Topicus Fincare

Efficient database auditing

And entity reversion

Dennis Windhouwer

Supervised by: Pim van den Broek,

Jasper Laagland and Johan te Winkel

9 April 2014

SUMMARY

Topicus wants their current auditing system revised because this solution's impact on the performance is big and requires much disk space. They also want to be capable of reverting entities in a database to previous versions, for which a complete and correct audit trail is required. Because of this, an investigation is executed in order to find an alternative method for auditing, which better matches Topicus' needs and which supports reverting entities to earlier versions.

We investigated existing audit solutions and compared them by a list of criteria, extracted from the ISO 25010 software quality standard. We then created prototypes of the most promising solutions. Both prototypes were put through several tests to validate their performance. The best solution was the Service Broker, in combination with Triggers. Triggers automatically run after an insert, update or delete action and gather information about the action. The Service Broker allows for reliable messaging between databases, making it possible to store the audit information in another database, which can be located on a different system. After this the prototypes were adapted to the following setups; in one test the audit and audited databases were present on the same system, and in the other these two databases were present on different systems. This allowed us to determine if running the audit database on a second, potentially dedicated, system would reduce the overhead of the audit solution on the audited database by a significant amount. It was found that running the audit database on another system lowered the performance overhead of the Service Broker audit solution by more than 50%.

After we finished our investigation into suitable audit solutions, we investigated if, and how, we can use an audit trail to revert entities to previous versions. We examined several issues and discussed possible approaches for each issue. Through this investigation it became clear that an audit trail can indeed be used to revert entities to previous versions, as long as a complete and correct audit trail exists. There are many different strategies which can be used to revert an entity to a previous version. Which strategy you want to use depends on your goals and is very much project dependent.

Finally we investigated which audit table designs exist and are compatible with Topicus' needs. The performance of reverting entities and auditing is affected by the used audit table design and the amount of disk space required to store the audit trail also depends on the used design. Topicus currently uses full copies of entities in their audit tables, thus their audit trail also contains information on columns which aren't modified by an action. This is redundant information, taking up unnecessary disk space. To resolve this problem, two designs were found which reduce the disk space consumed by the audit trail, as these designs only store data about changed columns. The first design stores all these changes in one column, using an XML format. The other design uses two tables, one table contains all the metadata about the action, while the other table contains one row for each changed column. The exact performance effects of these two designs are still unknown.

Because of these findings we advise Topicus to use the following setup as an auditing system:

- Install the audit database on a separate system from the audited database.
- Setup communication between databases based on Microsoft's Service Broker principle
- Use triggers to gather audit information.
- Save the information in audit tables which make it possible to revert entities to a previous version.
- Use one of the discussed audit table designs to reduce disk space usage.

Index

1. Introduction

Topicus [34], a Dutch IT company which operates in health, education and finance sectors, would like to have the option to revert specific entities in a database to an earlier version. This document describes a strategy for reverting entities in a database, to the version they had at an earlier time, by using the information stored in an audit trail. It includes an investigation into suitable audit solutions for the creation of this audit trail.

1.1 Motivation

Topicus has its own workflow system, which amongst other things is used for handling (healthcare) invoices. Currently this system defines the possible states for entities and which state transitions can occur. Each state transition has actions linked to it, for example the creation of invoices, or requesting extra information. Some of these state transitions happen automatically, while others require user input. The system stores the state history of each entity, thus it is known which states it has been in. During state transitions entities are modified, and the different versions are saved. In this way track and tracing is supported. Nevertheless the way it is implemented now requires much resources, and an easy to use rollback mechanism, with which a given entity can be reverted to an earlier version, is not available. Topicus has a system which needs to handle several million invoices in four days. Keeping track of all the changes slows this system down too much and consumes a lot of disk space. Because of this Topicus wants to investigate alternative audit solutions.

1.2 Problem statement

In order to roll back entities to an earlier version, we must first know how they were changed. We also need to know when each change was made, so we can revert an entity to the version it had at the specified time. Topicus also requires the system to keep track of who is responsible for a change, so this information must also be included in the audit trail. The auditing system must ensure the creation of a complete audit trail, as any holes in the trail will make it impossible to ensure that an entity is correctly rolled back without storing a lot of redundant data, which consumes unnecessary disk space. The auditing system must also require as few resources as possible, so that it negatively affects the performance of the databases, and the applications which make use of these databases, as little as possible. Topicus uses Microsoft SQL Server, so the auditing system needs to be either compatible with that, or with their applications, as the actions could also be intercepted there.

1.3 Research Questions

From these problems we can define a main research question:

MQ: Is there an alternative method for auditing, which better matches Topicus' needs and which can support the reverting of specific entities to earlier versions?

This question is split into several sub-questions, one relating to the auditing solution, another to the rolling back of entities and one to the tables used to store audit information. The sub-questions of Q1 have been determined together with Topicus, and they represent their needs.

Q1: Are there alternative auditing solutions which meet Topicus' needs, and are compatible with Microsoft SQL Server or Topicus' applications?

- Can these solutions gather all the needed audit information?
- Can these solutions guarantee a complete and correct audit trail?
- How much performance overhead do these solutions introduce?
- How fault tolerant are these solutions?

- Can the systems, on which these solutions are applied, continue working if the audit solution crashes?
- How modular and extendable are these solutions?
- How easy is it to install and remove these solutions?

Q2: Can we use the audit information to revert entities to an earlier version?

Q3: Are there Audit Table formats compatible with Topicus' needs?

- How much space do they consume?
- How do they affect the performance when rolling back entities?

1.4 Approach

In order to determine which auditing solutions are best suited for Topicus' needs, we first create a list of criteria. These criteria's are chosen from the ISO 25010 standard, which describes many quality standards for software.

Then a literature study will be done to find out which Auditing solutions exists, both on the server side, compatible with Microsoft SQL Server, and on the application side. The criteria are used to compare them to each other. Once this study is completed, the most promising solutions are used for further research, in order to get a more accurate picture of their actual efficiency.

A prototype is created for each of the chosen solutions, so that the solutions can be better compared to each other. The structure and design of each prototype is then discussed. Both prototypes are capable of intercepting Insert, Update and Delete actions and can then audit those actions.

Then we define the test environment and test setup which we used, so that the tests can be replicated. We will explain how we tested the performance of the chosen solutions.

We then discuss the results of these tests, and compare the chosen solutions to each other by using the test results. Based on the discussion, we then make a final choice between the solutions.

After choosing the audit solution, we discuss strategies for rolling back a given entity to an earlier version. We investigated if there are any existing strategies and then defined our own strategies. Issues we ran into as we implemented these strategies, are also discussed.

Then we investigate which design strategies exist for audit tables, and we investigate which of these designs are most suited, for Topicus' needs, and for rolling back entities.

Finally we conclude on this research by discussing the results from the investigations, and we give recommendations for further research.

2. Relevant Criteria

In this chapter we discuss the criteria by which we will measure the investigated audit solutions. First we investigate if any standards exist for measuring the quality of software. Then we define the criteria which we will use to compare audit solutions.

2.1 Software quality standards

Over the years there have been several standards for measuring the quality of software. The International Organization for Standardization [26], introduced the ISO/IEC 9126-1 standard in 2001. In 2011 this standard was replaced by the ISO 25010 standard. The ISO 25010 quality standard is the current standard for measuring the quality of software, and covers all the requirements for the audit solution, thus we decided to use this standard. The solution needs to be capable of gathering all the information we wish to audit, and must ensure the creation of a complete and correct audit trail. The performance cost of the solution must also be low, the overhead it introduces must be as low as possible in order to limit the effect on existing applications. The solution must also be capable of handling errors and crashes, if it isn't reliable then we cannot ensure the creation of a complete and correct audit trail. The solution might also need to be deployed on multiple databases and/or applications, thus it should be modular, extendable and easy to install on other applications/servers. Using these requirements, we selected the following criteria from the ISO 25010 standard.

2.2 Criteria

The audit solution needs to be functionally suitable, it needs to be capable of auditing all the information we desire. This is called functional completeness. The minimum amount of information we wish to audit is which user is responsible for the event, when the event happened, what kind of event it was (insert, update or delete), which entity was modified by the event, which columns were changed and how they were changed. This information also needs to be correct and the audit trail must contain all the events which occurred, this is called functional correctness.

Besides a correct and complete audit trail, the audit solution must also perform efficiently, introducing as little overhead as possible for the existing applications and using as little disk space as possible. The overhead can be judged based on several factors. One of these factors is if the solution is blocking or non-blocking. With a blocking solution it could block access to the table for other concurrent actions, or the application needs to wait for the auditing to complete, before the application can finish the action and continue. A non-blocking solution allows the application to continue without having to wait for the solution to store the audit information, and it does not block access to a table in the database for other concurrent actions. Another factor is if the solution offers the option to offload some of the auditing work to another system. For solutions where this is possible, the overhead can be a lot lower compared to solutions which need to do everything on the same system as the database.

The auditing solution must also be reliable. It needs to be fault tolerant and continue to work if it runs into errors. If the solution crashes then it should be capable of recovering from the crash without losing any audit information, the audit trail may not end up with holes as this will make reverting entities to earlier versions very hard to do correctly, if not impossible.

The solution must also be maintainable, this can be measured by how modular and reusable the solution is. The solution should both be reusable and modular, so it can easily be applied to multiple existing applications or databases, without having to change existing code or table structures. If an audit solution requires a lot of changes to existing systems or tables, then it will be a lot of work to apply it to every system and could cause other compatibility issues.

Finally the solution needs to be portable. This means that the solution should be adaptable. If it is applied to the applications then it would be preferable if it is not limited to just one programming language. For solutions on the database side we will limit us to Microsoft SQL Server as Topicus makes use of that, so it is important that upgrading to a newer version of SQL Server will not pose a problem for the solution. For the solution to be portable, the process of installing and removing the solution must also be efficient, this is called Installability. It might need to be installed on many servers and applications, so it needs to be efficient to install.

2.3 Criteria priority

Some of these criteria are more important than others. In deliberation with Topicus we determined which of the criteria are must haves, and which are should haves.

Functional Suitability is most important by far, if the found solution cannot store all the audit information we need, or if it cannot guarantee the creation of a complete and correct audit trail, then the solution is unusable. Secondly comes Efficiency. If the solution introduces too much overhead, then it cannot be used on a live system. It also needs to introduce less overhead than the audit system Topicus currently uses. Lower overhead is better. Reliability is also very important, if the solution isn't fault tolerant, or if it can cause holes in the audit trail when it crashes, then it isn't usable. The solution should not cause the system, on which it is applied, to grind to a halt when something goes wrong.

Less important than these three, but still nice to have, are the criteria that the solution should be maintainable and portable. It would be great if it is maintainable, so the solution can easily be modified or extended when changes, or slightly different versions, are needed. Portability would also be nice to have. If the solution needs to be installed on a lot of different systems then it would be nice if this is a quick and smooth process. If it needs to be removed, then it would also be great if this can be done quickly. It would also be great if there aren't any potential compatibility issues between different SQL Server versions, or between different versions of the used programming language on the application side.

The following list summarizes the criteria which we will use to judge and compare the solutions:

- Must Haves
    1. Functional Suitability.
    2. Efficiency.
    3. Reliability.
- Should Haves
    4. Maintainability.
    5. Portability.

3. Investigated Solutions

Now we investigate which methods are available to audit information on the database side, for Microsoft SQL server 2012, and on the application side. Four different solutions were found which work on the database side, which are detailed below. They are Triggers, Change Data Capture, Reading transaction logs and the Service Broker. We also investigate handling the auditing from the application side, in a modular fashion, by using Aspect Oriented Programming and Hibernate events. To determine which of the found solutions meet Topicus' needs, and which solution meets them the best, we compare them to each other with the criteria defined in chapter 2.

3.1 Solutions

3.1.1 Microsoft SQL Server 2012 compatible audit solutions

First we investigated which audit solutions currently exist, and can be used with Microsoft SQL Server 2012. When searching for "SQL Server Audit Trail" we found the Triggers solution [22]. Searching for "SQL Server track and tracing" gave us Change Data Capture [16]. The Service Broker solution [13] was found by searching for "SQL Server 2012 auditing solutions". Other options were found along the way, like Change Tracking [17], SQL Trace [25] and SQL Server Audit [31], but these three can't be used to audit the actual changes to the entities in the database, and thus aren't discussed in more detail here.

3.1.1.1 Triggers

Triggers are special stored procedures which are automatically executed after certain events. There are several kinds of triggers, the one relevant to us is the DML trigger [22]. DML triggers are run when data is modified by an Insert, Update or Delete statement on a table or view. They can be run before, after or instead of these events. For Auditing, the After version is well suited and has access to the *Inserted* and *Deleted* tables. When a row is updated, the old version can be found in the deleted table, while the new version can be found in the inserted table. This makes it possible to find out exactly which changes were made to the row. These triggers need to be created and enabled per table.

3.1.1.2 Change Data Capture

Change Data Capture (CDC) [16] can be used to record changes (Inserts, Updates and Delete actions) applied to an SQL Server table. CDC has to be enabled per table. All Insert, Update and Delete actions on those tables are recorded in CDC's change tables. Each audited table has a corresponding CDC change table, which mirrors the column structure of the original table. These change tables have several extra columns added to them, which contain metadata.

Change Data Capture works asynchronously from the actual inserts, updates and deletes. CDC checks the transaction logs once every 5 seconds and reads a maximum of 1000 transactions from that log. If the database is under a heavy load then it will read less entries from the transaction log. It then writes the information from those transactions to the corresponding change tables. Even in simple recovery mode, where normally the log is automatically truncated SQL Server, the log truncation point will not advance until all the changes marked for capture by CDC have been processed. Thus the transaction log can become quite large if the database is under a heavy load, so proper measures have to be taken to allow the transaction log to grow. A cleanup job runs at 2 am by default, removing all entries from the change tables which are older than 3 days. It is possible to adjust this amount of days, and the cleanup job can also be disabled if all information should be retained indefinitely.

3.1.1.3 Transaction Log Reading

Change Data Capture reads information from the transaction logs, this can be a way to audit the data without specifically using CDC. There are third party applications, which can be used to look at the transaction logs [33] [32], they can show you which changes were made and even who made them. This confirms that the necessary information can be found in the transaction logs. The "DBCC Log(databasename, typeofoutput)" command can be used to query the transition log, but unfortunately Microsoft hasn't given out any documentation for this command. The "fn_dblog" command can also be used for this, but it too hasn't been documented.

3.1.1.4 Service Broker

The Service Broker is a feature in SQL Server since 2005. It can be used to send and receive guaranteed, asynchronous messages [13]. These messages are sent to a queue, which can be located in the same database as the sender, in another database in the same SQL Server instance or in another SQL Server instance.

The Service Broker can be combined with a simple DML trigger, which triggers on Insert/Update/Delete events and then creates a message. This message is then send to the target database, where it arrives in a queue. An internal or external process can then be activated [28], which reads messages from this queue, processes them and moves the information into an audit table. It is also possible to create the message on the application side, and then hand it over to the Service Broker through a query, bypassing the need for a trigger.

The Service Broker supports transactional messaging, so messages are sent and received as transactions. If a transaction fails then the actions are rolled back. Messages which were received by the internal or external process are returned to the queue, so that they can be received and processed again. This ensures that no message is lost. Messages which are send out aren't actually dispatched until the transaction has been committed, so they will be removed if the transaction fails. The Service Broker also guarantees that the messages are processed in order, and it ensures that the same message is never delivered twice. If the target received a message but the acknowledgement which it had sent back was lost, then the same message will be send again. The Service Broker on the target side will see that this message was already received, discard it and send another acknowledgment.

Multiple databases can send messages to the same target, making it possible to use one database for centralized auditing. There are examples where a database can receive up to 18.000 messages per second when configured correctly [12], and techniques exist through which it is possible to process up to 2700 messages per second [11].

3.1.2 Application side compatible Audit Solutions

Auditing can also be implemented from the application side. While these solutions cannot audit events which were done directly on the database, they potentially make up for this in other areas. As we want a modular and reusable solution, we searched for techniques which allow us to add auditing without having to change much, if any, of the base code of the application. Topicus makes use of C# and java, so we searched for techniques compatible with those languages. A technique we found is Aspect Oriented Programming (AoP). AoP was introduced to address concerns like Tracing and Logging [36], and can be used to log changes in data values. With auditing we are keeping a log of every change made to every entity, so this fits right in with the AoP use cases for Logging. Topicus also makes use of Hibernate [24] for ORM purposes, so we also investigated if Hibernate has any features which can be used for auditing. Hibernate fires events when, for example, entities are updated, inserted and deleted. Several event listeners can be implemented in order to react to those events. They can be used to add auditing to the application.

3.1.2.1 Aspect Oriented Programming

With Aspect Oriented Programming it is possible to add extra functionality before and after existing methods. This can be used to add functionality around the methods which modify the database. For C# there is PostSharp [21], which can weave it´s logic into the program at compile time. This can result in little to no noticeable performance impact, compared to implementing the same functionality without PostSharp [3]. For Java there is AspectJ [37], which can also weave its logic into the program at compile time, at little to no noticable performance impact.

AoP can then be used to add functionality around existing methods used for inserting, updating and deleting entities. This extra functionality can be used to write the audit data directly to the audit database after every action. It can also be used to create a message with all the audit information in it, which can then be send to an application on the same system as the audit database. This application can then take care of writing the information to the audit database, making it possible to offload some of the work to that system, but it does require extra work to create a reliable messaging system. The Service Broker can also fulfill the role of reliable messaging system for the AoP solution.

3.1.2.2 Hibernate Events

Hibernate has events and event listeners, which can be used to add extra functionality around insert, update and delete actions. These event listeners can be used to add functionality before and after these actions. The event gives you access to a state array, which contains the values of all the rows of the entity, and there is another array which contains the names of those columns. Both arrays have the exact same ordering. For update events there is also an *oldState* array, containing the values of the columns from before they were updated. By using the state and oldState arrays, it is possible to determine exactly which changes were made.

Then the same things can be done as with the Aspects, you can create a message and send it to an application on the system with the audit database, or you can audit the information directly.

3.2 Functional suitability

First we compare the found solutions on their functional suitability. We investigated if they can supply all the audit information we need, and if they can assure the creation of a complete audit trail.

3.2.1 Triggers

A DML trigger on a table will ensure that every single insert, update and delete action on that table is audited. If the trigger fails to audit the action then as a consequence the entire action will fail, preventing the creation of holes in the audit trail. The trigger can contain a query to retrieve the current time and the user responsible for the action. It can also retrieve the column values from the inserted and deleted tables. This gives it access to all the information we need for auditing, allowing this solution to create a correct and complete audit trail.

3.2.2 Change Data Capture

Change Data Capture reads entries from the transaction log and then audits the actions into its own audit tables. Unfortunately you have no control over which columns are present in these audit tables. The audit tables contain the same columns as the tables for which they contain audit information, with some extra metadata columns added to them. None of these metadata columns contain which user is responsible for the action. According to Microsoft, Change Data Capture wasn't designed with auditing in mind, it is a feature targeted at data warehousing scenarios [18]. The only way to store this information in the change tables is to add an extra column to the original table, with a default value equal to the user who last modified the table. A similar issue exists for the time at which the action occurred. This is not recorded in the audit tables, instead it can be found in the *cdc.lsn_time_mapping* table, but finding the right time for each action can come at a hefty performance cost. Adding an extra column to all your tables, defaulting to *GetDate()*, is another way to add the change time to the audit data. Adding the extra column again requires the modification of all original tables. So Change Data Capture can only be used to create a complete audit trail with all the information we need, if all the tables are modified and have these two columns added to them.

3.2.3 Transaction Log Reading

As mentioned in 3.1.1.3, applications exist which can read from the transaction logs. They can recover all the information we are interested in, like which user is responsible for the action, when the action occurred and which columns were changed. So clearly this information is present in the transactions logs and we should be able to access the information by parsing the logs. Once the parser has retrieved the information it just needs to move it into an audit table. This solution gives us more freedom than Change Data Capture as we can determine the structure of our audit, and we do not need to add extra columns to the original tables in order to retrieve the name of the user responsible for the action, or the time at which the action occurred.

3.2.4 Service Broker

The Service Broker is mainly a reliable messaging system, many auditing examples of the service broker use a Trigger to gather the audit information. The trigger then turns the audit information into a message and sends it to the other database by using the Service Broker. The Aspect Oriented Programming solution, or the Hibernate events, could also be used to gather the audit information. The reliable message handling of the Service Broker guarantees that no messages are lost along the way, allowing us to audit all events and create a complete and accurate audit trail.

3.2.5 Aspect Oriented Programming

Aspect Oriented Programming works on the application side, so the current time and the name of the current user can be retrieved there. Finding out which columns were modified by an action and how they were modified is also possible by extending the right methods. Take for example an aspect which works around the method responsible for updating entities. Before the start of the update it can retrieve the current state of the entity from the database. After the update has been completed, it can compare the retrieved state to the updated state of the entity. It can then compare these two versions and determine what was changed. If auditing happens in the same database then it can use the same transaction, as the one in which the update occurred, to insert a new entry into the audit table. If the transaction fails then both the update action and the insertion of its audit information are rolled back, ensuring that the audit trail is both complete and correct.

If we want to store the audit information in another database then a reliable messaging system needs to be used, so we can ensure that the message cannot be lost. The Service Broker solution is viable for this, but we can also build our own messaging system. There are no obstacles present which make it impossible to create a complete and correct audit trail, as long as all changes are made through the application. Actions which are done directly on the database, bypassing the application, cannot be audited, instead resulting in an incomplete audit trail.

3.2.6 Hibernate Events

Hibernate Events also work on the application side. The Event Listeners have access to a state array of the entity, which contains the column values that the entity will have after the action has been completed. For updates there is also an *OldState* array, which contains the current values of the entity's columns. We can determine what was changed by comparing these two arrays. The event listener can also access the session and transaction in which the event occurred, which allows us to insert the new audit entry into the database in the same transaction. As with the AoP solution, this ensures that the audit trail is both complete and correct. The option to store the audit information in another database, by using a reliable messaging system, also exists. There are no obstacles present which prevent the creation of a complete and correct audit trail, as long as all changes are made through the application. Actions which are done directly on the database, bypassing the application, cannot be audited, instead resulting in an incomplete audit trail.

3.2.7 Functional Suitability Conclusions

Almost every solution scores the same on Functional Suitability. Each of them can ensure the creation of a complete and correct audit trail, but Change Data Capture cannot give us all the audit information we need. We need to modify the existing tables in order to gather the missing information. Due to this limitation, CDC scores far lower on Functional Suitability. AoP and Hibernate Events score slightly lower because they can only audit actions which are done by the application, if actions are done directly on the database then an incomplete audit trail will be created.

	Functional Suitability
Trigger	++
CDC	
TLR	++
SB	++
AoP	+
Events	+

3.3 Efficiency

Now we compare the solutions on their efficiency. For this we have investigated what is known about the overhead of the solutions and whether they lock the table, or row, on which the audited action was executed. For the solutions which come with their own audit tables we also investigated how much space these tables consume.

3.3.1 Triggers

Triggers lock the table on which the action occurred. No other actions can happen on this table until the trigger has completed. If the trigger fails then so does the action itself. A performance benchmark has been found, which shows that using a trigger is slower than actually using a separate second query [14], though the total extra overhead was only a few microseconds. Topicus currently has an audit system using triggers, which has too great a performance cost. So The Efficiency of triggers doesn't seem to be that great.

3.3.2 Change Data Capture
Change Data Capture does not lock the table on which the action occurred. It happens asynchronously from the actual event. Once every 5 seconds it reads from the transaction logs and then audits that information. If the server load is high then it will read less lines from the logs in order to reduce the impact it has on the performance. It can also be disabled during high load times, to prevent it from having any performance impact during those times. If new transactions are entered into the log faster than that CDC processes them, then this can cause the transaction log to grow quite large and take up a lot of space. Change Data Capture does have the disadvantage that you cannot modify the structure of its audit tables and determine what data it stores in them. With every action it stores a complete copy of the entity in the audit tables. For an update action it will even include the values of columns which weren’t modified by the action. Thus a lot of unnecessary information is stored in the audit table, making it consume more disk space than necessary. A possible solution to this is to create your own audit tables and to regularly move data from the CDC audit tables to your own audit tables. This way you can cut out the information you don’t need, but moving the data will be costly on busy servers.

3.3.3 Transaction Log Reading
Transaction log reading, just like CDC, does not lock the table on which the action occurred. They also share the problem that the transaction log can become quite large if sufficient entries aren’t taken from it regularly enough. A well implemented reader should have a low performance impact. Unlike with Change Data Capture, you can use your own audit tables from the start, so it will not be necessary to move the audit data from one table to another. Thanks to this a custom transaction log reader is more efficient than Change Data Capture.

3.3.4 Service Broker
For the Service Broker, if it locks the table, on which the action occurred, depends on which other audit solution is used to gather the audit information and create the message for the service broker. If a trigger is used then it will lock the table, but if an application side audit solution is used then it won’t lock the table. When using an application side audit solution it will only affect the performance of the application as it has to wait until the message has been sent, before it can commit the transaction in which the action itself occurred. If the audit information is stored in another database on another server, then for update actions the message can contain the column values of the entity before and after the action, determining which columns have changed and storing the information in audit tables can be done in the other server. This makes it possible to offload a lot of the performance impact to a different system, allowing the Service Broker to be very efficient and improve the efficiency of some of the other solutions.

3.3.5 Aspect Oriented Programming
The Aspect Oriented Programming solution has very little to no noticeable performance impact compared to implementing the same functionality without AoP. For C# there is PostSharp [21], which can weave it´s logic into the program at compile time [3]. For Java there is the AspectJ library [37], which can also weave its logic into the program at compile time.

The Aspect Oriented can be used to add extra functionality around existing methods, like the methods responsible for the insert, update and delete actions. Those methods don’t finish until the extra functionality has completed its work, thus those methods will take longer to complete. In the case of an update action, before the update happens we first need to retrieve the current values of the entity from the database. This creates a lock on the row in question, preventing the row from being modified but allowing other reads. It is unlikely that the the row will be modified by another action at the same time as this update action, so this should have a minimal effect on the performance. Then we need to determine how the entity was changed and store this information in the audit table. Only then does the original method finish. The amount of extra functionality required to create and store the audit information and be reduced by storing the audit information in another database on another system and sending the information there by using a reliable messaging system.

3.3.6 Hibernate Events

Unlike the AoP solution, the Hibernate Events solution has access to arrays which contain the old and new values of an entity when it is updated, so it doesn't have to retrieve the entity from the database before the update action occurs. As with the AoP solution, a reliable messaging system can be used to process and store the audit information on a different system, further reducing the performance impact. This solution also has very little performance impact compared to implementing the same functionality without events, as event listeners are effectively called through a delegate invocation.

3.3.7 Efficiency Conclusions

Triggers score worst on Efficiency as they lock the entire table, and both benchmarks and the audit system current used by Topicus show that its performance isn't that great. Transaction Log reading scored the highest since it doesn't lock any tables and you have full control over the format of the audit tables. CDC also doesn't lock any tables, but the audit tables require much more disk space than necessary because we have no control over their format. The AoP and Hibernate Events solutions also scored well. The AoP and Hibernate solutions have in common that both add some overhead to the application on which they are applied. They can reduce this overhead by offloading some of the auditing workload to a second, separate, system. Neither locks a table like Triggers do. The Service Broker can make use of the Trigger, AoP or Hibernate Events solution in order to audit the events. these three solutions have in common that they can all be partially blocking and that they can be used to offload some of the auditing workload to a second, separate, system. Finally the Service Broker also scored well, as the SB itself does not lock any tables and can work together with Triggers, AoP or the Hibernate Events in order to audit information with less overhead, than when using one of those three solutions on its own.

	Efficiency
Trigger	
CDC	++
TLR	+++
SB	++
AoP	++
Events	++

3.4 Reliability

Now we compare the solutions on their reliability. We will discuss how fault tolerant they are and if they can recover from crashes without losing audit information.

3.4.1 Triggers
If something goes wrong inside the trigger, which causes the auditing to fail, then we can throw an exception. By doing this the original action will also fail, preventing the creation of holes in the audit trail. Try-Catch blocks can also be used in the Trigger to handle errors from which it can recover immediately. The actions of the trigger take place in the same transaction as the event which triggered the trigger. If a crash occurs then both the original action and the auditing done by the trigger will still happen, or they will both be rolled back. This ensures that the audit trail remains complete. So information will not be lost through crashes.

3.4.2 Change Data Capture
Change Data Capture has a capture job which takes entries from the transaction log, this job can be enabled or disabled without losing audit information. It just won't take entries from the transaction log while it is disabled. Since it reads from the transaction logs and audits that information, errors and crashes will not cause a loss of data.

3.4.3 Transaction Log Reading
A transaction log reader would work separately from the database, taking entries from the transaction log and auditing those. There are no inherent obstructions or limitations which would make it unreliable.

3.4.4 Service Broker
The Service Broker makes use of queues to store audit messages. An internal or external activator procedure can then receive messages from such a queue and process them. These queues can contain poison messages. A poison message is a message which cannot be successfully processed by the activator procedure. When the procedure receives messages from the queue and attempts to process them, it will fail to process a poison message. If this happens then the message should not be lost, as that would leave a hole in the audit trail. A simple implementation is to roll back the transaction in which the message was received from the queue, this returns the message to the queue. Each queue has poison message handling enabled by default. If a transaction is rolled back five times then the poison message handling will disable the queue, so that the activated procedure stops trying to process a message which it cannot process. A better implementation of the activated procedure does not roll back the transaction. Instead it can move the poison message to a dedicated error table in the database. Then the procedure can continue with the next message without having to roll back the receive action. This involves the proper usage of try-catch blocks in order to catch any errors, and never rolling back the transaction in which the messages were received from the queue. Receiving messages from the queue, auditing the messages and writing errors to the error table must be done in the same transaction. If this is done in separate transactions then message loss can occur if the procedure crashes, but if it is done in the same transaction then everything will be reverted if the procedure crashes and the messages will be returned to the queue. The messages stored in the error table can then be used to identify the problem in order to prevent it from occurring again. They still need to be processed manually to prevent holes from showing up in the audit trail.

Due to the usage of queues, the applications which make changes to the database can continue to work even if the Service Broker has stopped working, for example when the connection between databases isn't working. The generated messages will be stored in the queues and can be send out when the connection is working again. The queues can grow quite large, the main limit is the available size on your hard disk. It is also possible to use queries to monitor the status of the Service Broker, which makes it possible to detect such problems and inform the right people automatically, through for example e-mail [30].

3.4.5 Aspect Oriented Programming
The Aspect Oriented Programming solution is implemented on the application side. If the auditing information for an entity is stored in the same database as the entity, then we need to ensure that audit information is inserted in the same transaction which is used for the insert/update/delete action. This way if an error occurs while writing away the audit information, then the transaction can be rolled back and the action itself can be prevented, this prevents the creation of holes in the audit trail. If instead the audit information is stored in another database, and a reliable messaging system is used, then the transaction should only be rolled back if the application fails to give the message to the messaging system. Once the message has been given to the messaging system, the reliability of this solution depends on the used messaging system.

3.4.6 Hibernate Events
For the Hibernate Events solution the reliability is similar to the Aspect Oriented Programming solution. If the auditing happens in the same database then the audit information must be inserted in the same transaction just as with the AoP solution. If the audit information is stored in another database, by using a reliable messaging system, then the transaction should only be rolled back if the messaging system won't take the message.

3.4.7 Reliability Conclusions
When it comes to reliability, CDC and Transaction Log Reading score best as everything else can keep working if they fail and no audit information is lost. The Service Broker, Triggers, AoP and Hibernate Events also score quite well. SB prevents the loss of information and maintains a complete audit trail thanks to its reliable queues and transactional messaging system, while Triggers, AoP and Hibernate events can hook into the transactions used by the insert/update/delete actions, which makes it possible to ensure that the audit information is correct and complete. If a transaction is rolled back then all actions, including the insertion of the audit information, is reverted. The SB doesn't score as well as CDC and TRL because it depends on another solution to hand it the audit information, which means that it can't score better than Triggers, AoP and Hibernate events.

	Reliability
Trigger	+
CDC	++
TLR	++
SB	+
AoP	+
Events	+

3.5 Maintainability
Now we investigate how maintainable each solution is. We will look at both the modularity and reusability of the solutions. A database side solution shouldn't require any changes to existing database tables in order to function, and application side solutions should require little to no modifications to the existing applications.

3.5.1 Triggers
The trigger needs to be added to each table for which we want audit information. Modifications to the structure of the table will not be required for the trigger to function. If the trigger is written correctly then it should work for every table, without requiring any significant modifications. But if you need to change the trigger then you will need to change every trigger separately, this makes it difficult to maintain and modify them in large databases. This can be somewhat mediated by moving as much of the trigger as possible into a stored procedure, which the trigger then calls. The main thing the trigger will always have to do is to retrieve information from the inserted and deleted tables, but everything else can be done inside a stored procedure.

3.5.2 Change Data Capture
Change Data Capture requires the modification of every table as new columns need to be added to record the name of the user who last changed a row and to record the time at which this happened. The solution is anything but modular due to these required modification.

3.5.3 Transaction Log Reading
The Transaction Log Reading solution requires no modifications to existing tables as it can retrieve all the needed information from the transaction log. It can function as a completely separate module, all that it requires is that the database does not delete entries from the transaction log before they have been processed by the reader.

3.5.4 Service Broker
The Service Broker does not require modifications to existing tables and will work no matter their structure. It does require an extra module which intercepts insert, delete and update actions and then creates an audit message. This can be done by a Trigger, Aspect oriented Programming or the Hibernate Events. These three can use a query to send a message with the Service Broker. Multiple Service Broker connections are possible between databases, so adding an extra connection for auditing isn't a problem and does not require changes to the existing connections, making it a very modular solution. The Service Broker queues can also be accessed by internal and external procedures, making it possible to use the Service Broker as a part of other applications, this also makes it reusable.

3.5.5 Aspect Oriented Programming
The Aspect oriented Programming solution can add functionality around any desired method. The method itself doesn't need to be modified in any way and the AoP solution can be added to the application as a completely separate module. If the methods it needs to extend are different between applications, then only minor modifications will be required in order to make the aspects extend those methods instead. The solution can be added as an assembly to existing applications. This way if the event listeners need to be modified then you only have to do this once, making this a very maintainable solution.

3.5.6 Hibernate Events
The Hibernate Events solution requires that the event listeners are added to Hibernate's configuration. Besides that it can work as a completely separate module. It has access to all the information it needs through the event object, which is supplied as an argument to the event listener. The solution can be added as an assembly to existing applications. This way if the event listeners need to be modified then you only have to do this once, making this a very maintainable solution.

3.5.7 Maintainability Conclusions

Change Data Capture scores badly on maintainability because it is lacking in modularity. All the changes required to existing tables are a very strong disadvantage in this area. Triggers score better but also have a disadvantage, as a separate trigger needs to be created for each table, making it harder to maintain them. The Trigger solution still scores reasonably well, since this disadvantage can mostly be mediated by moving as much functionality from the trigger to a stored procedure and then calling this procedure from the trigger. Finally Transaction Log Reading, AoP, SB and Hibernate Events all score quite well as they are all modular, they do not require the modification of existing tables or code. They are also quite re-usable, if they need to be modified then this only has to be done once per database/application, instead of several times like with Triggers.

	Maintainability
Trigger	+
CDC	-
TLR	++
SB	++
AoP	++
Events	++

3.6 Portability

Finally we investigate how portable each solution is. For the database side solutions we investigate if upgrading the database to a newer version than SQL Server 2012 could cause a problem now or in the future. For the application side solutions we investigate if the solution can be applied with both Java and C#, or if it is limited to one of the two languages, as Topicus makes use of both languages. For all solutions we also consider how hard it is to install and remove the solution.

3.6.1 Triggers

It is easy to install and remove triggers, this can be done through SQL queries. Triggers haven't had compatibility problems between SQL Server versions, so it is unlikely that server upgrades will pose a problem.

3.6.2 Change Data Capture

Change Data Capture can be enabled or disabled through a query. It comes with SQL Server, so you don't need to install anything extra. You must first enable CDC for the database, once that is done you need to enable it for each table. Some settings might need to be configured to your liking, like how long information should be retained in the audit tables. This can all be done through a query. The only issue is that each table must be modified, two columns need to be added to each table in order to record who is responsible for an action and to record when the action occurred. If a lot of tables are present in the database then this can be a lot of effort. This makes it much harder to install and remove CDC.

3.6.3 Transaction Log Reading

Installing the Transaction Log Reader should be pretty straightforward, the only modification you need to make to the database is that it shouldn't delete entries from the transaction log by itself.

One big issue which affects the portability of this solution is that the syntax of the log has changed between SQL server versions in the past. It is possible that this will happen again in the future. This means that the parser might have to be completely rewritten after a database upgrade. Considering how badly the needed functions for parsing the log are documented, this would be a large amount of work.

3.6.4 Service Broker

The Service Broker can be enabled/disabled and set up on a database through SQL queries, making it possible to quickly install the Service Broker. The Service Broker is a part of SQL Server, so compatibility issues with future versions of SQL Server are unlikely.

3.6.5 Aspect Oriented Programming

Installing the Aspect oriented Programming solution is straightforward, the aspects just need to be added to the application and then configured to extend the correct methods. The solution can be removed by deleting the aspects or changing the configuration of the aspects, so they no longer hook into any existing method. The AoP solution is compatible with both C# and Java as AoP projects exist for both languages. With C#, when using PostSharp, you can also disable the aspects when building your application, this way you don't have to change the aspects or their configurations. In Java, with AspectJ, it is also possible to prevent the aspects from being added at compile time.

3.6.6 Hibernate Events

The Hibernate Events solution can be used in both Java and C#, as for C# a port named NHibernate exists. The event listeners need to be added to Hibernate's configuration object in order to enable them. Disabling them can be done by removing the code where they are added to the configuration object. Unfortunately this solution only works for applications which make use of Hibernate or NHibernate, it cannot be applied on an application which does not use this. If such an application exists and makes use of the same database, then you will get an incomplete audit trail unless you apply a different audit solution to that application.

3.6.7 Portability Conclusions

Transaction Log Reading scores horribly, as the risk of having to rewrite the entire solution is far too great a disadvantage. Change Data Capture also doesn't score that well because installing it takes quite a bit of work, especially because you need to add extra columns to every table. Hibernate events score reasonably, but fall slightly short because they are only compatible with applications which use (N)Hibernate. Triggers, AoP and the Service Broker all score quite well on compatibility, they are also easy to install and remove. This gives them the highest scores on Portability.

	Portability
Trigger	++
CDC	--
TLR	---
SB	++
AoP	++
Events	+

3.7 Conclusions

From the previous investigations we can now conclude which solutions are better suited for our goals than others. The trigger solution isn't efficient enough and also has some reliability issues, but scores well in the other areas. Change Data Tracking isn't modular enough due to the changes which must be made to each table, this also makes it harder to install and remove. Transaction Log Reading looked like a great solution until we came to portability, there it falls short due to compatibility issues between SQL Server versions and the amount of work which would be required to make it compatible if the transaction log syntax is changed in the future. The Service Broker does well overall, it is not the most efficient solution and there can be issues with sending and auditing faulty messages if the system isn't implemented properly, but it is quite portable and maintainable. The Aspect oriented Programming solution overall does quite well too, coming out only slightly lower than the Service Broker. Finally the Hibernate Events solution also does well, but loses points on portability as it isn't compatible with systems which don't use (N)Hibernate.

The following table sums up the results. The criteria which were defined as must haves in chapter two are weighted 50% heavier than the criteria which were defined as should haves. The *Total* column then contains a number, which is how many plusses the solution achieved over all five criteria after taking the weights into account.

	Functional Suitability	Efficiency	Reliability	Maintainability	Portability	Total
Trigger	++		+	+	++	6
CDC		++	++	-	--	3
TLR	++	+++	++	++	---	9.5
SB	++	++	+	++	++	11.5
AoP	+	++	+	++	++	10
Events	+	++	+	++	+	9

TABLE 3.7.1

As can be seen by these results, there are alternative auditing solutions which meet Topicus' needs, and are compatible with Microsoft SQL Server or Topicus' applications. The Service Broker scores higher, but we cannot yet, with sufficient accuracy, answer the sub-question about how much performance overhead these solutions introduce. Just a one point difference in Efficiency for the SB and AoP can change the outcome on which solution is the most suitable, thus accurate information on the performance of the best solutions, so the best decision can be made, is important.

In order to be able to answer this question on the actual overhead of the solutions, we have decided to create prototypes of the most promising solutions; the Service Broker and Aspect Oriented Programming. These prototypes will be put through tests in order to get a more accurate picture of their actual Efficiency. In the following three chapters we will discuss these prototypes, the tests and the test results.

4. Prototypes

In chapter three we discussed the possible solutions and their advantages and disadvantages, using the criteria defined in chapter two. At the end of chapter three, we picked the most promising solutions. We created prototypes of each in order to find out more about their performance.

We created one prototype of the Service Broker, using a very basic trigger in order to react to the events. We decided to use a trigger in this prototype so it is a completely database sided prototype. The second best scoring database sided solution was the trigger, and it is the only other database sided solution which is compatible with the service broker, making it the obvious choice.

For the AoP solution we created a prototype, which makes use of TCP in order to send audit messages to another application. This application then inserts the audit information into an audit database. This way the prototype uses a second system just like the Service Broker, making for a better comparison between the two.

Finally a hybrid prototype was created, in this hybrid the trigger from the Service Broker prototype is replaced with the aspects from the AoP prototype. We choose to create this hybrid prototypes because we are very interested in seeing how the solutions perform when we combine them. It also allows us to directly compare the performance of the aspects to that of the triggers. It will also provide a clearer picture of the Service Broker's own performance overhead, if we have more than one prototype using the service broker.

We will now discuss the designs of these prototypes, so that the tests can be duplicated. All the tests assume that the database where the audit information is stored, which we shall call the *Target, is* a different database from the one which contains the entities, which we shall call the *Initiator*. Each database is contained in a separate SQL Server instance. Chapter five will go into the details of the test environment, the server instances and the tests themselves, while chapter six will discuss the results of the tests.

4.1 Service Broker Prototype

A Service Broker requires several components in order to work. Diagrams 4.1.1 and Diagrams 4.1.2 give a quick overview of how the Service Broker needs to be set up on each side. The individual parts shown in the diagrams will be discussed in the following sections.

4.1.1 Endpoints

For communication between SQL Server instances an Endpoint needs to be created for each SQL Server instance [19], so that we are listening for Service Broker messages on a port of our choice. We will need to use these port numbers later, when setting up the Routes between the Server instances.

4.1.2 Users and Certificates

Two users are needed, one representing the *Target* database and another representing the *Initiator* database. Certificates need to be linked to each of them, so communication between the server instances can happen in a secure manner. Each database must have a Master Key before these certificates can be created. Figures 4.1.1 and 4.1.2 show that these users are needed for the local and remote services.

So first we create a Master Key for both the *Initiator* and *Target* database, if they don't have one already. Then we create the *AuditTargetUser* in the *Target* database, without login, and we create an *AuditInitiatorUser* in the *Initiator* database, also without login.

Then we create a certificate for each of these users and we back both certificates up to files. Once this is done, we can create an *AuditTargetUser* in the *Initiator* database and an *AuditInitiatorUser* in the *Target* database. These users need to use the same certificates, so we create them from the files we created earlier. Now each database should have two users and two certificates.

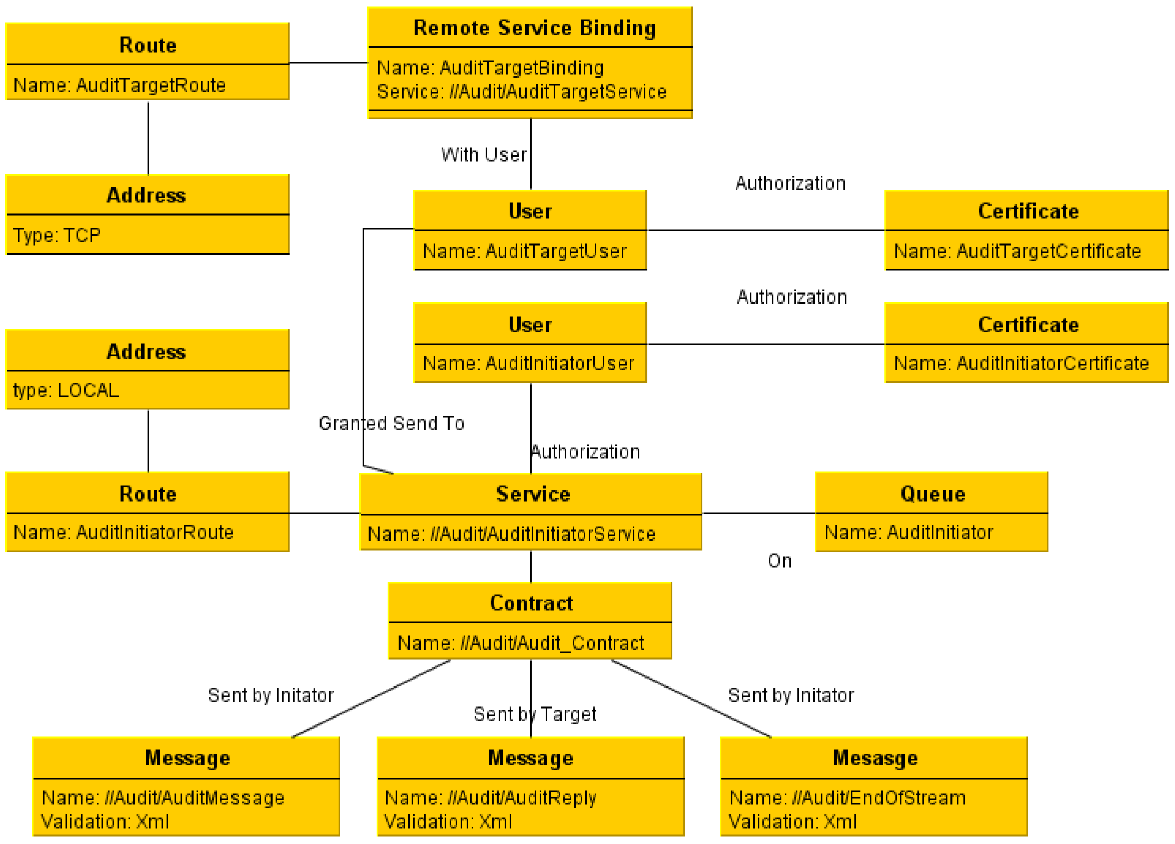

FIGURE 4.1.1 SERVICE BROKER SETUP – INITIATOR DATABASE

4.1.3 Messages, Contracts and Conversations

In order to send messages, and so we can have conversations between both databases, we need to define which messages can be send and who can send them. This needs to be defined in both databases. As can be seen in Figure 4.1.1, the *AuditInitiatorService* will require this contract in the *Initiator* database, while in Figure 4.1.2 the *AuditTargetService* will require it in the *Target* database. The defined contracts and messages are exactly the same on both ends.

One message we want to send is the *//Audit/AuditMessage*. This message will contain all the audit information related to the Update, Insert or Delete event. This message needs to have validation turned on, of the Well_Formed_Xml type. By turning on validation, the Service Broker will check if the syntax of the message is well-formed Xml. If it isn't, then the message won't be send, instead an exception will be thrown. Once the *Target* database has audited the contents of the *AuditMessage,* it will send out a *//Audit/AuditReply*. The *AuditReply* can also be used to inform the *Initiator* if the *Target* has failed to audit the event.

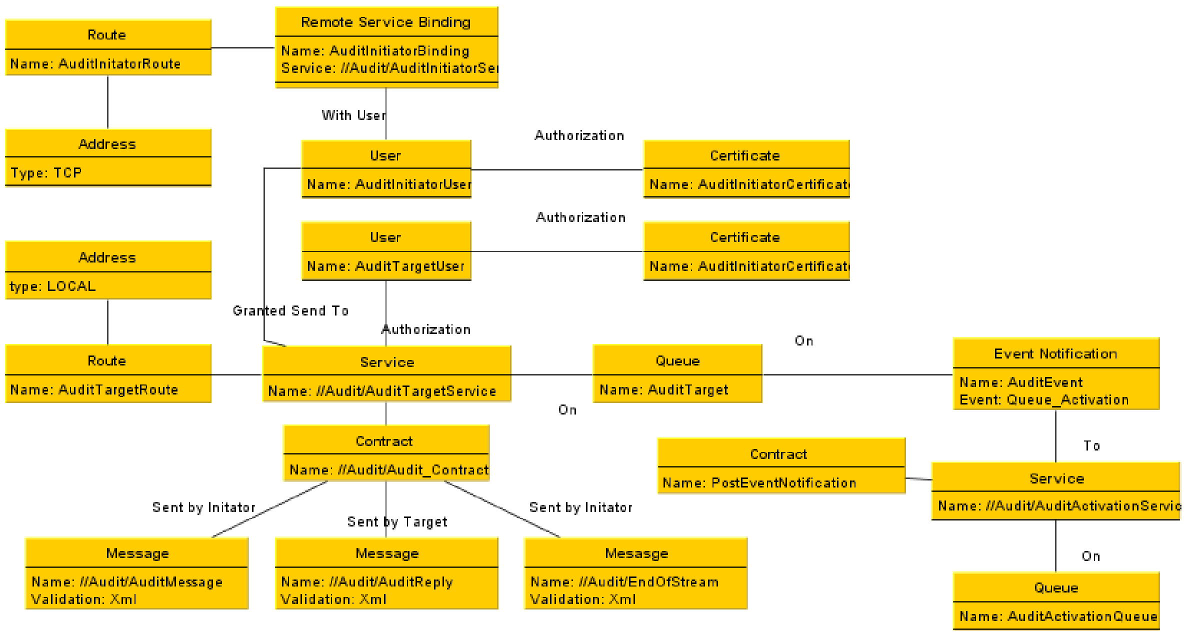

FIGURE 4.1.2 SERVICE BROKER SETUP – TARGET DATABASE

Messages are send in conversations [35]. Each conversation is a communication channel. Conversations between the two databases need to be closed at some point, as it is possible for an error to occur. Such an error will make it impossible to use the conversation further. When an error occurs, the messages which were still pending, and haven't yet been send to the *Target* database, will need to be reprocessed and send out in a new conversation. If conversations are used for a very long time, then a large backlog of unsent messages can be created. Reprocessing a large backlog will take a very long time and cost a lot of resources. Another reason for limiting the lifetime of conversations is that it makes it possible to use retention on queues. By using retention, a copy of every message send to/from a queue, during the conversation, is kept until that conversation is ended. When creating a conversation, it is possible to activate a Conversation Timer on it. This timer will fire a *DialogTimer* message once the time has expired. In response to the *DialogTimer* message, the conversation can then be closed by a stored procedure [9].

A conversation can be closed by using the END CONVERSATION command [23]. After using this, the Service Broker will disconnect the conversation on that side and ignore any further messages send on this conversation. The command should never be used first on the *Initiator* side. If you use it first on the *Initiator* side, then any acknowledgements sent by the *Target,* for *messages* which the Target has received, will be ignored. If the *Initiator* had yet to receive an acknowledgement for some of the messages which it had sent, then it will be ignoring the acknowledgements from now on. This makes it impossible to know if all the messages arrived or not, messages which did not arrive will never be resend and will thus never arrive. This is known as the fire and forget pattern and can result in message loss [8].

Instead the *Target* must be the one to initiate the End Conversation, in response the *Initiator* can then safely end the conversation on its side too. This way the conversation is closed in a safe manner, preventing the loss of audit data. The *DialogTimer* for the conversation is running on the *Initiator* side, since the conversations are started there. So when the timer finishes, a message needs to be send to the *Target* so it can close the conversation. An extra message type is required to achieve this. This will be the *//Audit/EndOfStream* message.

Now that we have all the messages, a contract needs to be created so we can define who is allowed to send which message. As shown in figures 4.1.1 and 4.1.2, this contract should state that the *Initiator* may send the *AuditMessage* and the *EndOfStream* messages, while the *Target* may only send the *AuditReply* message.

4.1.4 Queues, Services and Routes

In order to receive messages, each database needs a Queue in which the messages can be stored [20]. The *Initiator* database needs a Queue so it can receive *AuditReply* messages, while the *Target* needs a queue to receive *AuditMessage* and *EndOfStream* messages. Error messages will also be received in these queues. The queues will be named *AuditInitiatorQueue* and *AuditTargetQueue*.

In order to send messages from one database to the queue of the other database, routes will need to be defined between the databases, and each queue needs to have a service assigned to it [29]. Messages are send from one service to another, the route ensures that the message arrives in the queue to which the service has been assigned. On the *Initiator* side we first create an *AuditInitiatorService* on the *AuditInitiatorQueue, using the AuditInitiatorUser* for authorization. We do the same on the *Target* side, creating an *AuditTargetService* on the *AuditTargetQueue,* using the *AuditTargetUser* for authorization. Then we create a remote service binding in each database. In the *Initiator* database we need one remote service binding in order to bind the *Target's AuditTargetService* to the *AuditTargetUser*, while in the *Target* database we need to bind the *AuditIniatiatorService* to the *AuditInitiatorUser*. The routes need these remote service bindings, as they are used to represent the destinations for messages.

Once the services have been created, we can create the routes. On each side an *AuditTargetRoute* and an *AuditInitiatorRoute* need to be created. These routes use the services we created earlier. For the *Initiator,* the route to the *Target* needs to be created in the *Initiator* database, while a route to the *Initator* needs to be created in the *msdb* database. The same must be done in the *Target* database, but there the route to the *Initiator* should be defined in the *Target* database while the route to the *Target* database needs to be defined in the *msdb* database. See figures 4.1.1 and 4.1.2 for these routes. The Initiator and Target each need to have a route in the msdb database because this route needs to have a LOCAL destination, telling the Service Broker that the destination can be found in this server instance. When the SQL Server instance containing the *Inititator* receives a message from the Target, it will use this route to direct the message to the *Initiator*. It works exactly the same in the *Target* database.

4.1.5 Stored Procedures and Trigger

Now that the Service Broker's infrastructure is in place, we need to create a Trigger and several stored procedures, so that the prototype can handle the creation of the messages, and so the conversations can be managed.

A DML trigger [22] needs to be added to the table in the *Initiator* database, it creates the message body and then passes all necessary information to a stored procedure (which we will refer to as *audit_send)*. This procedure gives the message to the Service Broker, so messages can be send to the *Target* database. We have an extra table in the *Initiator* database, which contains all the currently active conversations, this allows us to re-use existing conversations. There is a max of one conversation per @@SPID [15]. If no conversation is found for the current @@SPID, then a new conversation is started and added to the table. The procedure then sends the message on this conversation.

This takes care of sending the messages, but on the *Initiator* side we also need to process received *AuditReply* messages and '*http://schemas.microsoft.com/SQL/ServiceBroker/EndDialog'* messages. The *EndDialog messages* are automatically send if the other side has used End Conversation on the conversation [23]. We will refer to this procedure as *Audit_ProcessResponse*. If an *EndDialog* message is received then End Conversation needs to be called on the conversation, if a *DialogTimer* message is received then an *EndOfStream* message needs to be send to the. Finally if an error has been received then the conversation on which the error was received is no longer usable. If the error is of a type from which we can recover immediately [10], then a new conversation is created and all unsent messages are reprocessed and send out again in this new conversation. If we fail to process a message then this message is moved into a Service Broker Error table, which we have created for this purpose. This way we don't have to roll back the transaction which was used to receive messages from the queue, preventing the queue's poison message handling from kicking in and shutting down the queue.

4.1.6 Service Broker External Activator

Now we are capable of holding partial conversations between the two databases, *AuditMessages* can be send by the *Initiator* database and they will arrive in the *AuditTargetQueue* at the *Target* database, but replies are not yet being sent. We need to create an internal or external procedure on the *Target* side, which can process these messages and send replies to the *Initiator*.

Since the AuditMessages are all in XML, we will need to parse them. For inserts, the old and new values of each column need to be compared, if they differ then the old value needs to be stored. The new values are discarded, as these values can be retrieved from the entity in the database. The changes are then stored in the audit table. We have chosen to do this in an external procedure, written in C#. An External Activator needs to be set up so that this application is automatically launched when a message is present in the queue.

The external activator [4] is a service, which can be used to launch an executable in response to an event in the queue. It can be configured to launch a C# application when a *PostEventNotification* arrives in a queue. Depending on the used configuration, it can let several instances of the application run concurrently.

In order to get the external activator working we first set up an extra queue. Then we create a new service on this queue, using the system defined contract for Event Notifications 'http://schemas.microsoft.com/SQL/Notifications/PostEventNotification'. Finally we create an event notification on the queue, for the QUEUE_ACTIVATION event type, to the service we created earlier.

Now an event notification will arrive in the new queue whenever an *AuditMessage* arrives in the *TargetQueue*, unless there is already an event notification present in the new queue. The External Activator will see this event notification, consume it, and launch the application. The application will then receive messages from the *TargetQueue*, add entries to the audit table and send *AuditReply* messages to the *Initiator*.

4.1.7 External Procedure

We have chosen to use set based processing [11] in the application, which is started by the External Activator, to receive messages from the queue. This is an efficient method for receiving messages. The application receives all the messages from one conversation, from the queue, and then processes them one at a time. This way we can limit how often we need to query the database for messages, as we receive those all in one go. The application doesn't use concurrency to process the messages, because multiple instances of this application can already be running at the same time, one per conversation and up to eight on our test system. Messages which cannot be written to the audit database due to an error will not be returned to the queue, as the same error would just come up again. Instead these messages are written to a dedicated Service Broker Error table in the *Target* database, and then the application continues with the next message.

4.2 Aspect Oriented Programming Prototype

The Aspect solution is written in C# and makes use of PostSharp [21]. The aspects in this prototype are used to extend the methods which are responsible for making Inserts, Updates and Deletes to the *Initiator* database. They gather information about these events and then send this audit information to another application, which is running on the same system as the *Target* database. We use three aspects for this, an *InsertAuditAspect*, a *DeleteAuditAspect* and an *UpdateAuditAspect*. All three inherit from AuditAspect, which inherits from PostSharp's OnMethodBoundaryAspect. The OnMethodboundaryAspect can be used to add extra functionality to a method, Figure 4.2.1 gives a good example of how it wraps a method. It has four different areas where functionality can be added. By overriding OnMethodBoundaryAspect's OnEnty method you can add functionality before the start of the target method. By overriding OnExit you can add functionality after the method has exited. There are also the OnSuccess and OnException methods, which can be used to add functionality when the method exits without an exception, or when it exists with an exception. Either way the functionality in those two methods will be executed before the functionality in the OnExit method.

```
int WrappedMethod()
{
  OnEntry();
  try
  {
    // Original method body.
    OnSuccess();
    return returnValue;
  }
  catch ( Exception e )
  {
    OnException();
  }
  finally
  {
    OnExit();
  }
}
```

FIGURE 4.2.1 WRAPPED METHOD

AuditAspect contains methods used by all three of the aspects and has access to an AspectAuditCommunicator. The AspectAuditCommunicator is used to send audit messages to the C# audit application. The audit application then writes the audit information to the *Target's* audit tables.

In this prototype, each Insert/Update/Delete happens in its own transaction, thus once the transaction has been successfully committed, we can audit the event in question. In other systems it could also happen that multiple actions take place during one transaction, in this case audit messages should not be sent until the transaction in question has been successfully committed. In that case an extra Aspect can be used to monitor the method used to commit a transaction.

4.2.1 AuditAspect

The AuditAspect has functionality which the other aspects use. Amongst this functionality is a method which can parse an entity and turn it into XML. Currently this is done through reflection. By using the *GetProperties()* method we can retrieve an array, containing all the *PropertyInfos* belonging to the type of the entity. We can then retrieve the values and names of each PropertyInfo, making it possible to process the object without knowing anything about it. Using reflection in this way is actually quite slow [2], but we can take this performance cost into account when testing the performance of the prototype, since the overhead of the used reflection is known.

4.2.2 InsertAuditAspect
The *InsertAuditAspect* uses the *OnSuccess* method. It retrieves the inserted entity, and uses the retrieved information to constructs an audit message. It then gives this message to the AspectAuditCommunicator, which then send it out. Since OnSuccess is used, we don't need to wait for the transaction to be committed as it already has been. Thus the message can be send out immediately.

4.2.3 DeleteAuditAspect
The *DeleteAuditAspect* uses the *OnEntry* method, retrieving the entity from the database just before it is deleted. It then constructs an audit message and stores this for later use. It uses the OnSuccess method in order to send the stored audit message once the transaction, in which the delete action occurred, has been committed.

4.2.4 UpdateAuditAspect
The *UpdateAuditAspect* uses the OnEntry method, retrieving the entity from the database just before it is updated. It then stores this entity for later use. It also uses the OnSuccess method, here it retrieves the updated entity. The updated entity is then compared to the entity which was retrieved by the OnEntry method. The changes are put in the audit message, which is then sent to the AspectAuditCommunicator.

4.2.5 AspectAuditCommunicator and Audit application
Now that we have an audit message, it needs to be inserted into the audit database. It is possible to establish a direct connection to the *Target* database and do the auditing directly from the test application, but this would make the aspect prototype block a lot longer than necessary and would skew the test results in favor of the Service Broker.

The *AspectAuditCommunicator* uses a *TcpClient* to connect to the Audit application. The messages created by the AuditAspects are then send to the Audit application, where they are moved into the audit table using Parallel.Foreach, to allow concurrent processing of the audit messages. Message loss can occur as the process isn't entirely reliable. Messages which have arrived in the Audit application are stored in memory only, so a crash, or other memory issues, can result in message loss. This is not an issue for the actual tests which we will be running, and implementing an entire reliable messaging system is beyond the scope of this research.

4.3 Hybrid Prototype
The hybrid prototype uses the exact same Service Broker infrastructure as the Service Broker prototype, with the exception of the Trigger. This trigger has been replaced with several aspects, using PostSharp.

4.3.1 Insert, Update and Delete Aspects
These three aspects are similar to the aspects mentioned in 4.2.1, 4.2.2 and 4.2.3. The exception is that they do not make use of the AspectAuditCommunicator. They just gather the data related to the entity. This information is then stored in an AuditWork object, which is managed by the AspectTransactionManager class.

4.3.2 AspectTransactionManager
The AspectTransactionManager manages the AuditWork object created by the Insert, Update and Delete Aspects. These work object are stored in a dictionary, where the key is the Transaction in which the Insert, Update and Delete events happened. This allows the TransactionAspect to aquire all the work items related to one transaction.

4.3.3 TransactionAspect

The TransactionAspect intercepts the *Commit()* method. Just before a transaction commits it will request all auditWork items related to the transaction from the AspectTransactionManager. Those work items contain all the audit information about the actions, which needs to be send to the *Target* database for auditing. The *audit_send* stored procedure, mentioned in 4.1.5, is then called. It sends this audit information, in an *AuditMessage,* to the *Target* database. The TransactionAspect makes sure that these messages are sent in the same transaction as the actions. If the messages were not sent out in the same transaction, then you'd run the risk that the original events were rolled back, but that the messages were still sent out. The *Target* database would then contain audit information for events which didn't occur. Messages remain in the *transaction_queue* until the transaction has been commited, only then are they sent out.

5. Test Setup

Now we will discuss the tests which we will use to test the performance of the chosen solutions. First we discuss the test setup and the environment in which the tests happened, then we will discuss the tests which we ran. In chapter six we will discuss the results of these tests.

5.1 Test Environment

Two SQL Server instances are used for the test, one contains the *Initiator* database while the other contains the *Target* database. The *Initiator* database contains a Table named '*TestTable*'. This table is used during the tests. Inserts, updates and delete actions are done on it. This table has 20 different columns. It has a bigint for a Primary Key, it has two DateTime columns, five nvarchar(255) columns, four int columns, three decimal(19,5) columns, two bit columns, one bigint column, one smallint column and one float column. All of these columns are allowed to be null.

During some of the tests, both SQL Server instances are present on the same system. During other tests a secondary system is used, the *Target* database is then present on the second system. The primary system always houses the *Initiator* database. These systems have the following specs:

Primary test system:

- OS: Windows 7. 64 bit
- Memory: 8gb ddr3 – 1333 Mhz
- CPU: Intel Core i7-2720 QM 2.2 Ghz
- Harddisk: 7200rpm

Secondary test system:

- OS: Windows 7. 32 bit
- Memory: 4gb ddr2 – 1066 Mhz
- CPU: Intel Core 2 Duo E7400 2.8 Ghz
- Harddisk: 7200rpm

5.2 Performance tests

The test used to measure the performance takes three different arguments. The first is the amount of rows which should be present in the *TestTable* before we start testing. The second is the amount of insert actions which are to be done as part of the test. For each of these insert actions, five update actions will also take place.

5.2.1 Test process

All inserts and updates were done while using Parallel.Foreach, which allowed the test to push IO usage as far as possible. At the start of the test, rows are first inserted or removed, depending on how many rows are present, compared to the amount which was given as the first argument. Once the desired amount of rows are present in the table the test starts by inserting the amount of rows, specified by the second argument, into the table. An aspect extends the method which inserts the rows. This aspect keeps track of how much time elapsed while inserting the row. Once all the rows are inserted, the aspect adds the recorded times together, and divides this total by the amount of inserted rows. This gives us the average amount of time per insert action.

Each row is inserted with each of the columns having a value, none of them are null. Once the rows have been inserted, each row will be updated five times. During the first update an nvarchar, an int and a DateTime column are modified. During the second update the earlier three columns are again modified, but now the other nvarchar columns are also modified. During the third update the previously modified columns are all updated again, except for three of the five nvarchar columns. The other 2 int column are also modified, as are two of the Decimal columns. During the fourth update the same columns as in update 3 are modified, the second DateTime column is also modified, the float, smallint and last int column are all modified as well. During the final update the columns from update 4 are again modified, but now the last decimal column, both bit columns and the bigint column are also modified.

An aspect keeps track of how much time elapses per update. Once all updates are done, it calculates the average time in miliseconds per update action. At the end of the test, the average time per insert action and the average time per update action are both given.

5.2.2 Test versions

Two types of tests were run. The first type has both server instances present on the same system, while for the second type the *Target* database was moved to the secondary test system. This allows us to investigate how much of a difference it makes to have the audit database on a dedicated system. Two different tests were done of the first type, while one was done of the second type. These tests differed in the initial amount of rows in the table and in how many inserts and updates were done. Table 5.2.2.1 gives an overview of the three different tests.

Type of test	Same system		Two systems
Initial amount of rows	0.5 million	1 million	10 million
Amount of inserts per test	10.000 inserts		100.000 inserts
Amount of updates per test	50.000 updates		500.000 updates

TABLE 5.2.2.1.

Each of these tests were first run ten times on a database without any form of auditing, then ten times on the service broker prototype, then ten times with the aspect prototype and finally ten times with the hybrid prototype. This allows us to determine the overhead of each solution, compared to the situation where no auditing was done. It also makes it possible to compare Triggers to Aspects, as the only differences between the Hybrid and Service Broker solution are the aspects and triggers. As mentioned in 4.2.1, some very slow reflection was used in the aspect prototype, and thus also in the Hybrid prototype. The total overhead of this reflection comes to 0.048ms per update action and 0.024ms per insert action. These values have already been deducted from the test results in the next chapter.

6. Performance Test results

In this chapter we give the results for the tests which were described in chapter five. We then discuss the results, and we discuss which of the two prototypes we believe is best suited for Topicus.

6.1 Results

The results of the tests are given in the following four tables. One table per tested prototype and one table with the results from the test where no auditing was used at all.

	ms per inserts			ms per update		
Test #	0.5 million	1 million	10 million	0.5 million	1 million	10 million
1	0,1731	0,1383	0,15133	0,13356	0,13322	0,17252
2	0,1497	0,1249	0,18257	0,13728	0,1202	0,23877
3	0,131	0,1551	0,22616	0,1925	0,12898	0,267682
4	0,1856	0,1265	0,19908	0,14644	0,12194	0,266592
5	0,1326	0,1246	0,23527	0,1348	0,12996	0,271292
6	0,1178	0,1473	0,30271	0,13254	0,15286	0,24663
7	0,1482	0,1302	0,32566	0,12814	0,15154	0,271184
8	0,1731	0,1589	0,3182	0,14598	0,16258	0,26046
9	0,134	0,1295	0,30535	0,14232	0,11926	0,278282
10	0,1058	0,1279	0,2933	0,1403	0,13932	0,262174
average	0,14509	0,13632	0,253963	0,143386	0,135986	0,2535586

TABLE 4.1.2.1 – NO AUDITING

	ms per inserts			ms per update		
Test #	0.5 million	1 million	10 million	0.5 million	1 million	10 million
1	1,3993	0,8431	0,40301	0,98196	0,94098	1,00253
2	0,6614	0,6269	0,43689	1,53138	1,34684	0,858458
3	0,5909	0,5303	0,71955	1,27308	1,4819	0,87227
4	0,428	1,2164	0,5519	1,2434	1,34726	0,729856
5	0,5887	0,8969	0,5306	1,2473	1,28084	0,688166
6	1,2199	1,2992	1,01958	1,03504	1,16008	0,827318
7	0,4789	0,624	0,3474	1,53806	1,15506	0,64351
8	0,7296	0,6644	0,64597	1,16992	1,48624	0,656628
9	1,01672	0,756	0,57327	1,43406	1,57094	0,74114
10	0,9232	0,7576	0,61123	1,33288	1,41272	0,88808
average	0,803662	0,82148	0,58394	1,278708	1,318286	0,7907956
overhead	0,658572	0,68516	0,329977	1,135322	1,1823	0,537237

TABLE 4.1.2.2 – SERVICE BROKER

	ms per inserts			ms per update		
Test #	0.5 million	1 million	10 million	0.5 million	1 million	10 million
1	0,4787	0,4446	0,16304	0,51148	0,7699	0,366566
2	0,1805	0,1472	0,3734	0,63246	0,6486	0,510022
3	0,1569	0,2186	0,39441	0,628	0,57552	0,691478
4	0,1903	0,1615	0,61309	0,61844	0,66922	0,9236949
5	0,2049	0,2997	0,49296	0,63216	0,6372	0,709482
6	0,2006	0,2468	0,44022	0,79728	0,6129	0,64079
7	0,3318	0,158	0,55597	0,72108	0,63506	0,90756
8	0,2615	0,2426	0,45724	0,72046	0,6941	0,712494
9	0,2178	0,3016	0,28062	0,713	0,60788	0,629858
10	0,2879	0,141	0,35486	0,73668	0,58538	0,567624
average	0,25109	0,23616	0,412581	0,671104	0,643576	0,66595689
overhead	0,106	0,09984	0,158618	0,527718	0,50759	0,41239829

TABLE 4.1.2.3 – ASPECTS

	ms per inserts			ms per update		
Test #	0.5 million	1 million	10 million	0.5 million	1 million	10 million
1	0,789	0,7535	0,51568	1,15136	1,18706	0,75453
2	0,5532	0,6498	0,44102	1,3386	1,39654	0,76538
3	0,8598	0,606	0,50768	1,212	1,27694	0,75177
4	0,8869	0,7149	0,30535	1,31704	1,20164	0,7153
5	0,8075	0,5619	0,42149	1,29912	1,09862	0,748262
6	0,7846	0,814	0,75535	1,1562	1,17654	0,779508
7	0,7723	0,884	0,38009	1,33236	1,23974	0,664162
8	0,7933	1,0943	0,46496	1,33228	1,41124	0,668422
9	0,6039	0,7654	0,39513	1,18314	1,28642	0,723832
10	0,6093	1,0435	0,40796	1,18778	1,17654	0,748294
average	0,74598	0,78873	0,459471	1,250988	1,245128	0,731946
overhead	0,60089	0,65241	0,205508	1,107602	1,109142	0,4783874

TABLE 4.1.2.4 – HYBRID

Since the amount of updates per test is five times greater than the amount of inserts, the accuracy of the update results is greater than that of the insert results. This is why we will mostly refer to the time per update action when discussing the results.

6.2 Discussion

With an initial 0.5 million rows, the overhead for the Service Broker prototype is around 1,14ms per update action, while the Aspect prototype has an overhead around 0,53ms and the Hybrid has around 1,11ms per update action. The Service Broker and Hybrid are almost equal performance wise, with the Aspect version being quite a bit faster.

With an initial 1 million rows, the overhead of the Service Broker rose slightly too around 1,18ms per update action, while the overhead of the Aspect prototype dropped slightly to 0,51ms and the Hybrid remained at 1,11ms per update action. These results are really close to that of the 0.5 million tests.

The results of the 10 million rows tests, when using two different systems, shows a huge reduction in overhead for al solutions, but especially for the Service Broker and Hybrid prototypes. The Service Broker dropped down to 0,54ms per update action. The Hybrid dropped to 0,48ms per action. This is more than a 50% drop in overhead for both of those prototypes. The Aspect prototype on the other hand only drops to 0,41ms per action, which is around 10% a reduction in overhead.

During both the 0.5 million and 1 million test runs the aspect prototype clearly performs better than the service broker prototype. This can mainly be explained by the lack of a proper message handling system, as switching from a Trigger in the Service Broker to using aspects in the Hybrid only results in a difference of around 0,05ms. The Service Broker assures the delivery of all messages, each message is inserted and deleted from several queues during its lifetime. Each queue is backed by an internal table, so effectively each message is being inserted and deleted from several tables before auditing is completed. With the aspect solution, the message is just send by TCP to the audit app, where it remains in memory until it’s inserted into the audit table. With a proper, reliable, messaging system the overhead of the aspect prototype is likely to increase quite a bit.

During the 10 million test run, all prototypes are very close to each other overhead wise, with the Service Broker having only 0,13ms more overhead per action than the Aspect prototype and the Hybrid only having 0,07ms more overhead than the Aspect prototype. The difference smaller than we had expected and shows that the extra overhead of the Service Broker is quite small. The Aspect prototype used TCP to directly send the message while the Hybrid had the same aspects, but instead used the Service Broker. An overhead cost of 0,07ms for such a reliable messaging system is lower than we had expected.

The overhead difference between the tests using one system, and the tests using multiple systems, show that a lot of performance can be won by auditing the events on a different system. The prototypes, which used the Service Broker, benefit a lot and the Aspect prototype will likely show similar results when used with a proper reliable messaging system.

During the tests it was noticeable that messages were being audited as fast as they were being received by the *Target* table. The auditing had no problem keeping up with the ~900 messages it was receiving per second during the Service Broker and Hybrid tests on one system. This was what we had expected to see, as the set based processing we used was able to handle up to 2000 messages per second in other tests [11]. One of their techniques even got it up to 2700 messages per second, so with optimization we should be able to reach more. But even at 2000 messages per second, one dedicated audit system, using the service broker, could process 173 million messages per day.

6.3 Audit Solution Conclusions

From these tests we can conclude that Triggers are indeed slower than Aspects, as we assumed in 3.3, but the overhead difference is less than 0,1ms. Storing the audit information in a different server, on a different system, results in a large reduction in the overhead, more than halving it with the service broker and hybrid prototypes. As a reliable messaging system, the Service Broker performs well, it introduces a low amount of overhead and is very reliable, maintainable and portable. We have updated table 3.7.1, which results in the following table:

	Functional Suitability	Efficiency	Reliability	Maintainability	Portability	Total
SB	++	++	+	++	++	11.5
AoP	+	++	+	++	++	10

TABLE 6.3.1

The overhead of the Service Broker and Aspects was so close that no adjustment to their Efficiency scores was required. This leaves the Service Broker as the better scoring solution. As mentioned in 3.1.2, the application side solutions do have the disadvantage that they cannot audit events which are done directly on the database, as these events do not take place in the application where the audit solution is deployed. The AoP solution does not manage to make up for this disadvantage in another area. Its advantage in overhead is far too small to compensate for this disadvantage. This also answers our question on if there are alternative auditing solutions which meet Topicus' needs, and are compatible with Microsoft SQL Server or Topicus' applications. Both the AoP and Service Broker solutions are suitable alternatives, with the Service Broker, using Triggers, being the better of the two.

Thus the most suited auditing solution for Topicus is the Service Broker, using triggers. We advise to use a separate server for storing the audit information, as this greatly reduces the overhead, and to use the Service Broker as a reliable messaging system to prevent the loss of audit data.

7. Reverting Entities

In this chapter we investigate if any strategies exist for using audit data to revert specific entities in a database to an earlier version. Then we discuss and define our own strategies. Finally we discuss the issues which were encountered while implementing some of these strategies.

7.1 Existing implementations

We searched for existing implementations, but to no success. We did find one other person who was looking to create a similar system, also by using the audit data [1]. He was asking for resources specific to this issue, but he was not given any. We searched for “database row reversion”, “database row rollback”, “database entity reversion”, “database entity rollback”, “fine grained rollback” and “database versioning”, but most results discussed rolling back the last transaction, or rolling back the entire database, to a certain point in time, or to a certain checkpoint. So existing implementations do not seem to exist for this.

7.2 Entity reversion considerations

When reverting an entity, there are several cases which should be considered in order to create a working strategy. The simplest case is a single entity, which does not have any foreign keys to other entities. A more complex case can involve foreign keys from the entity we wish to revert, to other entities, which we shall call dependencies. Even more complex cases involve shared dependencies and looping foreign keys.

7.2.1 Single entity

An entity, which is not involved in any foreign key relations, can be inserted, updated and deleted without breaking any foreign key constraints. If we want to revert such an entity to a desired time, then we need to retrieve all the audit entries of this entity, which happened after the given time. These audit entries can be used to determine which columns need to be modified, and which values they need to be given. Then we need to check if the entity currently exists in the database and if the first audit entry, after the desired time, is an insert action. If the first audit entry is an insert action, then the entity didn’t exist at the desired time as it was inserted later than that. This means that the entity should be deleted if it currently exists, as the desired version of this entity, at the given time, is that it doesn’t exist. If the entity doesn’t currently exist, then we don’t have to do anything as the current version is equal to the desired version. If the first action after the given time is not an insert, and the entity currently doesn’t exist, then we need to insert it, else we need to update it. Once this is done, the entity has been reverted to the version which it had at the desired time.

This case becomes more complex if one incorrect update was done to an entity, followed by two correct updates. For example, if we have an Insured Person entity and the first update incorrectly changed his name, while the second update correctly updated him from single to married, while the third changed his address. If we want to undo the incorrect update, by reverting the entity back to just before this update was done, then the correct updates are also undone. So instead of just reverting the Insured Person to a previous version, a user might want to have the option to revert specific actions between now and the desired time. This way only the incorrect update can be reverted, without losing the changes made by the correct updates. If the incorrect update modified several columns, say both the name and the address, then reverting just the incorrect update would still undo some of the changes made by the correct updates. So another option to consider is, if the user should have the ability to choose exactly which columns will be reverted, and which will not be. This way the name change can be reverted without also reverting the address change.

7.2.2 Foreign keys to other entities

A more complex case involves foreign keys. For example, if we have a structure like in figure 7.2.2.1. Here we have two different tables, one table contains a list of insured people, and the other contains a list of insurance policies. In figure 7.2.1 we have one insured person, whom has a policy. By accident his policy was replaced by another policy. Now we want to undo this mistake, by re-inserting the old policy, changing the person's foreign key so it links to the old policy instead of the incorrect policy, and deleting the incorrect policy.

This mistake can be reverted in several ways. One way is to revert the entities one at a time. First reverting the deletion of the old policy, then reverting the changes made to the Insured Person (changing the foreign key) and then reverting the insertion of the incorrect policy, deleting it. Doing it like this requires the user to investigate, by digging through the audit data, which entities he has to revert. The old policy was deleted, so the user can only find mention of this policy by digging through the audit data of the insured person. He can find the value of the foreign key in audit data. By using this value he can then find the old policy in the audit table for the insurance policies. This can be a lot of work and is error prone. The user must also be careful in which order he does the actions. He can't first delete the old policy, as this would cause a foreign key constraint violation. So a more automated approach could be desired.

An example of such an automated approach would be to also revert the dependencies of the entity. A limit can be put in place on how deep this can go, or the user could be allowed to define the maximum depth. If the user tells the system that he/she wants to revert the Insured Person entity to an earlier moment in time, then the system checks which foreign keys this entity has, if any. One foreign key would then be found, which leads to the incorrect policy. If the maximum depth isn't set to one, then the system then repeats this for the incorrect policy, but doesn't find any new foreign keys. Then the system retrieves the audit data for both of the found entities, and determines what modifications it needs to make to them, if any. From these modifications, it notices that the foreign key of the Insured Person needs to be changed. It can then use the desired value for this foreign key to retrieve the old policy from the audit database. By using the audit data it can then determine that the old policy needs to be inserted, the incorrect policy deleted and that the insured person needs to be updated. It will do the inserts first, then the updates and then the deletes, this way foreign key constraint violations can be prevented.

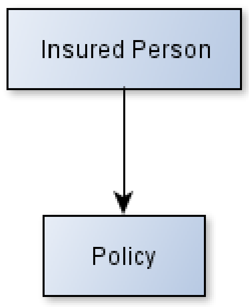

FIGURE 7.2.2.1

This example can be made more complex. The insured person can also have a foreign key to some other entity. This entity was modified after the policy change was done, and the changes to this other entity are correct and should not be reverted. If we use the above approach to also revert the dependencies of the entity which we are reverting, then these correct changes would also be lost. This returns us to the discussion in 7.2.1, where we mentioned that the user might want to have more control over what is reverted, like being able to choose which columns are reverted, and in this case, also which entities.

7.2.3 Entity we wish to revert is a dependency

Another case we should consider is what we do if the entity we try to revert, is a dependency of another entity. Take Figure 7.2.2.1, but this time we don't want to revert the insured person, but just the policy. If the desired time given by the user, would revert the entity back to a point where it didn't exist, then the policy would be deleted. The policy is a dependency of another entity, which the policy knows nothing about. Deleting the policy will cause a foreign key constraint violation, which will cause the reversion to fail.

One option would be to throw an error when this case is detected. For this to work we first need to determine if there is an entity which has this policy as a dependency. If such an entity exists then the error can be thrown, informing the user of the problem, and then halting the reversion.

Another option is to automatically revert all the entities which have this policy as a dependency, and to revert all dependencies as mentioned in 7.2.2. Let's use the example from 7.2.2. The insured person has a policy, which was deleted and replaced by the current, incorrect policy. Reverting this incorrect policy to before its insertion will delete it. If we also revert the entities which have the policy as a dependency, then the person is also reverted to this time, changing its foreign key so it links back to the correct policy. The correct policy is re-inserted in the process, as we are also reverting all the dependencies. This way the end result is the same as in 7.2.2. With this approach it doesn't matter which entity in the relational structure you revert, the outcome will be the same.

7.2.4 Foreign keys with shared dependency

Now we consider a more complex case of shared dependencies. We once again have an insured person with a policy like in figure 7.2.2.1. This time we want to add a second insured person, with another policy. But by accident this goes wrong. Instead of adding a new policy for this second person, the policy of the first insured person is modified and given the values which the second policy was supposed to have. The foreign key of the second person is then linked to this policy. From the point of view of the second insured person, everything looks fine, but everything is now wrong from the point of view of the first insured person. Figure 7.2.4.1 shows how the relations changed.

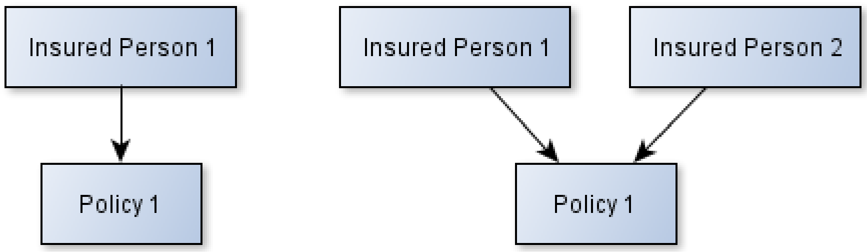

FIGURE 7.2.4.1

There are several ways we could revert this mess, we will now discuss two. We can just revert Insured Person 1 and automatically revert its dependencies, or revert just Policy 1, as Insured Person 1 wasn't even changed. This sets things right from Insured Person 1's point of view, as the incorrect modifications are reverted, but Insured Persons 2's policy is now incorrect. The damage from the mistake is restored and that was the main goal. Insured Person 2 can then be reverted to before it was linked up to Policy 1, and then the action of giving it a new policy can be done correctly this time around.

Another possible option is to detect that Policy 1 is the dependency of another entity, and to then create a copy of Policy 1 while leaving the original intact. This way Insured Person 2 can remain linked to Policy 1, while Insured Person 1 can be linked up to a copy of Policy 1. This copy can then be reverted to the version of Policy 1 before the incorrect modifications were done to it, with the exception of the primary key value. This way of fixing the mess will work in simple cases like these, but can cause issues in more complex cases. For example what if Policy 1 has dependencies of its own. Should copies also be made of these dependencies, or can the copy have foreign keys to the same dependencies without any issues?

This example can be made more complex. For example, as in figure 7.2.4.2, what if Policy 1 wasn't just incorrectly changed, but was actually deleted. A new policy was added for Insured Person 2 and Insured Person 1 was also linked up to it, after Insured Person 2. Policy 1 should not have been deleted and Insured Person 1's foreign key should not have been changed. In this case, reverting Insured Person 1 and all its dependencies won't work, as this will delete Policy 2 and cause a foreign key constraint violation for Insured Person 2. This constraint violation can be prevented by checking, for every entity which we want to delete, if it is a dependency of another entity. If this is the case then don't execute the delete action, instead just leave the entity untouched.

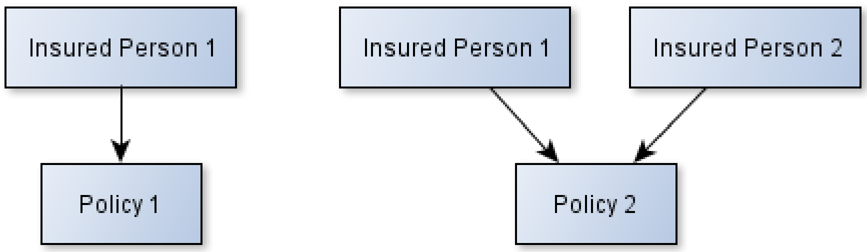

FIGURE 7.2.4.2

By not executing the delete action, the final result of this reversion is that Policy 1 is re-inserted, Insured Person 1 has its foreign key changed to link it back to this entity, and Insured Person 2 is still linked to an unchanged Policy 2.

7.2.5 Foreign key loops

The last case we consider is when foreign keys cause a loop. If we need to re-insert entities which contain a loop, then no matter where we start we will always violate the foreign key constraints. Two ways by which this issue can be solved are the following; temporarily disable the foreign key constraints, and splitting up the insert action into several actions.

Foreign key constraints can be temporarily disabled. Once they are disabled, we can safely do all the insert and delete actions which would otherwise violate the constraints. Once the actions have been done we can re-enable the foreign key constraints. This approach could have unforeseen side-effects, if another applications does an incorrect action before the constraints are re-enabled.

The looping actions could also be split up into multiple actions. For example take three looping insert actions. If even one of the three involved foreign keys is allowed to be null, then the loop can be broken by first inserting this entity with a null value for that column. Once all three entities have been inserted, the column can safely be updated to the correct value. For a delete, a foreign key first need to be updated to null, this breaks the loop and makes it possible to delete the entities without causing a foreign key constraint violation. In both cases, in order to break the loop, at least one foreign key must be allowed to be null. If none of them are allowed to be null, then the only option is to disable the foreign key constraints.

7.3 Reversion strategy examples

We have identified several points where the desired course of action can differ. Depending on the situation a different approach might be desired, so a modular implementation, which can support these alternatives, would be preferable. The strategies can differ when determining which entities need to be changed, and how they need to be changed, as discussed in 7.2.2, 7.2.3 and 7.2.4. The strategies can also differ in how they execute the insert and delete actions, as can be seen in 7.2.5. Some approaches add extra update actions after inserts or before deletes in order to prevent foreign key constraint violations.

We have created some strategies by using the approaches mentioned in 7.2, but many others are of course also possible. These are merely examples. One strategy, which we shall refer to as 'Full Reversion', is to always revert the entire relational structure connected to the entity, so all its dependencies, all their dependencies and all entities which have any of these entities as a dependency, and so on. This takes care of any dependency issues as all entities are reverted, but it greatly restricts you in your freedom. No matter which entity you revert, all the other entities are also reverted.

Another strategy, which we shall refer to as 'All Dependencies' is to only revert the given entity and its dependencies, and their dependencies, until no new entities are found. So no maximum depth is used here. If one of those dependencies is shared by an unrelated entity and we want to update the dependency, then we make a copy of it, so that the unrelated entity is not affected. If we want to delete this shared dependency, then the delete action is not executed. If the entity we wish to revert will be deleted by this action, and it is a dependency of another entity, then we will not allow the reversion to complete, instead an error will be raised.

Another strategy is to only revert the given entity, we shall refer to this strategy as 'Single Entity'. Its dependencies are not reverted, thus the action cannot be completed if it is missing a dependency after the change, as mentioned in 7.2.2. As in the previous strategy, an error will be raised if we try to delete this entity while it is a dependency of another entity.

We can identify several steps which are used in all these strategies. First the foreign key relations need to be mapped, as a lot of reasoning will need to be done about the relations. Then we need to identify which entities need to be changed, and how they need to be changed. Audit data needs to be retrieved for this. It is at this point that the last two strategies determine if the target entity needs to be deleted and if it is a dependency of another entity, raising an error if both are the case. Then we need to define the actions for each entity, these are update, insert and delete actions, or a combination of these actions if a foreign key loop is found. Finally we need to execute these actions on the database. The parts where the implementations of the above strategies differ are; determining which entities need to be changed and how they need to be changed, including if they may be executed or if an error should be raised, and defining which actions are required in order to revert the entities. As discussed in 7.2.2, it can also be desired to give the user more control over exactly what is reverted, as you might only want to revert one specific action, or you might want to exclude several columns from the reversion process so that they aren't affected. This could take place during the step where we define the actions for each entity.

7.4 Entity reversion remaining issues

We have implemented two strategies mentioned in 7.3, the 'Full Reversion' and 'All Dependencies' strategies. For the second strategy, we decided to make a copy of a shared dependency if we try to update it, as discussed in 7.2.4. We successfully reverted a multitude of entities and different relation structures. While implementing the strategies we came across two not yet discussed issues.

Take a look to the right side of figure 7.2.4.1. If we try to revert Insured Person 1, then in both strategies we will also revert its dependency, in this case the policy. In both strategies we must also check if the policy and Insured Person 1 are dependencies of other entities. At this point in the implementation we don't yet know that Insured Person 2 exists, we've only found Insured Person 1 and Policy 1. We do know that there's a foreign key between the Insured Person and Policy tables. We also know which column, in each table, is used for the foreign key, and we know the value of this column. So in order to determine if Policy 1 is a dependency of other entities, we can use this information to search through the Insured Person table. This will give us Insured Person 1 and Insured Person 2 as entities which have Policy 1 as a dependency. If we are using the strategy where we revert everything related to the given entity, back to a certain time, then depending on the chosen time we might also need to revert entities which had Policy 1 as a dependency and are currently deleted, and thus need to be re-inserted. To find out if such a deleted entity exists, we have to search through the audit information.

The latter is where a potential issue is located. Depending on how the audit information is stored, it can either be very easy to determine if such an entity existed, or very hard. For example, if we want to limit the space consumed by the audit data, then we could decide to store the changes of an action in one column, for example by storing them in an XML format. If we want to find the specific column, with the known foreign key value, then we need to search through the XML data for each action which occurred after the desired time. This means that we potentially have to search through millions of XML objects. This will negatively affect performance, and due to this, reverting an entity can take a long time. On the other hand, if the audit table contains complete copies of the old versions of the entities like with Change Data Capture's audit tables, then we can directly query the audit table for all rows with the given column and value. If a strategy is used where this can be necessary, then an investigation should be made in order to determine which audit table format works best, from both a disk space and performance point of view.

Another issue is that column names in the database can be modified. If this happens, then the audit data pre-dating this modification can no longer be mapped back to that table. Attempting to revert entities in this table, to a time before this modification, will likely fail or produce incorrect results. The same issue occurs when deleting columns, and also when adding new columns which may not be null. Thus an investigation needs to be done into auditing alterations on a table, and these changes then need to be taken into account when reverting an entity.

7.5 Entity Reversion Conclusion

So several complex cases have been found and discussed, from these we have created at least three choices for strategies, but many more are possible. It is clear that we can use the audit information to revert entities to an earlier version, as long as column names are not modified, no columns are deleted and no new columns, whose values may not be null, are added. Further is required in order to remove those obstacles.

We don't give advice on which strategy to use, as the desired strategy/approach depends on what your desired goal is, and is very much project dependent. For some projects, multiple strategies might even be desired, so we do advise to aim for a modular and re-usable design in which multiple strategies can be implemented alongside each other.

8. Audit Table Designs

The design of the audit tables is very important. It will have both an effect on how much disk space is required to store the information, and on the performance of applications which need to use this audit data. As discussed in 7.3, the used table format can have an impact on the performance when trying to revert entities. We have investigated several audit table designs and now discuss their advantages and disadvantages. In depth testing to determine the actual disk space usage and performance impact of each design has not been done.

8.1 Complete row copies

Topicus currently stores complete copies for rows in their audit data. In this design, whenever a row is modified, a copy of the old version is stored in the audit table. If a row is inserted then you also insert a copy of it into the audit table, when you delete a row then the old version is also stored in the audit table. The rows in the audit table also contains extra columns, like which action took place, when it took place, and which user is responsible for them. In this design, every audit table will be different as it must mimic the format of the audited table. Even columns which didn’t have their data changed will be stored after an action, thus a lot of unnecessary data is stored and disk space is wasted. Change Data Capture also uses this design for its audit tables.

8.2 Changes in XML

Another way to store the changes, made by an action, is by putting storing all the changes in one XML file, and storing this in a column [7]. This way the same table format can be used to store audit data for any entity, as these tables are not dependent on the format of the entity. Any changes made to the audited table, like column renames, will not require the modification of the audit table’s format like in 8.1. The XML data only has to contain the values of columns which have actually been changed, and only the old values are needed, as the current values can be found inside the audited table. The exception are insert and delete actions, there the current values need to be stored so an entity can be re-inserted if it has been deleted. This design we can greatly reduce the disk space required to store the information, because no unnecessary information is stored. In return it is harder to access the change data and more costly to retrieve all the changes.

8.3 Column based auditing

The idea of column based auditing is to store an audit entry for each changed column [6] [38]. Such an audit table can contain the primary key of the entity, the name of the table, the name of the column, when and by whom the column was changed, what kind of action it was and the old value of the column, as the new value can currently be found in the database. This design, just like 8.2, has the advantage that unchanged columns aren’t stored, only information on changed columns is stored. The value of the changed columns should be stored as a varchar, this way the audited table also isn’t dependent on the format of the audit table. This design also has a disadvantage, if a lot of columns were changed by one action then a lot of rows are inserted into this audit table and each of those rows contains the metadata on what kind of event happened, when it happened, etc. This results in a lot of duplicate data, potentially completely undoing any disk space we saved by not including the unchanged columns.

8.4 Two audit tables

Another option, which also isn't dependent on the format of the audited table, is to slightly modify the design proposed in 8.3. In 8.3 we mentioned that a disadvantage of the design is that a lot of metadata, like who is responsible for the change and when it occurred, is stored multiple times if more than one column was modified in the action. By using two audit tables [37], one containing the metadata around the event and the other containing the actual changes. By using two tables, the duplicate data disadvantage is removed. The first audit table contains information about the action itself, like which kind of action it was, when it happened, who is responsible for the action, which database the entity is from and which table it is from. This table needs a primary key, which can for example be called *Event_ID*. The second audit table only contains information about the columns that were changed by the action. Each row represents one changed column, thus if 5 columns were changed by one action then 5 rows need to be inserted into this audit table and one row with metadata into the other table. This design does still retain the disadvantage that a lot of rows need to be inserted if a lot of columns were changed. Resulting in more rows than in the first two designs.

8.5 Discussion

Of the four designs, the first is a naïve design, the audit tables are simply copies of the audited tables, with some extra metadata columns added to record when the change occurred and who was responsible. It clearly consumes a lot of disk space, as a lot of unnecessary information is stored in almost every audit entry. The second and fourth designs reduce the disk space required to store the audit information, but potentially make it more expensive to access some of the information, like what was changed. You either need to parse the XML or join the two tables together. The third design reduces the consumed disk space in some cases, but it has the potential to increase the required disk space when a lot of columns are modified in one, because it then stores a lot of duplicate metadata.

So there are Audit Table formats compatible with Topicus' needs. Especially the second and fourth designs, storing the changes in XML or using two audit tables, are suitable. Exact statistics on the amount of disk space which they consume, and on their performance impact, is not available. Testing needs to be done if more information about the actual disk space usage, and performance impact, is desired.

9. Conclusions

It is clear that there are alternative methods for auditing, which better match Topicus' needs and which can support the reverting of specific entities to earlier versions.

The investigations into audit solutions and the tests with the prototypes revealed that the Service Broker, together with a dedicated audit server, make it possible to audit actions at a low performance cost while meeting all other needs of Topicus. This auditing solution also works well without the dedicated audit server. Without a dedicated server the Service Broker is still the best solution, because part of the audit process can be delayed until the server is no longer under a heavy load. The audit information can be contained in audit messages, stored in the service broker queues, until the server is under a lighter load. Then the messages can be received and processed. The queue can even be disabled while the server is under a heavy load, it then will store all messages which are send to it, but messages cannot be taken from the queue. Then the queue can be re-enabled once the server is no longer under a heavy load, so the audit messages can be processed then. The Service Broker needs to be combined with another solution as it cannot react to an action, and thus cannot gather the audit information by itself. Aspect Oriented Programming and Triggers were both found to be suitable for this, but AoP has the disadvantage that it is applied on the application and thus cannot create audit information for actions which were executed directly on the database. It does offer a slightly lower overhead than the Triggers, but not a significant enough difference to make up for this disadvantage.

We found that the audit information can be used to revert entities to an earlier version correctly, if a complete and accurate audit trail exists. For every action, we need to store when the action happened, on which entity the action was executed, which columns were changed and what their old values were. If an audit trail with this information in it is available, then entities can be reverted to their previous versions. Information on foreign keys can be retrieved from the database, making it possible to take the relational structure into account. With the information on foreign keys, it also becomes possible to prevent violations of foreign key constraints.

We also found that several audit table formats exist which are compatible with Topicus' needs. By reducing the amount of redundant data stored in the audit table, the amount of disk space which they consume can be reduced. But depending on the used format, this can come at a performance cost. Retrieving information from the tables can become more expensive. Exact data on the performance cost and saved disk space is not available, so we do not yet have an accurate answer to those sub-questions.

After these investigations we propose the following solutions: In order to create a complete and correct audit trail the Service Broker should be used, together with a dedicated auditing server. Triggers should be used to react to inserts, updates and deletes. These triggers can then pass on the audit information to the Service Broker.

For reverting entities we don't give an advice on which strategies to use, instead we propose the creation of a modular and re-usable design, as the desired strategy depends on your desired goal, which is very much project dependent.

For the audit table formats we propose to use one of the two designs, which greatly limit the amount of redundant data stored. The first of these two designs is the one mentioned in 8.2, where all the changes are stored in one column, in XML format, with other columns containing metadata about the action. The second design is the one mentioned in 8.4, where two audit tables are used, one audit table contains the metadata while the other contains one row for each changed column.

9.1 Future Work

Some issues and questions still remain which can use further investigation.

The Service Broker auditing solution has not yet been implemented in one of Topicus' systems. By implementing it in one of the systems which uses their current auditing solution, it is possible to better compare the Service Broker to Topicus' current auditing system. This will give clear information on if the proposed audit solution is an improvement, and how much of an improvement it is.

Auditing table changes can also be investigated. If the name of a column is changed, or a new column whose value may not be bull is added, or a column is deleted, then the existing audit data no longer accurately maps back to that table. It is impossible to correctly revert an entity from this table if the audit data doesn't accurately map back to the table. If such table changes can happen in your database then we advise to investigate how such table changes can be audited, and how the information can be used when reverting entities.

There is no accurate data about the performance cost and disk space consumption of the audit table designs from chapter 8. An investigation can be done into the actual disk space consumption of the designs, and on the performance impact of the designs when reverting entities and when storing audit information into the audit tables.

10. References

[1] Dominic. Implementing a Record Keeper/Rollback System for a MySQL Database (June 28, 2013). Retrieved April 16, 2014: http://dba.stackexchange.com/questions/45439/implementing-a-record-keeper-rollback-system-for-a-mysql-database.

[2] Fagerlund, M. Getting data through reflection; GetValue (April 12, 2010). Retrieved April 17, 2014: http://lotsacode.wordpress.com/2010/04/12/getting-data-through-reflection-getvalue.

[3] Fraiteur, G. Introducing PostSharp 2.0: #2 - Amazing Runtime Performance Enhancements (September 24, 2009). Retrieved April 15, 2014: http://www.postsharp.net/blog/post/introducing-postsharp-2-0-2-amazing-runtime-performance-enhancements.

[4] junan-microsoft. Announcing Service Broker External Activator (November 21, 2008). Retrieved April 15, 2014: http://blogs.msdn.com/b/sql_service_broker/archive/2008/11/21/announcing-service-broker-external-activator.aspx.

[5] Keister, P. sql history table changes (2012). Retrieved May 7, 2014: http://stackoverflow.com/questions/13710103/sql-history-table-design.

[6] Kozelek, P. Audit Trail Tracing Data Changes in Database (2010). Retrieved May 7, 2014: http://www.codeproject.com/Articles/105768/Audit-Trail-Tracing-Data-Changes-in-Database.

[7] Mitchell, Scott. Maintaining a Log of Database Changes - Part 1 (April 18, 2007). Retrieved April 16, 2014: http://www.4guysfromrolla.com/webtech/041807-1.shtml.

[8] Rusanu, R. Fire and Forget: Good for the military, but not for Service Broker conversations (April 6, 2006). Retrieved April 16, 2014: http://rusanu.com/2006/04/06/fire-and-forget-good-for-the-military-but-not-for-service-broker-conversations/.

[9] Rusanu, R. Recyling Conversations (May 3, 2007). Retrieved April 14, 2014: http://rusanu.com/2007/05/03/recycling-conversations/.

[10] Rusanu, R. Resending Messages (December 3, 2007). Retrieved April 16, 2014: http://rusanu.com/2007/12/03/resending-messages/.

[11] Rusanu, R. Writing Service Broker Procedures (October 16, 2006). Retrieved April 16, 2014: http://rusanu.com/2006/10/16/writing-service-broker-procedures/.

[12] Thomassy, M. Service Broker: Performance and Scalability Techniques (March 2009). Retrieved April 16, 2014: http://msdn.microsoft.com/en-us/library/dd576261.aspx.

[13] Wolter, R. An Introduction to SQL Server Service Broker (June 2005). Retrieved April 15, 2014: http://technet.microsoft.com/en-us/library/ms345108%28v=sql.90%29.aspx.

[14] גלנצר, גיא. What is the real performance impact of triggers? (2013). Retrieved April 16, 2014: http://www.madeira.co.il/what-is-the-real-performance-impact-of-triggers-2/.

[15] @@SPID (Transact-SQL). Retrieved April 16, 2014 from Microsoft: http://technet.microsoft.com/en-us/library/ms189535.aspx.

[16] About Change Data Capture (SQL Server). Retrieved April 16, 2014 from Microsoft: http://technet.microsoft.com/en-us/library/cc645937.aspx.

[17] About Change Tracking (SQL Server). Retrieved April 16, 2014 from Microsoft: http://technet.microsoft.com/en-us/library/bb933875.aspx.

[18] CDC : options to capture more data (username, date/time, etc) (August 28, 2007). Retrieved April 16, 2014 from Microsoft: http://connect.microsoft.com/SQLServer/feedback/details/283707/cdc-options-to-capture-more-data-username-date-time-etc.

[19] CREATE ENDPOINT (Transact-SQL). Retrieved April 17, 2014 from Microsoft: http://technet.microsoft.com/en-us/library/ms181591.aspx.

[20] Creating Service Broker Queues. Retrieved April 17, 2014 from Microsoft: http://technet.microsoft.com/en-us/library/ms171612%28v=sql.105%29.aspx.

[21] Design Pattern Automation. Retrieved April 16, 2014 from s.r.o, SharpCrafters: http://www.postsharp.net/.

[22] DML Triggers. Retrieved April 16, 2014 from Microsoft: http://msdn.microsoft.com/en-us/library/ms178110.aspx.

[23] END CONVERSATION (Transact-SQL). Retrieved April 16, 2014 from Microsoft: http://technet.microsoft.com/en-us/library/ms177521.aspx.

[24] Hibernate. Everything data. Retrieved April 16, 2014: http://hibernate.org/.

[25] Introduction SQL Trace. Retrieved April 16, 2014 from Microsoft: http://technet.microsoft.com/en-us/library/ms191006%28v=sql.105%29.aspx.

[26] ISO/IEC 25010:2011 - Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models (April 17, 2014). Retrieved April 16, 2014 from ISO: http://www.iso.org/iso/catalogue_detail.htm?csnumber=35733.

[27] OnMethodBoundaryAspect Class. Retrieved April 10, 2014 from SharpCrafters, s.r.o: http://doc.postsharp.net/##T_PostSharp_Aspects_OnMethodBoundaryAspect.

[28] Service Broker Activation. Retrieved April 16, 2014 from Microsoft: http://technet.microsoft.com/en-us/library/ms171617%28v=sql.105%29.aspx.

[29] Service Broker Routing. Retrieved April 16, 2014 from Microsoft: http://technet.microsoft.com/en-us/library/ms166052%28v=sql.105%29.aspx.

[30] sp_send_dbmail (Transact-SQL). Retrieved April 16, 2014 from Microsoft: http://technet.microsoft.com/en-us/library/ms190307.aspx.

[31] SQL Server Audit (Database Engine). Retrieved April 16, 2014 from Microsoft: http://msdn.microsoft.com/en-us/library/cc280386.aspx.

[32] SQL Log Rescue - Undo for SQL Server. Retrieved April 16, 2014 from Software, Red Gate: http://www.red-gate.com/products/dba/sql-log-rescue/.

[33] SQL Server transaction log explorer, and disaster recovery tool | ApexSQL Log. Retrieved April 16, 2014 from ApexSQL: http://www.apexsql.com/sql_tools_log.aspx.

[34] Topicus. Retrieved April 16, 2014 from Topicus: http://www.topicus.nl/.

[35] What Does Service Broker Do? Retrieved April 17, 2014 from Microsoft: http://technet.microsoft.com/en-us/library/ms166049%28v=sql.105%29.aspx.

[36] Jacobson Ivar, Ng Pan-Wei. *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2004.

[37] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G. An Overview of AspectJ. In *ECOOP '01 Proceedings of the 15th European Conference on Object-Oriented Programming* (Londen 2001), Springer-Verlag, 327-353.

[38] Waraporn, N. Database Auditing Design on Historical Data. *Proceedings of the Second International Symposium on Networking and Network Security* (April 2010), 275-281.