = The calculated response time for 85 *WU* = $resUse63 + (\frac{resUse125-resUse63}{62} \times (85-63))$ = 1,323
$resTqu$ = Response time for *Query 3* = 0,955
$resHar$ = Response time for *Scenario1* = 0,955

The final value for the application is calculated by multiplying all factors with their corresponding weights and then adding the factors together:
Final response time for the application = $resQue \times 0,01562 + resCom \times 0,08509 + resDat \times 0,00091 + resUse \times 0,39705 + resTqu \times 0,42219 + resHar \times 0,07910$ = 77,427

T-SQL

$resQue1$ = Response time for 1 *number of queries* and closest values for the other factors = 0,978
$resQue6$ = Response time for 6 *number of queries* and closest values for the other factors = 1,187
$resCom1$ = Response time for 1 *computations* and closest values for the other factors = 0,606
$resCom6$ = Response time for 6 *computations* and closest values for the other factors = 0,978
$resDat1$ = Response time for 1 GB *data volume* and closest values for the other factors = 1,024
$resDat17$ = Response time for 17 GB *data volume* and closest values for the other factors = 0,978
$resUse63$ = Response time for 63 *WU* and closest values for the other factors = 0,978
$resUse125$ = Response time for 125 *WU* and closest values for the other factors = 1,978
$resQue$ = The calculated response time for 2 *number of queries* = $resQue1 + (\frac{resQue6-resQue1}{5} \times (2-1))$ = 0,978
$resCom$ = The calculated response time for 4 *computations* = $resCom1 + (\frac{resCom6-resCom1}{5} \times (4-1))$ = 0,830
$resDat$ = The calculated response time for 12,4 GB *data volume* = $resDat1 + (\frac{resDat17-resDat1}{16} \times (12,4-1))$ = 0,991
$resUse$ = The calculated response time for 85 *WU* = $resUse63 + (\frac{resUse125-resUse63}{62} \times$

$(85 - 63)) = 1{,}333$
$resTqu =$ Response time for *Query 3* $= 0{,}978$
$resHar =$ Response time for *Scenario1* $= 0{,}978$

The final value for the application is calculated by multiplying all factors with their corresponding weights and then adding the factors together:
Final response time for the application $= resQue \times 0,01562 + resCom \times 0,08509 + resDat \times 0,00091 + resUse \times 0,39705 + resTqu \times 0,42219 + resHar \times 0,07910 = 78{,}650$

Comparing Locality

As the results show, the lowest value, and thereby the best *locality*, is for SQL-CLR, the next best is T-SQL, and the worst is the application server.

Examining the amount in percent the three location deviate from each other, gives the following result:

SQLCLR better than T-SQL $= \frac{\texttt{T-SQL-SQLCLR}}{\texttt{SQLCLR}} \times 100 = \frac{78{,}650-77{,}427}{77{,}427} \times 100 = 1{,}58\%$

SQLCLR better than App $= \frac{\texttt{App-SQLCLR}}{\texttt{SQLCLR}} \times 100 = \frac{85{,}314-77{,}427}{77{,}427} \times 100 = 10{,}19\%$

Appendix M

Commonly Used Normal Quantiles

Table M.1 is originally presented in [7] on p. 630.

Confidence Level (%)	α	$\alpha/2$	$z_{1-\alpha/2}$
20	0.8	0.4	0.253
40	0.6	0.3	0.524
60	0.4	0.2	0.842
68.26	0.3174	0.1587	1.000
80	0.2	0.1	1.282
90	0.1	0.05	1.645
95	0.05	0.025	1.960
95.46	0.0454	0.0228	2.000
98	0.02	0.01	2.326
99	0.01	0.005	2.576
99.74	0.0026	0.0013	3.000
99.8	0.002	0.001	3.090
99.9	0.001	0.0005	3.29
99.98	0.0002	0.0001	3.72

Table M.1: Commonly Used Normal Quantiles

Appendix N

Summary

The objective of this master thesis was to examine the possibilities introduced by the integration of CLR in Microsoft SQL Server 2005. The focus of the examination was to determine whether the CLR integration posed an impact on the design of a C# application concerning the performance and scalability criteria. Subsequently, the obtained result of the examination was used to define a metric, which indicates whether a given functionality achieve the best performance and scalability on the application server or database.

As mentioned, the metric concerns the placement of functionality on either the application server or database. Hence, to provide a fair comparison of the application server and database it was necessary to consider the different implementation possibilities on the database. The possibilities were using XP, T-SQL, and SQLCLR. XP was excluded as the use of it was predicted to decline according to the references. Therefore, only T-SQL and SQLCLR had to be considered.

In order to develop the definition of the metric, an analysis of the parameters affecting the performance and scalability criteria was performed. The result of the analysis determined a large amount of parameters affecting the criteria. Among these parameters, only seven were used as factors in the full factorial experimental design. The seven parameters were *type of queries*, *number of queries*, *computations*, *workload units*, *hardware*, *data volume*, and *locality*. The parameters included in the experimental design were provided with either two or three levels. Each of the remaining parameters was set to a fixed level if possible, during the execution of the experiments.

The data obtained from the execution of the experiments was analysed in order to determine the impact of each parameter. Subsequently, the result of the analysis was used to define the metric. As the metric was defined, the accuracy of the metric was examined by using the characteristics of the performed experiments as

input.

In order to produce an output from the metric, several time consuming calculations must be performed, which are generally considered inappropriate. Therefore, the thesis provides a tool, which is able to perform these calculations automatically. The developer provides the values for the seven parameters examined in the thesis and the tool outputs a value indicating whether the functionality should be located on the application server or database.

Even though the results were obtained in a specific research context, the thesis discovered several scenarios, which benefit from the CLR integration in the database. The scenarios proved the hypotheses on design impact in a C# application by using the CLR integration in the database. The conclusions are however, to be cautiously drawn as the full factorial experimental design only contained seven parameters. One of the scenarios, which benefits from the CLR integration is functionality retrieving data from the database in order to perform additional insert and delete queries using this data. As the functionality requires access to the database several times, the amount of round trips are reduced, as both the SQLCLR and T-SQL in the database are able to access the data more efficiently.

Bibliography

[1] Erik Hejlskov, Morten K. Hansen, and Jacob E. Hansen. Java vs. .net - a comparison of persistence solutions. http://www.cs.aau.dk/library/files/rapbibfiles1/1136533201.pdf.

[2] Erik Frøkjær. Amanda og problemerne med statens it–projekter. http://erfa.teknologisk.dk/erfaswudv/Mat/0101/AmandaFinal.pdf.

[3] Balaji Rathakrishnan, Christian Kleinerman, Brad Richards, Ramachandran Venkatesh, Vineet Rao, and Isaac Kunen. Using clr integration in sql server 2005. http://msdn.microsoft.com/library/en-us/dnsql90/html/sqlclrguidance.asp?frame=true&_r=1.

[4] Transaction Processing Performance Council. Top ten tpc-h by performance. http://www.tpc.org/tpch/results/tpch_perf_results.asp.

[5] Andrew Binstock. 2005 survey spots trends in software development. http://www.infoworld.com/pdf/special_report/2005/49SRrrDevelop.pdf.

[6] Roger S. Pressman. *Software Engineering - A Practitioner's Approach*. McGraw-Hill, 2005.

[7] Raj Jain. *Art of Computer Systems Performance Analysis Techniques For Experimental Design Measurements Simulation And Modeling*. Wiley Computer Publishing, John Wiley & Sons, Inc., 1991.

[8] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems - Concepts and Design*. Pearson Education Ltd., 2001.

[9] Connie U. Smith and Lloyd G. Williams. *Performance Solutions A practical guide to creating responsive, scalable software*. Addison-Wesley, 2002.

[10] Transaction Processing Performance Council. Transaction processing performance council. http://www.tpc.org/.

[11] Christopher Koch. The abcs of erp. http://www.cio.com/research/erp/edit/erpbasics.html.

[12] ComPiere Inc. Smart open source erp software with integrated crm solutions by compiere erp & crm. http://www.compiere.com.

[13] William R. Cook and Siddhartha Rai. Safe query objects. http://www.cs.utexas.edu/~wcook/papers/SafeQuery/SafeQueryFinal.pdf, 2005.

[14] SAS Institute. Jmp 6. http://www.jmp.com.

[15] Sybase Inc. Sybase database systems. http://www.sybase.com.

[16] Inc. Scalability Experts. *Microsoft SQL 2005: Changing the Paradigm (SQL 2005 Public Beta Edition)*. Sams, 2005.

[17] Bob Beauchemin, Niels Berglund, and Dan Sullivan. *A First Look at SQL Server 2005 for Developers*. Addison-Wesley, 2004.

[18] Craig S. Mullins. Sql server update. http://www.craigsmullins.com/ssu_sql.htm.

[19] Microsoft. Clr hosted environment. http://msdn2.microsoft.com/en-us/library/ms131047.aspx.

[20] Mcgraw-Hill. Microsoft sql server 2005 developer's guide. http://books.mcgraw-hill.com/downloads/products/0072260998/0072260998_ch03.pdf.

[21] Microsoft Corporation. Sqlcontext class (microsoft.sqlserver.server). http://msdn2.microsoft.com/en-us/library/microsoft.sqlserver.server.sqlcontext.aspx.

[22] Glenn Johnson. *Programming Microsoft ADO.NET 2.0 Applications: Advanced Topics*. Microsoft Press, 2006.

[23] Oracle. Oracle® database performance tuning guide. http://www.stanford.edu/dept/itss/docs/oracle/10g/server.101/b10752/title.htm.

[24] Thomas Rizzo. *Pro SQL Server 2005*. Apress, 2006.

[25] the free encyclopedia Wikipedia. Sql. http://en.wikipedia.org/wiki/SQL.

[26] J.D. Meier, Srinath Vasireddy, Ashish Babbar, and Alex Mackman. *Improving .NET Application Performance and Scalability*. Microsoft Corporation, 2004.

[27] Exact Software. Additionele server aanbevelingen. http://www.exact.nl/docs/BDDocument.asp?Action=View&ID=%7B00716E63-DE66-423E-B992-D858D761426D%7D.

[28] European Commission. Europa - enterprise - sme definition. http://europa.eu.int/comm/enterprise/enterprise_policy/sme_definition/index_en.htm.

[29] Ye Wu Jerry Zeyu Gao, H S Jacob Tsao. *Testing and Quality Assurance for Component-Based Software*. Artech, 2003.

[30] Andrew D. Birrell. An introduction to programming with c# threads. http://birrell.org/andrew/papers/ThreadsCSharp.pdf, 2005.

[31] Transaction Processing Performance Council. Tpc-h. http://www.tpc.org/tpch/.

[32] Brad M. McGehee. How to perform a sql server performance audit article series. http://www.sql-server-performance.com/articles_audit.asp.

[33] Microsoft. How to use performancecounter to time code. http://support.microsoft.com/kb/172338/en-us.

[34] Oracle. Oracle toplink. http://www.oracle.com/technology/products/ias/toplink/index.html.

[35] Oracle. Oracle database 10*g*. http://www.oracle.com/technology/products/database/oracle10g/index.html.

[36] IBM. Db2 product family. http://www-306.ibm.com/software/data/db2/.

[37] ComPiere Inc. Sourceforge.net: Compiere erp + crm business solution. http://sourceforge.net/projects/compiere.

All links working June 13, 2006