Maintenance and Performance Issues in Three Different Multi-tier Designs

Andreas Börstad (anbo1767@student.uu.se)

Information Technology
Computing Science Department
Uppsala University
Box 337
S-751 05 Uppsala
Sweden

Abstract:
In a typical 3-tier application, the three layers; presentation, application and data are physically separated. However it is often possible to find additional layers within each physical module. These are often called logical layers. Some systems have all their application logic as stored procedures in the database, hence mixing application and data layers in one physical module, the database. Another approach is to separate these layers into separate modules: the data layer in stored procedures as before, but the application layer in a separate module. Using Traci, a time-reporting application, I compare these techniques with focus on maintenance and performance. I also look at a variant of the case with both application and data layers as stored procedures, where the layers will be logically separated. The latter turns out to be the most preferable design for both performance and maintenance reasons. This study also served as a basis to start using that design in Traci.

Supervisor: Emil Gustafsson (emil.gustafsson@ntier.se)
Examiner: Roland Bol (rolandb@csd.uu.se)
Passed:
Index

1 Background .......................................................................................................................................3
  1.1 Three-tier Solutions ..................................................................................................................3
  1.2 Traci..........................................................................................................................................4

2 Stored Procedures ................................................................................................................................6
  2.1 Performance Issues ...................................................................................................................6
    2.1.1 Recompilation During Run-Time .....................................................................................6
    2.1.2 Temporary Tables .............................................................................................................8
    2.1.3 The Table Data Type ........................................................................................................8
    2.1.4 Indices...............................................................................................................................9
  2.2 Error Handling ..........................................................................................................................9
    2.2.1 Fatal and Non-fatal Errors ..............................................................................................10
    2.2.2 Error Messages ...............................................................................................................10

3 Methodology ...................................................................................................................................12
  3.1 The Three Designs ..................................................................................................................12
    3.1.1 Application and data layer “classically” mixed in stored procedures.........................13
    3.1.2 Application and data layer separated, but both as stored procedures........................14
    3.1.3 Application and data layer separated into separate modules. .........................................14
    3.1.4 A Fourth Alternative? .....................................................................................................16
  3.2 Naming Conventions ..............................................................................................................16
  3.3 A Changing Environment .......................................................................................................17

4 Results ............................................................................................................................................18
  4.1 Testing ....................................................................................................................................18
    4.1.1 Expected Results .............................................................................................................18
    4.1.2 Time Measurements ........................................................................................................19
  4.2 Massive Data Updates and Computation ................................................................................21
  4.3 System Changes ......................................................................................................................22
    4.3.1 Different Database ...........................................................................................................22
    4.3.2 Different Data Model .......................................................................................................23
  4.4 Maintenance ............................................................................................................................23
    4.4.1 Bug Fixing ......................................................................................................................24
    4.4.2 New Functionality ..........................................................................................................24

5 Summary ........................................................................................................................................26

6 Glossary ..........................................................................................................................................28

7 Acknowledgements .....................................................................................................................28

8 Resources .......................................................................................................................................29
1 Background

1.1 Three-tier Solutions

The three-tier software architecture emerged in the 1990s to overcome the limitations of the two-tier architecture. The new tier is the application tier, which provides business logic and resides between the presentation tier and the data (management) tier. It is used when, for example, increased performance, maintainability and scalability is needed. In some applications, the middle tier is divided into several different units, and these applications are then referred to as multi layer applications.

Another advantage with the three-tier architecture is that the implementation process is simplified. Each tier can be developed concurrently by different teams, and because the programming of a tier can be changed or relocated without affecting the other tiers, the tree-tier model makes it easier to continually evolve an application as new needs and demands arise. The three-tier architecture allows different tiers to be developed in different languages, such as HTML for the presentation tier, Visual Basic for the middle tier, and SQL for the data tier.

A problem with three tier architectures is that separation of presentation logic, business logic, and data logic is not always obvious. A few things to look at when deciding where to place a certain function are; ease of development, ease of testing, ease of administration, scalability and performance.

Note that the boundaries between tiers are logical, and that it is quite possible to run all tiers on the same physical machine. The main importance is that the system is neatly structured and that the communication between the tiers is well defined.

Generally, a three-tier architecture looks like this:

![Three-tier architecture diagram]

The data layer contains the database and various functions for reading and writing data from and to the database. Calls from the application layer execute stored procedures, which sometimes result in a recordset being sent from the data layer to the application layer. Languages: SQL 92, T-SQL, Oracle SQL…

The application layer contains business logic, which is business rules, data manipulation and transformation of data into information. It also handles calls from
the presentation layer and communicates with the data layer. Languages: Visual Basic, C++, Java...

The presentation layer handles input from the client and sends information to the application layer. Output from the application layer is received and presented to the end user. Languages: Visual Basic, ASP, HTML, WML... (see glossary)

1.2 Traci

Traci is a time-reporting system developed and used at NTier Solutions. It makes it easy for employees to report for example worked time, absence and travel to and from work, and it also provides functionality to get a quick overview of everything reported. Furthermore, Traci handles a lot of administrative tasks, like generating salary specifications and invoice bases.

Traci is built using the three-tier architecture described in figure 2.

![Figure 2. The Architecture of Traci](image)

A brief description of the different layers and how data is sent and transformed between and within these layers is presented here.

The client asks for a specific HTML / WML page from ASP (Active Server Pages) which constitutes the presentation layer. ASP transforms the request to XML and sends it to the application layer. The application layer will transform the XML using XSL (Extensible Stylesheet Language) and create a dynamic stored procedure, which will be sent to the Database Interface, DBI. The DBI opens a connection to the database and sends the stored procedure there. It is executed in the database and the result will be returned to the DBI (via return parameters or a record set). The result is transformed to XML and passed on to the application layer, where the XML is modified using the Document Object Model (DOM) to a purer XML format with error handling codes. This XML is sent to the presentation layer, where it is converted into the format used by the current client (HTML or WML), and then passed on to the client.
In many three-tier systems, the application layer handles most of the business logic (rules for calculations, conditions etc). This is not the case with Traci. The application layer handles modifications and transformations of the request from the presentation layer and the result from the data layer, but the business logic mainly resides in stored procedures in the data layer. This architecture has many advantages compared to the traditional three-tier one, the most important being that changes in business rules only result in stored procedure changes, which are quickly deployed with no system downtime at all. A script is used to insert an altered stored procedure into the database while the system is running. No recompilation and deployment of DLLs (Dynamically Linked Libraries) has to be made. The main disadvantage with the design is the decrease in portability. Since we do not use the basic SQL-92 but a dialect called T-SQL, a change of database will result in the rewriting of a majority of the stored procedures in which the business logic resides. A database change with most business logic in the application layer would be far easier. This disadvantage could be reduced and perhaps eliminated by using SQL-92, but this would have meant that for example the CASE-statement could not be used. This will be discussed in more detail further on.
2 Stored Procedures

A stored procedure is a set of SQL statements with an assigned name that is stored in a database in compiled form. [1] Some of the most important advantages of using stored procedures are performance improvement, reduced network traffic and easy reuse of code.

When you want to update the code or modify it for use in another application, you only have to make the change in one place. That means more maintainable code and less time trying to track down problems.

Whenever a client application uses a stored procedure instead of a script or embedded T-SQL it not only reduces network traffic, but it will most likely speed up the T-SQL because the stored procedure code is already pre-compiled. The first time a stored procedure is executed, it is optimised and a query plan is compiled and cached in the buffer of SQL Server. Next time the stored procedure is called from the same connection, it will use the cached query plan instead of creating a new one, saving time and boosting performance.

2.1 Performance Issues

Because the code is already compiled, stored procedures usually perform better than scripts or embedded T-SQL. There are, however, quite a few situations where queries take more time to run than expected or acceptable. Some examples are table scans, recompilation during run-time and certain use of temporary tables. These things can severely damage performance, as I have experienced during my work with this thesis. However, there is often a fairly simple way of solving many of these performance problems.

2.1.1 Recompilation During Run-Time

As stated before, one of the main advantages with using stored procedures is that they are stored in the database in compiled form, and a query plan is compiled and cached in the buffer of SQL Server. However, there are situations where the stored procedure is recompiled during run-time, which severely hurts the server’s performance. Some examples:

SQL Server will automatically recompile a stored procedure if: [2]

- A sufficient percentage of data is changed in a table that is referenced by the stored procedure.
- The procedure interleaves Data Definition Language (DDL) and Data Manipulation Language (DML) operations.
- The procedure performs certain operations on temporary tables.
- High server activity causes the plan to be aged out of cache.
I noticed significant changes in performance when moving a statement creating a temporary table out from an if-else clause. The procedure’s execution time went down from 120ms to 70ms, an improvement of 40%!

The best way to prevent these recompilations is of course to avoid all situations where recompilations take place. However, this is not always possible. In certain situations you have to create a temporary table in an if-else clause, or you have to make a lot of changes to a table referenced by the procedure. What can be done in those situations? A very efficient way of reducing the effect of recompilations is breaking down large stored procedures into two or more sub-procedures and calling them from a controlling stored procedure. Even if recompilations do take place they will affect less code and therefore reduce the negative impact on server performance. Sometimes it is even possible to entirely avoid recompilations. An example:

```sql
create table t1 (a int)
select * from t1
create index idx_t1 on t1(a)
select * from t1
create table t2 (a int)
select * from t2
```

Because DDL and DML operations are interleaved, this procedure will be recompiled three times on every execution. The first recompilation will be at the first select statement, because the table `t1` does not exist at the initial compilation of the procedure so no plan could be made then. The second recompilation will be at the second select statement, because the schema of table `t1` has been changed since the original plan was generated, so the procedure must be recompiled again to create a new plan for the select from `t1`. The third recompilation will be at the last select statement for the same reason as the first one.

To avoid recompilations in this case, the procedure can be modified so that all DDL operations are performed before the DML operations.

```sql
create table t1 (a int)
create index idx_t1 on t1(a)
create table t2 (a int)
select * from t1
select * from t1
select * from t2
```

The first execution will result in one recompilation, because tables `t1` and `t2` did not exist at the initial compilation of the procedure so no plan could be made then. Following executions will result in no recompilations.

If a temporary table created by a stored procedure has changed more than six times, the procedure will be recompiled when the next statement references that table. This is important when changes to the table can seriously affect the optimal query plan for the statement, but in large procedures that frequently modify temporary tables, but only make small modifications, the recompilations will likely result in slower overall performance. In these situations, the KEEP PLAN option of the SELECT statement should be used.
create table #dummyInfo (name char(3), val int)
select * from #dummyInfo
insert #dummyInfo values ('kef', 1)
insert #dummyInfo values ('kaf', 2)
... 4 more insert statements ...
insert #dummyInfo values ('kof', 7)
select count(*) from #dummyInfo
option (KEEP PLAN)

If the KEEP PLAN option is not used, the procedure will be recompiled on every execution.

2.1.2 Temporary Tables

Temporary tables are used more and more as queries become more complex. They are usually pretty quick since they are only stored in memory and never make it to disk. However, there are some performance issues involved when using temporary tables. Improper use of temporary tables in a stored procedure can force recompilation every time the stored procedure is run. Here are some ways of preventing this from happening:

- Any references to temporary tables in a stored procedure should only refer to tables created by that stored procedure. In a few places in Traci this is not the case. Since we wanted to separate the application layer totally from the data layer, this was not possible (data access procedures insert information into temporary tables created by the calling procedure, because this is the only way to return record sets to a stored procedure). It is not really a big performance issue in these cases though, because the stored procedures that reference these temporary tables only consists of one insert statement, so the recompilation will only take place once, and it will be quite simple.

- Don’t declare any cursors that refer to a temporary table.

- Any statements in a stored procedure that refer to a temporary table should precede any DROP TABLE statement found in the stored procedure.

- The stored procedure should not create temporary tables inside a control-of-flow statement. This was the case in at least one place in Traci, but was removed with great performance gains. (See 2.1.1)

2.1.3 The Table Data Type

In SQL Server 2000, a new data type called “table” is introduced. The main purpose is temporary storage of a set of rows, just like temporary tables, and they can be used in functions, stored procedures and batches. When it is possible, table variables should be used instead of temporary tables because they are better in a number of ways [3]:

create table #dummyInfo (name char(3), val int)
select * from #dummyInfo
insert #dummyInfo values ('kef', 1)
insert #dummyInfo values ('kaf', 2)
... 4 more insert statements ...
insert #dummyInfo values ('kof', 7)
select count(*) from #dummyInfo
option (KEEP PLAN)

If the KEEP PLAN option is not used, the procedure will be recompiled on every execution.

2.1.2 Temporary Tables

Temporary tables are used more and more as queries become more complex. They are usually pretty quick since they are only stored in memory and never make it to disk. However, there are some performance issues involved when using temporary tables. Improper use of temporary tables in a stored procedure can force recompilation every time the stored procedure is run. Here are some ways of preventing this from happening:

- Any references to temporary tables in a stored procedure should only refer to tables created by that stored procedure. In a few places in Traci this is not the case. Since we wanted to separate the application layer totally from the data layer, this was not possible (data access procedures insert information into temporary tables created by the calling procedure, because this is the only way to return record sets to a stored procedure). It is not really a big performance issue in these cases though, because the stored procedures that reference these temporary tables only consists of one insert statement, so the recompilation will only take place once, and it will be quite simple.

- Don’t declare any cursors that refer to a temporary table.

- Any statements in a stored procedure that refer to a temporary table should precede any DROP TABLE statement found in the stored procedure.

- The stored procedure should not create temporary tables inside a control-of-flow statement. This was the case in at least one place in Traci, but was removed with great performance gains. (See 2.1.1)

2.1.3 The Table Data Type

In SQL Server 2000, a new data type called “table” is introduced. The main purpose is temporary storage of a set of rows, just like temporary tables, and they can be used in functions, stored procedures and batches. When it is possible, table variables should be used instead of temporary tables because they are better in a number of ways [3]:
A table variable behaves like a local variable. It has a well-defined scope, which is the stored procedure in which it is declared.

Within its scope, a table variable may be used like a regular table. It may be applied anywhere a table or table expression is used in select, insert, update and delete statements.

Table variables are cleaned up automatically at the end of the stored procedure in which they are defined.

Table variables used in stored procedures result in fewer recompilations of the stored procedures than when temporary tables are used.

Transactions involving table variables last only for the duration of an update on the table variable. Thus, table variables require less locking and logging resources.

Table variables are created and manipulated in memory instead of the tempdb database, making them much faster.

This data type is not used in Traci because I did not discover it until after the implementation and testing stages were finished.

2.1.4 Indices

Indices are used to speed up selection queries by avoiding time consuming full table scans. It will slow down insertions, but the benefits of using indices widely exceed the drawbacks.

Without special tools, the work of analyzing which indices to use in the tables of a database can be rather difficult and time consuming. Fortunately, SQL Server provides these tools and index analysis is made quickly and easily. SQL Server’s query optimizer will determine which indices would be most efficient for the specified queries. The decision of which queries to use can be done in different ways. Either pick the queries most frequently executed, or pick queries that are extremely complex and therefore time consuming.

2.2 Error Handling

It is important to implement appropriate error handling for several reasons. The most important is that it is of course better to handle the errors by automatically correcting them or displaying error messages to the user than doing nothing and maybe cause the system to crash. The error messages should be well designed and provide users with as much relevant information as seems fit. An error message like “This value has to be an integer” is probably sufficient for a simple error, but errors of more complicated sorts are likely to require more detailed messages. “Communication with the printer was lost. To solve this problem, make sure that the printer is switched on and that the
printer cable is securely connected in both ends. If this does not help, please restart this application” is far better than the shorter “Communication with the printer was lost”. Unfortunately, applications of today far too often introduce messages like “Error 0x000083d2 has occurred in module clswin.dll”, which informs the user that something went wrong, but not what it was or how to solve the problem. An introduction to error handling in stored procedures is presented here. [4]

2.2.1 Fatal and Non-fatal Errors

There are two types of errors in SQL Server: fatal and non-fatal. Fatal errors cause a procedure to abort processing immediately after the error and terminate the connection with the client application. Non-fatal errors within procedures do not affect the connection with the client application, but merely cause processing to continue on the following line of code. The best thing to do when a non-fatal error occurs in a stored procedure is to halt the processing of the code and either return processing to the calling application or jump to another code segment where the error is handled.

A fatal error arises for example when a select statement tries to reference a table that does not exist. The error message will look something like

Server:Msg 208, Level 21, State 1, Procedure exampleStoredProc, Line 12
Invalid object name 'myTable'.

A non-fatal error will arise when for example an insert of a null value into a column that does not allow null values is attempted. The error message will look like:

Server:Msg 515, Level 16, State 2, Procedure exampleStoredProc, Line 9
Cannot insert the value NULL into column 'name', table 'dbo.myTable';
column does not_allow nulls.INSERT fails.
The statement has been terminated.

There are actually 26 different severity levels of errors in SQL Server, but the user only has to worry about if they are fatal or non-fatal. Values ranging from 0 to 19 are non-fatal and above that they are fatal.

2.2.2 Error Messages

Instead of printing error messages using the print statement, the RAISERROR statement can be used to implement custom error messages in applications. It can be used either to produce an ad hoc error message or to retrieve a message from the sysmessages table. Storing custom messages in the sysmessages table and retrieving them with RAISERROR is probably the best way of handling error messages. It is also very easy to use. This stores an error message in the sysmessages table:

sp_addmessage @msgnum=50001, @severity=10, @msgtext='Access error'

@msgnum is the ID of the message. This must be greater than 50000 because SQL Server use the IDs below.
@severity is the level of severity for this error. This value must be between 0 and 25, but only members of the fixed-server role sysadmin can use values above 18.

@msgtext is the error message to be displayed.

Printing an error not in the sysmessages table can look like this:

RAISERROR('Access error', 10, 1)

where 1 is a state value not used by SQL Server.

If we have stored a message in sysmessages it will instead look like this:

RAISERROR(50001, 10, 1)

The @@ERROR system function is used to implement error handling code. When an SQL statement has executed @@ERROR will contain 0 if the statement executed successfully and something else if it did not. This makes error detection very easy. Just check the value of @@ERROR immediately after a statement executes and if differs from 0, appropriate error handling code can be executed.

Using @@ERROR in combination with the RAISERROR statement makes the error handling code very easy to read and understand.

insert t1 values (null)
if @@ERROR <> 0
  RAISERROR (50007, 10, 1)
3 Methodology

I have compared three different implementations of Traci, with focus on performance and maintenance issues. The designs were:

A. Application and data layer classically mixed in stored procedures.
B. Application and data layer separated, but both as stored procedures.
C. Application and data layer separated into separate modules.

Design A was in use when I started my work on this thesis, and designs B and C were implemented by me.

I started with an initial study of the existing structure of Traci. My main focus was on the stored procedures, since application logic together with data access parts were located there. I then turned to studying stored procedures in general, looking mostly at performance and maintenance issues. 3-tier and N-tier systems were looked into, especially the subdivision into different layers and the distinction and interaction between layers.

I examined how implementation of designs B and C could be made, where an interesting part was to analyze all information and try to predict the results of the planned testing. The idea was to get a general understanding of the pros and cons with each design in theory, and then, after testing, compare the expectations with the actual test results. Designs B and C were then implemented, in that order.

The art of performing and presenting time testing was studied, and I set up a test environment and performed tests on the three designs. These tests measured how much time the month summary function, described in section 4.1, took in each design. No tests on memory consumption were made, partly due to lack of time but mostly because the memory used would only be a small part of the physical memory installed in any standard computer today. Also, memory consumption differences would not be noticed by the end user since this would affect only the server running Traci. Initially I had planned to quantify maintenance issues and in a structured manner present and analyse the data of the maintenance tests. Unfortunately I could not find information about this anywhere, so this was not done. However, two Traci-developers familiar with both design A and B were interviewed and their views and opinions together with my results form the basis of the maintenance discussion.

The results of the time tests and of the maintenance examination were presented and some conclusions drawn from these tests.

3.1 The Three Designs

Recall figure 1 in 1.1 where the three-tier architecture is described as three different layers with specific functionality. When describing the three designs of this thesis it is necessary to get into more detail on how the application and data layers interact with
each other and the rest of the system. I will use simple code examples where a new welcome message is inserted into the table ‘TraciConstants’. These examples do not fully reflect the benefits and disadvantages of each design, but they do serve as a brief description of the main principle of each design.

Procedure and function headers, variable declarations and return statements are excluded in the examples.

The general idea was to separate all business logic from data access in the most straightforward manner. It sounds simple enough: the access parts merely read or write data from the database, and all logical operations and calculations are performed in business logic parts. However, a problem with this separation is that the select-statements themselves, which read data from the database, often contain lots of business logic. Conditions, case statements, comparisons and calculations are important parts of the vast majority of the more complex data access stored procedures.

3.1.1 Application and data layer “classically” mixed in stored procedures.

There is only one type of stored procedure, and it can contain both business logic and data access statements.

The code example describes the process of inserting a welcome message into the table TraciConstants. We first check if there already is a record holding a welcome message. If there is, that record is updated, otherwise a new record is created.

```sql
if exists( select *
    from TraciConstants TC
    where TC.Name = 'WelcomeMessage' )
begin
    update TraciConstants
    set StringValue = @Value
    where Name = 'WelcomeMessage'
end
else
begin
    insert TraciConstants( Name, Str, Num, PID, From, To )
    values( 'WelcomeMessage', @Value, null, null, null, null )
```

Figure 3. Design A.

The code example describes the process of inserting a welcome message into the table TraciConstants. We first check if there already is a record holding a welcome message. If there is, that record is updated, otherwise a new record is created.
3.1.2 Application and data layer separated, but both as stored procedures.

There are two different types of stored procedures. The business logic procedure contains business rules and has no direct connection to data stored in the database. To get data, the business logic procedure has to call one or more data access procedures, which will write data or read data and return to the calling procedure.

```sql
exec a_traciConstantsCount 'WelcomeMessage', @count OUTPUT
if @count > 0
   exec a_traciConstantsUpdate 'WelcomeMessage', @value
else
   exec a_traciConstantsInsert 'WelcomeMessage', @value
```

3.1.3 Application and data layer separated into separate modules.

There is only one type of stored procedure, and it contains data access statements to write to or read from the database. The business logic is placed in a Visual Basic module, and a Database Interface is used to handle all communication between these two modules in a uniform way.

```vb
... Business Logic (VB Module) Data Layer
... Database Interface (VB) Data Read / Write (Stored Procedures)
... Data
```
The creation and initialization of parameters are a vital part of the Visual Basic code example, so here these statements will be included. The explanation of parameter handling and DBI calls would call for an explanation of the DBI itself, which is not a part of this thesis. I think, however, that the code is rather self explanatory and not too hard to follow. Again, I use the example with the insertion of a welcome message.

```
Dim parameters(1) As SPparameter
Dim oldMsgParam As New SPparameter
Dim newMsgParam As New SPparameter

oldMsgParam.Init "@strVal", adVarChar, adParamOutput, 20
newMsgParam.Init "@strVal", adVarChar, adParamInput, len(msg), msg

Set parameters(0) = oldMsgParam
DBI.execSPWithParamOutput "welcomeMessageSelect", conn, parameters
Set parameters(0) = newMsgParam
if isNull( oldMsgParam.value ) then
    DBI.execSP "welcomeMessageInsert", conn, parameters
Else
    DBI.execSP "welcomeMessageUpdate", conn, parameters
End If
```

This code was actually not implemented in this design, because I considered it too extreme to extract a simple if statement from the data access stored procedures and put into the Visual Basic module. Since the calls to the database through the Database Interface are rather expensive, it would be very inefficient to proceed with the separation also in these cases.

The actual code from version c looks something like this:

```
Dim parameters(1) As SPparameter
Dim newMsgParam As New SPparameter

newMsgParam.Init "@strVal", adVarChar, adParamInput, len(msg), msg

Set parameters(0) = oldMsgParam
DBI.execSPWithParamOutput "newWelcomeMessageInsert", conn, parameters
```

This means that the Visual Basic module calls a stored procedure similar to the one in design a. Observe that this is an exception from the rule that only data read and write statements are to exist in the stored procedures in the data layer.
3.1.4 A Fourth Alternative?

A fourth design could be to have application and data layers in the same module by using for example Visual Basic and embedded SQL, and no stored procedures at all. However, this would be more difficult to maintain and update, and also slower because the SQL statements would have to be compiled on the arrival at SQL Server. Furthermore, the smallest change to an SQL statement would result in having to recompile the entire module.

3.2 Naming Conventions

The implementation of design B soon showed the need for using a good naming convention for the script files that create and alter the stored procedures. Having more than one hundred stored procedures means difficulties to get a quick overview when trying to identify which stored procedures belongs to which layer. By using an appropriate naming convention these difficulties are reduced, if not totally eliminated.

Design B, which is used in the current version of Traci, uses three separate prefixes to separate the four different types of stored procedures in the system:

**Application layer**
- Stored procedures called from the presentation layer have no prefix.
- Stored procedures used internally, i.e. they are called from application layer stored procedures and do not produce recordsets, have the prefix i_ (“internal”).

**Data layer**
- Stored procedures returning values using output parameters, or affecting values in the database have the prefix a_ (“access”).
- Stored procedures that access data and insert it into temporary tables, but do not return anything have the prefix att_ (“access temporary table”).

One minor downside with this convention can be showed with an example:
- In my new stored procedure I want to start with an access check to be sure the user trying to execute the stored procedure has permission to do so. I think there is some stored procedure that deals with access, but should I look for a_accessLevelSelect or i_accessLevelSelect? That is not possible to know.

The change to this naming convention was not without grief. Every stored procedure, a total of about sixty, was to be opened and edited to comply with this new convention. At the same time we adapted a new naming convention for all variables and procedure names (PersonIDSelect instead of GetPersonID for example).

This shows the need for a naming convention strategy to be adapted already in the beginning of projects. If it is not, the need for it will first become apparent when the project has become somewhat large, which will mean a lot of precious time will be used changing to this convention. Also, there is always a chance of something going wrong when altering code.

Many of the coding techniques and naming conventions suggested by Microsoft in [5] are used in the Traci project.
3.3 A Changing Environment

It can be somewhat complicated to work in an environment that is constantly and rapidly changing. Functionality changes and data model alterations easily delay for example the work of extracting data layer parts to separate stored procedures. Triggers may replace parts of the functionality in some stored procedures that suddenly become faulty. Two developers might be modifying the same stored procedure concurrently, which inevitably will lead to confusion.

There are a couple of different approaches to avoid these issues. The most common one is to use a program that handles source code version control, like CVS (Concurrent Versions System) or Visual SourceSafe. These programs keep track of the different versions of source code files (or any other file), keep files from being concurrently altered by multiple users, and much more. However, there should be rather strict guidelines for using these kinds of version control programs. The services they provide are excellent, but all users must know how to use them in a correct manner (and practice this knowledge!). – Only keep files checked out when working with them, other users may want access to them. – Only check in files with code that has been thoroughly tested, or problems are bound to appear further down the line. Developers might start looking for bugs in their own code that simply is not there, but somewhere in the code checked in by another developer.

This approach was not used in my case, because I needed a system with no changes to the data model or in functionality, but only to the structure of the logical layers of the system. When evaluating different implementations of the same problem, there must be a separate database to work with. There is of course always a separate production database and a developer database, but in this case the need for an extra “design test database” arose and therefore a copy of the production database was made. This way I did not have to deal with changes, which were of no interest to this thesis.
4 Results

4.1 Testing

Testing has been concentrated to the month summary calculation, which is one of the more extensive calculations performed in Traci. It presents to the user reported time in a number of different tables: The number of external and internal hours reported for the current month and which projects these hours concern, any possible overtime, flex bank, vacation days left and salary information with an excerpt from a standard salary specification. Here is an example of what a month summary can look like for the user.

<table>
<thead>
<tr>
<th>KID</th>
<th>Projekt</th>
<th>Aktivitet</th>
<th>Aktivitetskategori</th>
<th>Summa</th>
<th>Attested</th>
</tr>
</thead>
<tbody>
<tr>
<td>170</td>
<td>Lodningsgruppen</td>
<td>Standard</td>
<td>Intern tid</td>
<td>1.50 h</td>
<td>Q/1</td>
</tr>
<tr>
<td>194</td>
<td>Lodningsgruppen</td>
<td>Mote</td>
<td>Intern tid</td>
<td>1.50 h</td>
<td>Q/1</td>
</tr>
<tr>
<td>180</td>
<td>Personalgrennen</td>
<td>Standard</td>
<td>Intern tid</td>
<td>2.50 h</td>
<td>Q/1</td>
</tr>
<tr>
<td>527</td>
<td>Project 16</td>
<td>Standard</td>
<td>Intern tid</td>
<td>108.00 h</td>
<td>1/14</td>
</tr>
<tr>
<td>348</td>
<td>Project 16</td>
<td>Restid t f arbeta</td>
<td>Intern tid</td>
<td>27.00 h</td>
<td>1/14</td>
</tr>
<tr>
<td>236</td>
<td>Semester</td>
<td>Standard</td>
<td>Frånvaro</td>
<td>8.00 h</td>
<td>Q/1</td>
</tr>
<tr>
<td>32</td>
<td>Övrig intern verksamhet</td>
<td>Restid inom arbeta</td>
<td>Intern tid</td>
<td>1.00 h</td>
<td>Q/1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Repor</th>
<th>Flex / Övertid som lön</th>
<th>Övrigt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extern tid</td>
<td>108.00 h</td>
<td>Kvar ett arbeta</td>
</tr>
<tr>
<td>Intern tid</td>
<td>5.50 h</td>
<td>Semester (30/11)</td>
</tr>
<tr>
<td>Frånvaro</td>
<td>8.00 h</td>
<td>Lön</td>
</tr>
<tr>
<td>Arbetad tid</td>
<td>114.50 h 0v 160.00 h</td>
<td>Loppskattad flex (30/11)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flex (30/11)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Löneart</th>
<th>Antal</th>
<th>A-pris</th>
<th>Belopp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Månadslönn</td>
<td>1.00</td>
<td>20000.00</td>
<td>20000.00</td>
</tr>
<tr>
<td>Semester</td>
<td>8.00</td>
<td>.00</td>
<td>.00</td>
</tr>
<tr>
<td>Restid</td>
<td>27.00</td>
<td>166.67</td>
<td>4500.99</td>
</tr>
</tbody>
</table>

Figure 6. Month Summary displayed to user.

4.1.1 Expected Results

My initial thought was that design A would turn out to be the fastest one. The expensive calls through the Database Interface would probably make C a lot slower. Design B would be slightly slower than A because the vast number of calls between “business layer” stored procedures and data access ones. These calls are not that expensive, but many small additions of time will add up and make design B slower than A. Another performance killer for design B would be the use of temporary tables. Since recordsets cannot be returned to a calling stored procedure, temporary tables have to be used, and they dramatically slow down performance [6]. However, after
studying SQL Server and stored procedures in more detail I was not so sure. Design B has a great advantage because of the smaller stored procedures, which will recompile faster and, above all, fewer times than larger ones. One stored procedure with 500 lines of code will in theory result in 400% more lines being recompiled than five stored procedures with 100 lines each. The questions now are: Are there more factors that will affect the result in a significant manner? Possibly. What will affect performance the most; recompilations or procedure calls? Hopefully the answer will emerge during measurements.

4.1.2 Time Measurements

I measured how long time the month summary calculations took with the three different designs. Because it is likely that these calculations will be executed with SQL Server sometimes having the stored procedure’s execution plans in the stored procedure cache and sometimes not, I have performed tests for both these cases. In the same way, the data cache will sometimes contain data of interest and sometimes not, which is why I have performed tests for both these cases as well. However, these cases have been combined into two cases: both plans and data in cache, or neither plans nor data in cache. When the data cache contains data from a month summary, a month summary has just taken place, which means that the stored procedure’s execution plans will certainly be in the procedure cache, hence the case with both plans and data in cache. In the same manner, when the data cache is empty, no month summary has been made recently and the procedure cache will therefore also be empty. The latter case is expected to occur more frequently then the former one. Execution times for the cases where the execution plans and data are not in cache are of course expected to be much longer than when plans and data are cached.

Only the time between the call to the procedure that calculates the month summary and the time this procedure finishes was measured. In designs A and B, this means the call to the stored procedure monthSummaryCalculation and in design C it means the call to the corresponding Visual Basic procedure. The reason for not including the XML generation, XSL transformation, DOM processing and these things is that they are performed in the same way in all three designs, thus taking the same amount of time.

However, it would be interesting to know if the total amount of time is much greater than the time for the month summary procedures to complete. If the time from the user’s click on the month summary button until the summary is displayed is much longer than the time for the month summary procedure, then a 50 percent difference in performance for the different designs would be rather uninteresting. If the month summary procedures would stand for a larger part of the total time, a 50 percent difference between designs would be significant. I have found that month summaries take about 1.5 - 2.3 seconds. The rather large variation partly depends on differences in presented data: processing much data through DOM, XSL and XML takes time, and if less data is processed it goes much faster. The variation also depends on
performance differences in the three designs, which will show in the following measurement data presentations. In other words, the performance differences between the three designs are indeed significant.

All values are in milliseconds.

There were more parameters to the procedure calls, but I only present those relevant.

I chose test cases where the month summaries were somewhat different from each other. One person that only worked half time and in very few projects, one that worked full time and had an average amount of projects, and one that worked some overtime and had more projects than average.

**Test 1**: Person ID = 2, Date = 200111
A person that worked full time and had an average amount of projects.

**Test 2**: Person ID = 9, Date = 200109
A person that only worked two weeks that month, had an empty flex bank and very few projects.

**Test 3**: Person ID = 4, Date = 200106
A person that worked some overtime and had more projects than average.

![Figure 7. Test values for tests with empty procedure and data cache.](image-url)
As expected, design C is the slowest of the three, although it is not so significant when the cache is empty because A and B will result in the creation of quite a few execution plans, while large portions of the code in design C is already compiled. It is interesting that design A is faster than design B when procedures and data are cached, but slower when the cache is empty. I believe most reasons for this have already been mentioned in the chapter ‘Expected Results’. When the cache is empty, SQL Server will first create execution plans for the called stored procedures, and then recompile the stored procedures at run-time when necessary. As stated earlier, recompilations are far more expensive when carried out on large stored procedures, like the ones in design A. In the case of cached procedures and data, design B will lose the performance race because of the previously discussed use of temporary tables and large number of calls to data access stored procedures.

The reasons for the large differences between empty cache and cached data and procedures are rather obvious. Most significant is the fact that reading data from disk is a lot slower than retrieving it from cache memory. Another thing is that SQL Server will have to create execution plans for the called stored procedures when they are not already in procedure cache.

4.2 Massive Data Updates and Computation

The specification of this master thesis included the question “What design gives the best performance regarding massive data updates and computation?” The month summary contains the computation, but very few data updates. Actually, Traci does
not contain any massive data updates at all. Massive data updates would only be performed if all data in a table should be altered or something similar to that. If that were to be done, it would most likely only be done once, and therefore a rather uninteresting part of this thesis.

4.3 System Changes

Which design would be most desirable if we were to change the system, perhaps use a different database or a different data model?

4.3.1 Different Database

I have concentrated on studying a possible migration from SQL Server to Oracle because together they stand for about 78% of the Windows-based database market [7]. However, a migration to almost any other database would be undertaken in a similar fashion and have the same problems as a migration to an Oracle database. The only difference might be that there is an “Oracle Migration Workbench” which handles the migration of data from SQL Server (and some other databases) [8], but this is only interesting if a company using Traci suddenly decides to use Oracle instead of SQL Server.

Much of the work when changing the database would be in rewriting the stored procedures. SQL Server and Oracle use different dialects of SQL, T-SQL and Oracle SQL, and some T-SQL statements would have to be modified or completely rewritten when changing to Oracle. In recent years, however, the number of differences has decreased. An example is the case statement (used in more than 20% of Traci’s stored procedures), which did not exist in Oracle three years ago. Because the stored procedures will be the greatest concern when changing databases, designs A and B will be most affected. Design C would require less work, having much of the code outside stored procedures.

It would be possible to decrease the differences between these vast rewritings of design A and B, and the much lesser work with design C, by using only the features of SQL-92. Both T-SQL and Oracle SQL are built on this basic SQL, and less rewriting would have to be made. This sounds all good, but the problem with SQL-92 is that it is just that: basic. The loss of the case statement and other frequently used features of T-SQL would mean slower developing and, above all, less efficient code. Some code will inevitably have to be rewritten because of fundamental differences like different data types and such. There are documents and walkthroughs on how to migrate Oracle databases to SQL Server [9], but I have not found any documents providing information on how to go about a migration in the other direction. At “Oracle Professional” [10] such a document supposedly exists, but only for members.

The database interface would probably have to be modified, but this would be the case in all designs, so it is not taken into consideration here.
4.3.2 Different Data Model

Changes to the data model are quite common during the developing process. There is hopefully no need for a totally new model, but rather more or less extensive changes to the initial one. It is desirable that updates in the model only concern parts where code has not been written yet, but this is probably not the case in most development processes. Changing the data model by adding new tables and changing the relations between tables will almost exclusively affect data access parts, which means designs B and C are better than design A in this case. The separation of business logic and data access means that changes only have to be made to data access stored procedures, while updating the code of design A means changes in all stored procedures. The fact that the business logic parts of designs B and C are located in different places does not matter.

The data model updates in Traci were so insignificant that the differences in updating the three designs were actually quite small.

4.4 Maintenance

As briefly mentioned earlier, I wanted to quantify the maintenance discussion in some way, but no method for this was found. Instead I will discuss pros and cons of each design, mainly in terms of overview, both in general and in more detail on bug fixing and new functionality.

In design A it is highly depending on the level of complexity of the stored procedure if a change, in terms of either new or altered functionality or bug fixing, is easy or not. Only the stored procedure of interest has to be changed, but if this stored procedure contains more than 700 lines of code, this is not very easy. This was actually the case in Traci, where a stored procedure with 770 lines of code was to be altered. Needless to say, it was easier to perform the desired changes in the design B version, where these 770 lines of code was turned into a stored procedure with 200 lines of code and about ten calls to sub-procedures. Additionally, some of the sub-procedures were called several times with different parameters, which reduced the number of lines of code. Fewer lines of code means fewer bugs. One could of course argue that the stored procedure with 770 lines of code was bad programming to start with…

Design C produces rather lengthy code because of all parameter creation, initialization and procedure calls. An example is the access check, which is performed in 70 lines of code (access stored procedures not included), compared to design A where the same task takes 30 lines in total. Maybe this does not make it harder to get an overview, but again: more code => more bugs.

Another thing to consider in terms of overview is the number of procedures for each design. Design A uses 56 stored procedures, design B uses 125 and design C about 100 (including the business logic procedures in the Visual Basic module).
4.4.1 Bug Fixing

The task of locating bugs is similar in all designs. Visual Studio and SQL Server both provide the possibility to step through code, displaying values of variables or showing exactly where execution fails [11]. Visual Studio also has an add-in called T-SQL Debugger which means all debugging in design C can be made through Visual Studio. Because design C has never been a part of the release version of Traci, there is little on-hand experience of debugging in this design. The fact that Visual Studio has such a great debugger, including the add-in, and that Visual Basic code is perhaps a bit clearer than T-SQL code make design C a strong candidate when it comes to bug search and bug fixing.

There are some differences of opinion between the developers of Traci whether design A or B is more convenient when debugging. The main issue is which design provides the best general overview. As mentioned above, I feel that design B to a greater extent than A provides this overview, especially when looking at more complex stored procedures. Voices have been raised against this, claiming that the need to go through more, though smaller, stored procedures is worse than going through one stored procedure from start to end, regardless of its size. But getting a general understanding for what is performed in the code is much easier when looking at a stored procedure with 50 lines of code with a number of sub procedure calls like

exec a_UserNameSelect @userID

instead of a much larger stored procedure where select statements are interleaved with calculations and other business logic.

4.4.2 New Functionality

Only new functionality in terms of computations and data access is considered, not user interface updates or similar.

When a piece of data access code is to be altered in design B or C, this only has to be performed in one stored procedure. The same change in design A would mean an initial search for the code of interest and then changes to possibly many stored procedures. When separating application logic to separate stored procedures I discovered nine different data access code blocks that existed in more than one stored procedure.

It is of course always important to comment code, but especially when having 100+ stored procedures, where more than half serve as data access procedures. When new functionality is added through a new business logic stored procedure or Visual Basic code, it must be easy to find out if suiting data access procedures already exist. This will be very much easier if file headers are precise and properly updated than if they are incomplete and poorly updated. In Traci, a standard file header is used:
This makes it easier to see if the stored procedure provides the desired data, or if a new data access procedure has to be written. The problem is that this header (except the author's name) comes from a stored procedure that takes three parameters: @NTName, @PersonID and @GroupID, and has been changed due to a data model update. This might seem like details of little importance, but having inaccurate file headers will inevitably result in confusion or calls to procedures which don’t do what the header claims.
5 Summary

I will now present the pros and cons of each design and then answer the questions asked in the master thesis specification.

Design A

Pros: Performance is good.
- Little system down time when updating business rules.
- Little network traffic and data traffic between stored procedures.
- Easy to get an overview of stored procedures.

Cons: Not so easy to implement new functionality or find bugs.
- Some redundant code.
- Not very easy to overview because of the mix between logic and access.
- Bad portability due to logic in stored procedures.

Design B

Pros: Performance is good.
- Less redundant code segments.
- Easier to structure code.
- Better abstraction.
- Little system down time when updating business rules.
- Easy to implement new functionality and find bugs. (One developer disagrees)
- Rather easy to get a good overview.

Cons: Much data traffic between stored procedures.
- Bad portability due to logic in stored procedures.

Design C

Pros: Easier to write complex code in Visual Basic than in stored procedures.
- Better portability because of less code in stored procedures.
- Rather easy to implement new functionality and find bugs.

Cons: Performance is not so good.
- More system down time when updating business rules.
- More network traffic due to data being sent between stored procedures and the visual basic module.
This clearly shows that design B is the best of these three. Most important is perhaps the ease of finding bugs and implementing new functionality. The pros and cons should really be ranked, so important issues are of greater value than less important ones. If, for example, portability would be of great importance and performance did not matter as much, design C would perhaps be a better choice.

The performance results correspond quite well with the expectations of 4.1.1, with Designs A and B being faster than design C. They also imply that the temporary tables and smaller stored procedures in design B have about the same effect on performance, which was part of the discussion in that chapter.

Of course, the fact that design B is now used in Traci is also a good proof that that design is the most desirable.
6 Glossary


ASP       Active Server Pages – a specification for a dynamically created web page with an .asp extension that utilizes script code.

DLL       Dynamically Linked Library – Module that contains functions and data. A DLL is loaded at run time by its calling modules.

DOM       Document Object Model – an interface that allows programs and scripts to dynamically access and update the content, structure and style of documents.

HTML      HyperText Markup Language – a language that defines the structure and layout of web documents by using a variety of tags and attributes.

Query Plan A query plan is the optimized data access path that SQL Server uses to execute a stored procedure.

SQL       Structured Query Language – a standardized query language for requesting information from a database.

WML       Wireless Markup Language – an xml language used to specify content and user interface for wap devices (for example cell phones).

XML       eXtensible Markup Language – a system for organizing and tagging elements of web documents.

XSL       eXtensible Stylesheet Language – a specification for separating style from content when creating html or xml pages.

7 Acknowledgements

I would like to thank Emil Gustafsson and Roland Bol for valuable help and suggestions during my work with this thesis. I would also like to thank my family and my girlfriend for all their support.
8 Resources

[1] SearchDatabase.com Definitions: stored procedure
http://searchdatabase.techtarget.com/sDefinition/0,,sid13_gci334271,00.html

http://support.microsoft.com/?kbid=243586


http://www.sqlteam.com/item.asp?ItemID=2463

[5] Coding Techniques

[6] Eliminate the use of Temporary Tables for HUGE Performance Gains

http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2864350,00.html

[8] Oracle Migration Workbench
http://technet.oracle.com/tech/migration/workbench/content.html


[10] Oracle Professional Online
http://www.pinnaclepublishing.com/OP


All links worked as of 2002-12-16.
Appendix A - Time Testing

Presented here are the values of the performed time tests. Each test was performed 12 times and the highest and lowest values were discarded to eliminate extremes, so the resulting test value is the average of ten values. All values are milliseconds.

Tests with empty cache
These tests were performed with an empty cache for each execution.

Test 1

**Design A**
12 values: 1063, 1080, 1076, 1096, 1063, 1080, 1076, 1046, 1076, 1076, 1080, 1076
Average of 10 values: **1075**

**Design B**
12 values: 876, 876, 890, 890, 903, 890, 890, 873, 873, 890
Average of 10 values: **886**

**Design C**
12 values: 1151, 1162, 1151, 1172, 1142, 1152, 1152, 1162, 1152, 1172, 1151
Average of 10 values: **1158**

Test 2

**Design A**
12 values: 903, 950, 923, 936, 936, 940, 923, 903, 936, 923, 906, 920
Average of 10 values: **925**

**Design B**
12 values: 766, 780, 763, 783, 766, 763, 766, 750, 783, 766, 766, 766
Average of 10 values: **769**

**Design C**
12 values: 1112, 1072, 1101, 1112, 1101, 1102, 1102, 1091, 1112, 1102, 1102
Average of 10 values: **1104**

Test 3

**Design A**
12 values: 1140, 1140, 1123, 1140, 1110, 1123, 1110, 1106, 1126, 1140, 1126, 1136
Average of 10 values: **1127**

**Design B**
12 values: 890, 890, 890, 893, 893, 893, 893, 873, 873, 893, 873, 890
Average of 10 values: **888**

**Design C**
12 values: 1192, 1192, 1191, 1181, 1182, 1202, 1182, 1192, 1202, 1192, 1192, 1172
Average of 10 values: **1190**
Tests with cache
These tests were performed without emptying the cache between every execution.

Test 1

**Design A**
12 values: 140, 156, 143, 140, 140, 140, 156, 143, 140, 156, 143, 140
Average of 10 values: 144

**Design B**
12 values: 170, 170, 173, 173, 170, 173, 173, 173, 173, 173, 173, 173
Average of 10 values: 172

**Design C**
12 values: 451, 450, 431, 451, 430, 440, 431, 441, 441, 430, 441, 430
Average of 10 values: 439

Test 2

**Design A**
12 values: 93, 93, 93, 93, 96, 93, 93, 93, 96, 96, 96, 96
Average of 10 values: 94

**Design B**
12 values: 110, 106, 110, 106, 110, 110, 110, 93, 93, 106, 106, 96
Average of 10 values: 105

**Design C**
12 values: 401, 381, 371, 401, 410, 380, 381, 390, 400, 391, 380, 390
Average of 10 values: 390

Test 3

**Design A**
12 values: 156, 156, 153, 156, 156, 156, 156, 156, 153, 156, 156, 156
Average of 10 values: 156

**Design B**
12 values: 173, 173, 170, 173, 170, 170, 173, 173, 170, 170, 170, 170
Average of 10 values: 171

**Design C**
12 values: 441, 441, 440, 450, 440, 440, 441, 431, 451, 430, 441, 440
Average of 10 values: 440