MASTER THESIS

Towards a generic model for audit trails

T. Harleman

Department of Computer Science

August, 2011

UNIVERSITEIT TWENTE.

TOPICUS

Acknowledgements

This thesis marks the end of the master 'Computer Science – Software Engineering' of the University of Twente, which I followed since September 2008. This research is performed at Topicus Finance in Deventer where I have worked for about 8 months.

I would like to thank Topicus Finance for the opportunity to do this inspiring and challenging project. A special thanks to my supervisors from Topicus: Robin Zagers, for the technical assistance, feedback and for providing useful suggestions during my research. Further, for the fun at the office, which made the time fly. Jeroen Logtenberg, for endlessly reviewing my work and providing useful suggestions and positive criticism, which kept me sharp.

Also, I would like to thank my supervisors from the university of Twente: Ivan Kurtev, for guiding me throughout the project and providing me with useful suggestions to keep me in the right direction. Luis Ferreira Pires for reviewing my work in great detail, despite the late involvement in the project.

Last but not least, I want to thank my parents for their support during my study and Mark Oude Veldhuis and Martijn Adolfsen, my fellow students, for the good times we had while completing most of the courses.

-Tim Harleman

August 2011

Table of contents

1 Introduction

Financial applications that support processes involving money, like banking applications, need to be fault proof. To gain the trust of customers, financial companies spend a lot of time, money and resources on validating their software. Systems that concern money or health require a higher degree of fault proof systems then systems that don't.

Within the domain of mortgages, banks and mortgage companies loan money to clients so that they can finance a house. When a person wants to buy a house, generally they have to go to a mortgage company to get a mortgage for the house. Getting a mortgage involves a lot of paperwork and requires a variety of information about the current and historical financial situation of the buyer. For example, the buyer has to supply information about his yearly income, if the buyer has or had other loans, identification documents and so on. The mortgage company on their turn has a lot of different mortgage products. There are life insurance mortgages, linear mortgages, mortgages with a variable interest or mortgages where money is invested. These days there are over a hundred different mortgage products, often in combination with insurances. In order to keep track of all provided mortgages, with all these complex products and insurances, software systems are necessary.

Software that is used in this process has to be reliable for both parties involved. From the client's point of view, all information about their mortgage should be correct, such as interest, satisfied payments, interest rate, mortgage products, insurances and so on as been agreed upon during the negotiations. From the view of the mortgage company, the system should keep correct records about all the provided mortgages. When such system contains faults, the company could lose money because of incorrect interest rates or by paying out insurances that a client does not have. On the other side, the company has to be able to ensure the data is correct and be able to verify this to its clients. This could be the case when a client claims he paid his monthly payment or that a mortgage product is different than what was agreed upon during the negotiations.

Proving the correct information is stored is an important factor in financial systems. A common approach within administrative applications is the use of audit trails. Audit trail is a logging strategy which makes it possible to store and retrieve information about changes made in the process of creating the mortgage invoice. An audit trail can provide a complete history of how the end product is formed by backtracking the historical data of changes. However, strategy focuses more on storing and less on retrieving the historical information.

1.1 Context

Topicus is a software company that has different units located in Deventer, The Netherlands. Topicus Finance is the department where this research has been performed and they are one of the leading companies that focus on mortgage software systems. These systems support the whole process from the application by the customer to the invoice, which allows clients to get the mortgage. Due to legislations, all changes made by users of the system and by the system itself must be logged. To achieve this, an audit trail is used. An audit trail is a very detailed change log. The audit logs can be used for several purposes, such as possible mistakes which are made in the process can be backtracked and to see why values are as they exist in a specific invoice. When analyzing the logs in real-time, these logs can be used for functionality like fraud detection. Currently, when Topicus wants to query the audit trail, the database server crashes or the query gets a time out. Either way, no results are obtained from the audit trail. Because of poor performance, this research has focused on how we can improve the current use of the audit trails within Topicus and how to make the data questionable with acceptable performance, in units of time.

1.2 Problem statement

Some clients and Topicus Finance themselves are SAS 70 [1] certified. This standard prescribes that the client and Topicus Finance must define a that justifies the data in their current database. For this they use an Audit Trail. The audit trail logs every database change, made in the application in a separate database. With this log, the clients and Topicus can justify the current state of for example, a field within the database, by retrieving historical information. Further, with this log they can reconstruct the database state at any point in time.

The problem is that the audit trail was *optimized to store*. Later, clients asked for functionality to be able to retrieve information from the audit logs. Due to this change in requirements, the audit trail implementation now has poor structure and performance to meet the requests of the clients. The audit trail logs a lot of data *without being able to retrieve any information from the logs within reasonable time.* To get information from the audit trail, the data has to be distributed into multiple smaller databases in order to get any results at all. The data set is too large for the database server to handle in its current database schema. Therefore, functionality for the audit trail data cannot be added to their applications as requested by their clients.

1.3 Research Questions

In this section we propose the research questions which we have addressed during this research. This research has aimed at finding a solution, in the form of a design, which makes it possible to question the audit trail data with a better performance than the current audit trail.

What is a generic architecture for efficient questioning of audit trail logs?

1. What is an audit trail?

We have introduced the concept of audit trails in Section 1.1. More information about audit trails can be found in [2].

2. What data warehouse architecture is suited for storing audit trail logs?
3. What is a generic architecture for handling audit trails?
    a. What meta data is required by the architecture in order to understand the data?
    b. How should a (generic) model, to represent audit trail data, look like?

c. What architectural changes to the architecture or model have to be made in order to make it domain specific for any domain?

To efficiently go through a large bulk of data, extra information, called meta data, is commonly used to group and find data faster and more efficiently. In previous research [2], we have seen that data provenance uses the notion of labeling to label data in order to address data by referring to its labels.

4. What is the performance increase (measured in units of time) of the proposed architecture?

In order to verify the performance of the proposed solution for the business case, we have implemented the architecture in a prototype. By running tests, the architecture could be compared with the performance of the current audit trail implementation.

1.4 Approach and Outline

In order to answer the research questions mentioned in Section 1.3, we start with some preliminaries in Chapter 2, which give an introduction to the terms Audit trail, provenance, data warehousing, modeling and modeling concepts.

In order to find a suitable data warehouse architecture and to answer the third research question, we did some research on several types of data warehouse architectures. We compared these architectures based on relevant criteria and chose an architecture. The comparison is described in Chapter 3.

To see if practice matches the theory, a simple test was performed to see how the architecture would perform for this business case. A data conversion is performed to obtain information about the data storage efficiency of the new architecture compared to the current situation. All of that is described in Chapter 4.

To answer the fourth research question, first, a definition for the term *'generic model'* is given. Interviews were held to find out what types of questions are going to be asked about the audit trail data and what output is expected. From that, we abstract from the domain specific parts to design a model that could, ideally, be applied on any type of logging. While abstract from the domain, we kept in mind that the amount of architectural changes for a domain specific design should be kept at a minimum. Last, a solution regarding meta data is required to let the generic model understand what data it handles. All of that is described in Chapter 5.

Chapter 6 describes the audit trail architecture in more detail. We zoom in on every component in the model to show the responsibilities and how it works internally, specifically for our business case.

To be able to say anything about the performance of the model we implemented a small prototype. By running some example questions we compared the results, in units of time, with the current implementation of the audit trail. From these results and observations made during prototyping, we drew some conclusions about the performance of the model. The prototype and test results can be found in Chapter 7.

Finally, the thesis will end with the final conclusions in which we look back at the research questions and provide the obtained answers.

2 Basic Concepts

This chapter discusses the basic concepts that the reader should be familiar with to fully understand the terms and techniques used in this thesis.

Organization of this chapter Section 2.1 explains the term audit trail, Section 2.2 introduces the term provenance, what it is and how it is used. Section 2.3 talks about why there is need for a model for software in general. Section 2.4 focuses on what modeling can add to logged data. Section 2.5 shows some approaches of how the audit trail is used in practice. Last, Section 2.6 gives an introduction to the domain of mortgages and the process of getting a mortgage.

2.1 Audit Trail

We consider the definition of the concept 'Audit Trail' from [3]:

"An audit trail or audit log is a chronological sequence of audit records, each of which contains evidence directly pertaining to and resulting from the execution of a business process or system function."

In the context of this research an Audit Trail is a log that holds all changes made to a database, usually changes made by users of a system. A big problem with this form of logging, as with almost all forms of logging, is that the logs grow very big, very quickly. This is often compensated by selective logging or keeping the logs for shorter periods. Our business case consists of a situation where there is no room for compensations and all data needs to be logged for a long period. With this research we aim at finding a solution to obtain information from these large audit logs.

2.2 Provenance

Since logging, like an audit trail, easily grows out of manageable proportions, we need some data storage structure to keep some performance on the long run and that can support the storage and retrieval of historical data. For that, we look into *data provenance*. Provenance (also referred to as lineage or pedigree) means *origin* or *source*. Some call it *"the history of ownership of a valued object or work of art or literature"*[4]. From a scientific point of view, data sets are useless without knowing the exact provenance and processing pipeline used to produce derived data sets. In relation to our problem, when an invoice for a mortgage is created it is valuable knowing what data is altered during the process of creating the invoice.

We look into two flavors of provenance, namely workflow provenance and data provenance to determine which seems more applicable to our problem.

2.2.1 Workflow provenance

A workflow can be thought of as a sequence of steps which can be either computational steps, human-machine interaction or a combination of these two. Workflow provenance refers to the record of the entire history of the derivation of the final output of the workflow. In our scenario, the mortgage invoice generation process could be considered as the workflow. Applying workflow provenance provides us the functionality to retrieve the full history of the steps taken in order to produce an invoice. That is the final output in our workflow. The amount of information that is stored for the workflow provenance may vary. It may include complete historical records of the steps taken in the workflow to arrive at a particular dataset. Sometimes records are kept about software versions, brand and models of hardware and use of external software within the workflow. When using external processes within the workflow provenance is usually coarse-grained, which means, that only the input, output and external software is recorded. Those external processes are seen as black boxes.

The idea is shown in Figure 1 below. The audit process logs information on the log moments, indicated with vertical arrows. With black boxes in the process, the only log moments are before and after the black box. White boxes can have internal log moments. Workflow provenance logs more snapshots of the data at the log moments rather than the explicit changes.

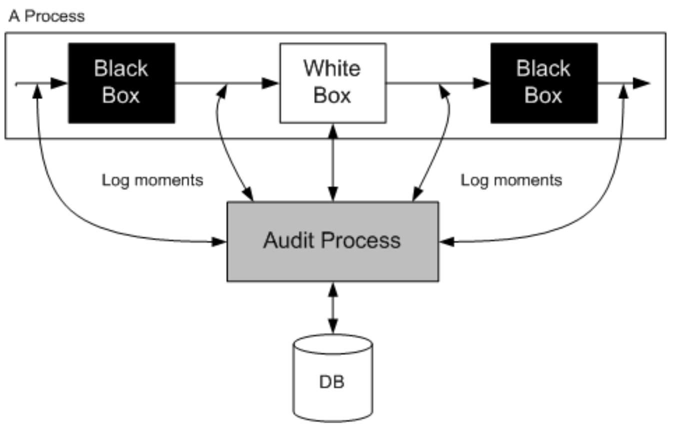


Figure 1: Workflow example

2.2.2 Data provenance

Data provenance gives a more detailed insight about the derivation of a piece of data that is the result of some transformation step. In our scenario, the resulting product could be the mortgage invoice. A particular case of data provenance is very popular within the database community and is extensively researched, which is the when this transformation is performed by database queries. The following explanation comes from [20]. Suppose a transformation on a database *D* is specified by query *Q*, the provenance of a piece of data *i* in the output of applying *Q* on *D* is the answer to the following question: "Which parts of the source database *D* contribute to *i* according to *Q*?" We can further categorize this into where- and why- (or how-) provenance.

- Where-provenance identifies the source elements *where* the data in the target is copied from.
- Why-provenance describes *why* a piece of data is present in the output.

Sometimes why-provenance is referred to as 'how'-provenance and some authorsdefined it as a variant of why-provenance. We ignore this variant in this research.

We explain these categories by means of an example derived from [5].

Suppose ("Jan", 657) is an answer to the following query on the tables shown below.

Select name, telephone
From employee e, department d
Where e.id = d.id *AND* d.name = "Computer Science"

Employee

id	name
1	Jan
2	Henk

Department

id	emp_id	name	phonenumber
11	1	Computer Science	657
12	2	Embedded Systems	739

The where-provenance of the name ‘Jan’ is simply the corresponding record in the name tuple of the Employee table. It only tells you where the data is copied from. This is marked in the table above. The why-provenance includes not only the record in the employee table, but also the Computer Science record corresponding to its employee id. Without the department record, ‘Jan’ would not be included in the result set. These records are marked in the table above.

Data provenance has two general approaches, namely annotated and non-annotated. Non-annotated provenance calculates the provenance using the input, output and query where the annotated provenance can be calculated lateron using extra information that is stored after the execution of a query. The problem with annotated provenance is that it adds a lot of overhead and might be less suitable for very large data sets. The positive side is that it gives more control about what data is more important by annotating it. The downside of non-annotated provenance is that calculations can only be performed during query execution. Afterwards new information cannot be added, unlike the annotated approach where annotations can be added at any time.

2.2.3 Provenance and archiving

Within data provenance, there is a notion of archiving. With the audit log, structural changes of the database can be expected and therefore should also be considered in the solution.

Database schemas change over time for almost every application. For logging systems, structural changes in the database are usually not a big concern. Logging is more about content than structure. Structural changes can be changes in the structure of a field (integer to character) and the removal or addition of fields. It is uncommon practice to reuse the original schema of the database for logging purposes. Archiving is especially crucial for scientific data, where scientific breakthroughs are typically based on information obtained from a particular version of a database. Hence, all changes or all versions of the database must be fully documented for scientific results to remain verifiable. Nonetheless, if there are requirements to reconstruct data structures from loggings, like we have in our case, we might want to keep structural changes to the logs in mind. In our problem domain, mortgage invoices are created with the information that was present at that time. For example, we would like to see the history of a mortgage application, and assume a column of a table was deleted at some point in time. To be able to retrieve information of an application before the column deletion, we have to somehow record this change. This problem can be solved by using provenance archiving. We explain how archiving works by means of an example shown in Figure 2.

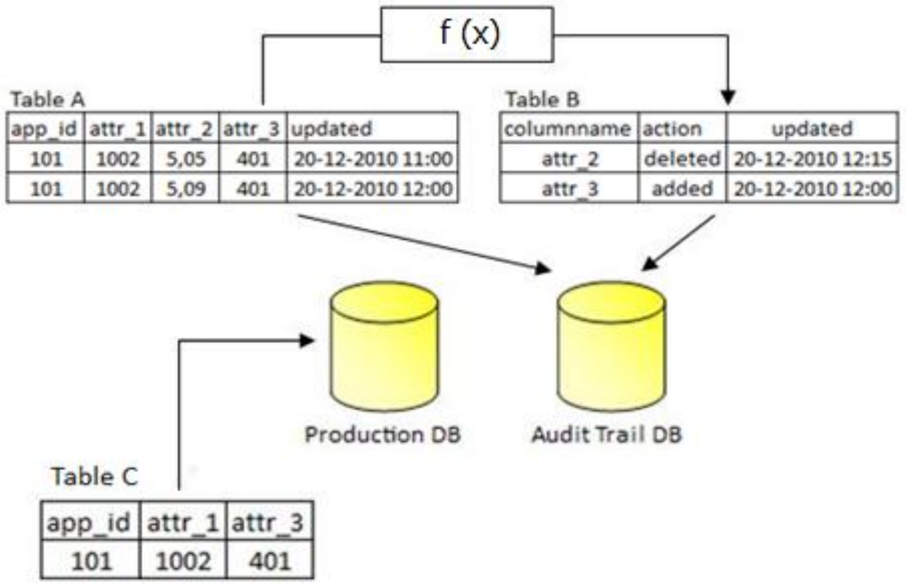


Table A

app_id	attr_1	attr_2	attr_3	updated
101	1002	5,05	401	20-12-2010 11:00
101	1002	5,09	401	20-12-2010 12:00

Table B

columnname	action	updated
attr_2	deleted	20-12-2010 12:15
attr_3	added	20-12-2010 12:00

Table C

app_id	attr_1	attr_3
101	1002	401

Figure 2: Archiving structural database changes

In our scenario we consider two databases: The production database where applications run on and which holds the most recent information (Table C in Figure 2), and the Audit trail database which is responsible for holding the logged information. Table A contains the logged information about Table C. Every record has an additional column with a timestamp indicating when a change occurred. Table B holds the structural changes about Table A, in which we can see which columns were *added*, *deleted* or *changed*. We have a function f(x), which takes a timestamp as value for *'x'* to calculate the state of a record or table during the given point in time x.

Assuming we would like to retrieve the values of the record of the application with *'app_id = 101'* at f(*20-12-2010 11:45*). From Table B we conclude that column *'attr_3'* was not included at that point in time, but *'attr_2'* would be present. From Table A we see that the deletion of *'attr_2'* did not occur yet before 11:45. The result is therefore, (101, 1002, 5.05).

Assuming we would ask the same question again, but now for *'x = 20-12-2010 12:10'*. From table B we conclude that *'attr_3'* was added and thus should the value for this column be included in the result. From Table A we see that some changes have been made before 12:10. The result is therefore (101, 1002, 5.09, 401).

From the archiving point of view this approach works. A detailed registration is held about the structural changes of the application and is easily retrievable. Nevertheless, the storage of Table A is inefficient since it holds a lot of redundant information. This example explains how the archiving of structural database changes can be recorded. How the content is stored is a different point of concern.

2.3 Modelling purposes

Modeling in software engineering is a way to represent complex structures in a simplified manner. Modeling is used to be able to understand the complexity of a problem and to make it easier to talk about it. When making models for data, models can bring structure to facilitate reasoning about the data.

Modeling is often used in the process of designing software. Small projects do not necessarily need a model before building the real system. These types of small projects mostly share the following five characteristics [6];

1. The problem domain is well known.
2. The solution is relatively easy to construct.
3. Very few people need to collaborate when building or using the system.
4. The solution requires minimal ongoing maintenance.
5. The scope of future needs is unlikely to grow substantially.

Larger projects, that do not have the characteristics above, usually require a some sort of model. There is always a possible tradeoff to model a system or build it straight away. The tradeoff is based on the complexity of the project and, the risk of building software without making a modeling. Modeling provides architects and others with the ability to visualize entire systems, assess different options and communicate designs more clearly before taking on the risks —technical, financial or otherwise — of actual construction. Some software systems support important health-related or money-related functions, making them complex to develop, test and maintain. These days, software become more and more important for almost any business process. Therefore, developers need a better understanding of what they are building. Modeling is an effective way increase the understanding. More specifically, by modeling software, developers can:

- Create and communicate about software designs before committing additional resources.
- Trace the design back to the requirements, helping to ensure that they are building the right system.
- Practice iterative development, in which models facilitate quick and frequent changes.

2.4 Modeling and logging

To be able to retrieve information from large datasets like audit trail logs, the structure of the data is an important aspect. Structuring large datasets has a few advantages. Structured logs, or databases in our case, are readable to some extent. Structure provides better understanding of the content, which facilitates the design of software or models to give purpose to the data. In practice, several purposes for audit trails can be found. For example, the logs can be used as logs as intended, thus when something goes wrong and the cause needs to be found. Another purpose which seems to become more popular is analysis of audit trails and other logs. In this report we define two types of analysis:

Passive analysis: A combination of questioning the logs for e.g. tracing problems and using the logs to generate statistics or use it for data mining.

Reactive analysis: Done real-time and can be used to detect errors made by the user and generate a notification..

Reactive analysis is more intertwined with applications while passive analysis can be supported by a separate component. A problem that arises with reactive analysis is that performance becomes an issue depending on what information is analyzed. When someone has permission to change a value can be done fairly quickly, To find out whether a set of values has been changed more than five times, with a large log can take too much time. Passive analysis has no real-time requirements and therefore performance is less of an issue.

2.5 Approaches for audit trail analysis

A lot of approaches, concerning audit trails, focus on the areas of intrusion detection and fraud detection. Audit trail are generally monitored and analyzed to build up a dataset. This dataset is used to compare real-time actions in a particular software system to detect fraud or intrusion by comparing the actions with what is known or expected behavior. In the following sections, some approaches are discussed.

2.5.1 Fraud in administrative ERP systems

In financial and administrative applications, logging analysis is used mainly to detect or prevent any form of fraud. In [7], P. Best refers to five essential steps for detecting fraud in software systems.

1. Understanding the business or operations.
2. Performing a risk analysis to identify the types of frauds that can occur.
3. Deducing the symptoms that the most likely frauds would generate.
4. Using computer software to search for these symptoms.
5. Investigating suspect transactions.

In [7] they use audit trails as a means to detect fraud in ERP (Enterprise Resources Planning) systems. ERP systems are software systems in which administrative information is stored about the company. ERP systems can be centralized so that there is one administrative system for large companies with multiple settlements in different cities or countries.

P. Best,[7], focuses on fraud in such administrative systems. To do so, they define various types of audit trails. Security audit trails log information of user activity to the system. These logs often include successful logins, failed logins, starting of a transaction or action, failed starts of (trans)actions (i.e. prevented because of role permissions) and changes in roles. Usually these audit trails may be retained for periodic review, then archived and/or deleted. Accounting audit trails log specific information concerning financial transactions, like who does payments, when are they performed, who made the payments, who checked financial balances and so on. With this log the financial companies can backtrack every payment that is performed or viewed within the system. [7] defines an audit trail approach to support detection of fraud. The approach consists of two stages:

1) threat monitoring, which involves high-level surveillance of security audit logs to detect possible ‘red flags’. To decide what a possible threat is, they use the audit logs to build up a profile of each user over a certain time period. This profile gives an indication of the frequently performed actions of a user, or patterns in the actions. A knowledge base system may also be developed to generate forecasts of expected user activity. Changes in actual user behavior may then be detected promptly and investigated. The forecast of predicted actions can be improved by creating user profiles in a smaller time frame and compare this to see shifts in the profile. For example, a user can perform different actions in the beginning of the week than at the end of the week.

2) Automated extraction and analysis of audit log data to provide documentation of user actions. This stage creates documentation of users and their activities focused on the fraud sensitive area’s (e.g. financial transactions). Those reports contain facts and can be reviewed to detect fraud that might not have been detected in the first stage.

2.5.2 Rule-based system for universal audit trail analysis

The University of Namur in cooperation with Siemens Nixdorf Software S.A. developed a rule based language for universal audit trail analysis for UNIX. The language is called RUSSEL [8] (Rule-baSed Sequence Evaluation Language) and tailored to the problem of searching for arbitrary patterns of records in sequential files, like audit trails. The built-in mechanism allows records to pass the analysis of the sequential file from left to right. The language provides common control structures, such as conditional, repetitive and compound actions. Primitive actions include assignments, external routine calls and rule triggering. A RUSSEL program consists of a set of rule declarations that are identified by a rule name, a list of formal parameters and local variables and an action part. The action part can consist of user-defined or built-in C-routines. A simple and clearly specified interface with C allows users to extend the RUSSEL language with any desirable feature. This can include simulation of complex data structures, specifications of alarm messages (mail, text message, popup), locking a user account and so on.

When analyzing audit trail logs, the system executes all the active rules on every record. The execution of an active rule may trigger or activate new rules, raise alarms, write report messages or alter (global) variables. Rules can be activated for the current record or the next. Once all rules are executed for a single record, a new record is obtained from the log and all rules return to their initial state. This means that, rules that were triggered to be active become inactive again unless triggered to stay active. The abstract syntax of RUSSEL can be found in [8]. Example rules can be found in [9]. The operational semantics of the RUSSEL language can be summarized as follows:

- Records are analyzed sequentially. The analysis of records consists of executing all active rules. An active rule can trigger other rules, raise alarms, write report messages, alter variables etc.
- Rule triggering is a special mechanism by which a rule is made active either for the current or the next record. In general, a rule is active for the current record because a prefix of a particular sequence of audit records has been detected. The rest of the sequence has to be possibly found in the rest of the log. Parameters in the set of active rules represent knowledge which is obtained from the already analyzed records. This knowledge is used while analyzing the rest of the records.
- When all the rules active for the current record have been executed, the next record is read and the rules triggered for this record in the previous step are executed in turn.
- To initialize the process, a set of so-called initialization rules are made active for the first record.

2.6 Domain of mortgages

The mortgage domain has its own vocabulary, concepts and terms. Therefore we first introduce some terms that are often used and later describe the general process of a mortgage request. The process description will end when the mortgage application is approved. The process of paying off mortgages and insurance activities are not described and irrelevant for this research.

2.6.1 Organization

In order for a mortgage company sell mortgages to clients, there are a few parties involved. The mortgage company also needs ways to be profitable and gain its money. Figure 3 indicates of the involved parties.

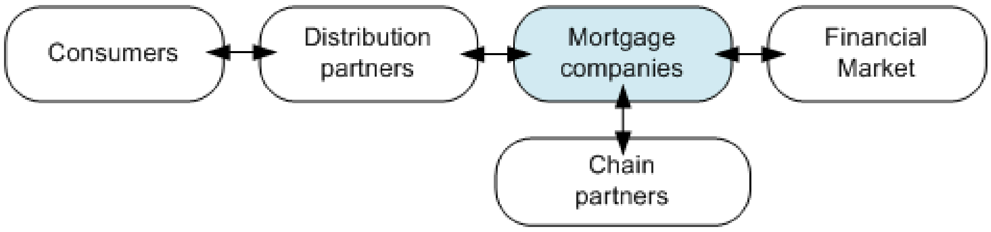


Figure 3: Global organization of a mortgage company (from [10])

Within *mortgage companies*, three offices are usually defined, namely Front-, mid- and back-office. The front-office usually refers to the different sales and distribution channels a mortgage company knows. Some mortgage companies have their own sales department, but outsourcing that activity to different distribution partners seem to happen more often. *Distribution partners* are intermediaries or franchisers. This kind of people sell mortgages for different companies. The advantage is that *consumers* (people looking for a mortgage) have more choice in mortgages from various companies with an ‘independent’ advice when they visit an intermediary of franchiser. Naturally, mortgage companies give bonuses when an intermediary sells their products. Such bonuses make advice less ‘independent’. The mid-office performs the process of making offers for consumers, validating the applications and accepting applications. The back-office comes after the mid-office and performs the process after the mortgage deal is closed. It focuses on consumers paying off their mortgages. *Chain partners* are specialized companies that can take over some of the processes of a mortgage company. Examples are outsourcing of the back-office, but also authorities that supervise the mortgage market, like AFM [11]. Like car manufacturers need resources to build cars, mortgage companies also need resources to be able to sell mortgages. Those resources are called financial means and are gained from the *financial market*. Traditionally, banks use its clients savings as resources but mortgage companies are not always banks, thus they need other ways for getting money.

Nowadays, mortgage companies sell their mortgage wallets (a set of mortgages) to third parties. In that way, mortgage companies can directly use money again and the paying off risks now lay at the third party holding the wallet. Another popular approach is ‘securitization’. Mortgage companies sell their wallet to a so called an SPV (Special Purpose Vehicle), which is a Ltd. Company to be founded. The SPV generally transforms the mortgage wallet into obligations on the stock exchange and sells them. Each SPV is responsible for getting money from its consumers, paying interest and repaying the obligations. When a wallet ends with loss, the obligations with the lowest rating end up with the loss. The higher the ranking of the obligation, the less risk is in case the wallet ends with loss.

2.6.2 Process

We consider the process here from the point of view of the mortgage company. Sometimes they have their own sales department although sales could also be done via an intermediary. An intermediary sells mortgages for various companies and the mortgage company makes deals with these intermediaries to boost the sales of their products. In Figure 4 a schematic overview of the primary process of a mortgage company is shown. These days, a lot of insurances are sold together with a mortgage. Insurances are not included in the primary process. The primary process describes only the phases that are usually adopted in the processes of the mortgage companies. Variations of this process and perhaps totally different processes. For simplicity, we use this general process to characterize the mortgage process.

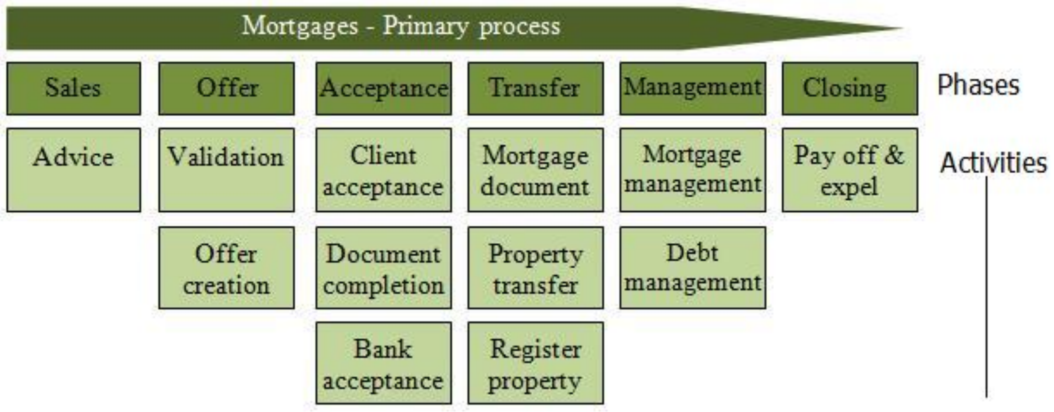


Figure 4: Primary process for mortgages

Each phase is discussed below.

Sales – Front office

Advice: The sales of mortgages is done by either a sales department of the bank or mortgage company or by intermediaries and franchisers. Sometimes you see mortgages are sold via the internet. In this phase, the seller tries to find out the needs of the client and an estimation is made whether the client would be accepted by the mortgage company. Based on the needs, a mortgage product is selected and proposed to the client.

Offer – Mid office

Validation: When the client has interest in the mortgage that was proposed during the sales phase, the client can ask for an offer. The mortgage company will validate the application and will check according to a checklist to see if the company can offer the mortgage the client requests. Mostly some background information is looked up about the client and some global financial information from BKR [12]. If the application passes these checks, the offer can be created. A full acceptance check is done later.

Offer creation: In this phase the offer is created and presented to the client.

Acceptance – Mid office

Client Acceptance: The client accepts the conditions described in the offer, signs the documents and sends them back to the mortgage company. Here he formally agrees on his side of the agreement.

Document completion: The mortgage company starts with gathering all sorts of documents that are required for the application of a mortgage. Sometimes via the intermediary. These documents can consist of copies of identification documents, pay slips and a variety other documents that highly depend on which mortgage is offered.

Bank acceptance: When all the required documents are gathered, the documents are read and checked where necessary. When all documents are checked and accepted, the mortgage company agrees on their side of the agreement for the mortgage offer. The mortgage application is now complete and the client gets access to the money.

Transfer

This phase is about the transfer of the house from seller to buyer.

Mortgage document: The mortgage company sets up a notary instructions document in which the mortgage company describes how the house will be financed. The payment can be fully covered by the mortgage or that a part will be paid by the client. For example, from money that was left over from the sales of another house. This document is also an agreement between the client and mortgage company.

Property transfer: The seller and buyer sign the house-transfer document. The buyer is now officially the new owner of the house.

Register property: Every property has to be registered at the 'Land register' This is a central register in which the property rights and mortgage rights on property are registered.

Management – Back office

Mortgage management: Once the property is officially transferred, the loan of the mortgage is activated. Mortgage management concerns about every action related to paying off the mortgage. An action can be a monthly fee over interest, a payment in between or any amount or changes to the mortgage conditions during the pay off period. This is the standard process.

Debt management: If a client does not pay his mortgage, the client ends up in the debt management. These clients are handled separately from the rest and are monitored more intensively. Once the client catches up with his payment he is transferred back to the standard mortgage management. If the client keeps paying late or not at all, the mortgage company might be forced to sell the property. The profit of sales is for the mortgage company.

Closing

Pay off & expel: Just like other products, mortgages have a lifecycle. The cycle starts with the first contact with the client and ends in this phase. The cycle ends when either the mortgage is paid off or on initiative of the client which has to pay off the mortgage by, usually, getting a new mortgage for another house.

3 Data warehouse architectures

A data warehouse is required to store the audit trail data. Therefore we spent some time selecting a suitable data warehouse architecture. This chapter focuses on the selection of a data warehouse architecture that is suitable for working with audit trails in general and how the selected architecture would be applied on our business case.

Organization of this chapter: Section 3.1 describes the approach that aims at answering the second research question. Section 3.2 looks into several data warehouse architectures from which we have made our choice. We discuss the characteristics of each individual architecture. Section 3.3 covers the comparison of the architectures by looking at their strengths and weaknesses and grading them according to the predefined criteria. Last, in Section 3.4 holds the conclusion about the chosen architecture and the results of the comparison.

3.1 Approach

In order to answer the second research question (*"What data warehouse architecture is suited for storing audit trail logs")*, the following has been done. We selected four data warehouse architectures. From those four, two are specific products, that have been selected since they implement an interesting variant of a standard architecture, or the underlying architecture is specifically designed for its product. Therefore some product were selected, but the focus is on their underlying architecture. The different architectures are selected based on the differences in the underlying architecture and their theoretical performance. The information about performance is obtained from experts and the (product) developers. Four different data warehouse architectures and products were selected, namely *OLAP*, *Normalized Data warehouse, FluidDB* [13] (uses EAV data triples [2]) and *InfoBright* [14] (which uses compression and a special form of indexing). We will introduce each architecture or product and list their strengths and weaknesses.

The selected data warehouse architectures are compared on several criteria that are relevant for our business case. The architectures are compared according to the following criteria:

- *(Analytic) query performance*. Most architectures are designed for either analytic or regular queries. We look at how the architectures perform on both these types of queries in relation to each other, as stated by experts and developers. Grading: 1(bad) – 5(very good)
- *Maximum data size.* Roughly for which the architectures still performs acceptable. Beyond the indicated size, the performance goes down exponentially. Grading: 1(small size) – 5(large size), relative comparison between the architectures and kept in mind an expected audit trail size in the order of 30 to 50 Gb.
- *Meta information*. Information that is added by the architecture itself. Meta information becomes overhead, unless it is helpful information to our solution. Therefore, meta information forced by the architecture should be limited. Grading: 1(a lot) – 5(none)
- *Support to textual data*. To what degree the architectures are designed to handle textual data Grading: 1(bad) – 5(very good)
- *Data Redundancy*. Whether the architectures allow redundant data and to what degree. Redundancy is seen as overhead and should be limited.

Audit logs tend to have a very high degree of redundancy, therefore the grading is doubled for this criteria. Grading: 1(A lot) – 10(None)

- *Scalability*. How the architecture scales, horizontally (addition of servers) and/or vertically (addition of CPU/memory to a server) Grading: 1(bad) – 5(very good)
- *Maintainability*. How much effort, in units of time, is required to maintain the data warehouse. Grading: 1(bad) – 5(very good)
- *Side-effects.* Caused by possible insert/update/delete actions. Audit trails log a lot of information. To keep the data warehouse up-to-date, a continuous stream of insertions are performed for new log records. The insertions should not have negative side effects. Grading: 1(None) – 5(a lot)

By looking at the strengths and weaknesses of the different architectures and by giving grades from 1(bad) to 5, or 10(very good) to the criteria above, we got an overall score for each architecture. The scores have been assigned based on literature, known research, opinions of experts or our own knowledge. Based on the score, the appropriate architecture has been selected.

3.2 Normalized database

For the structure of our data warehouse we looked into the field of sensor data storage. Sensors produce a lot of data which needs to be stored and retrieved. Our problem involves less data than sensor storage has to cope with, therefore we think that if a solution works for the storage of sensor data, it would perform well in our case.

The University of Central Florida [15] did a comparison between normalized data warehouses and denormalized data warehouses to show the advantages between the two. Based on their results, when normalizing a database, the number of records increases (records generally have 2 or 3 columns) and the data volume decreases, meaning less size on disk. Since we handle large data sets, we are more concerned in reducing the space on disk rather than the number of records. When normalizing, the costs for administration raises and the labeling takes a little extra storage. Depending on how the data warehouse is set up, this could be an issue. With logs, the redundancy rate is extremely high. By removing redundancy using normalization, the extra costs of the extra administration probably does not add up to the decrease in data size.

Since we do not have to handle the data real-time, the time it takes to populate the warehouse is less of an issue. The last ‘weakness’ in the comparison from [15] is that transformations on normalized databases are harder. This is true for the general case, since the normalized databases have more tables. Also, a data warehouse should not be used for other purposes than questioning the data in the way the data warehouse was set up. Apart from that, transforming a data warehouse is usually done only once and not on a regular bases. Considerations regarding possible transformations should be known and kept in mind when designing the data warehouse.

In the field of databases, normalizing a database is the first step towards maximizing performance. Five normal forms have been defined. The higher the form, the more normalized the database is. The different normal forms are explained in [2].

Comparison

Table 1 shows the evaluation of the normalized data warehouse architecture regarding the criteria, which are defined in Section 3.1

Table 1: Evaluation of the normalized database architecture.

Criteria	**Evaluation (score)**	**Motivation**
Query performance	Good (4)	Normalization is applied to increase the query performance in general
Analytic query performance	Good (4)	Due to the standard functionality (from SQL) to calculate multi cubes
Scalability	Good (4)	Tables could be divided over multiple servers and more memory increase cache size, thus performance
Data redundancy	None (10)	The goal of normalization is to minimize or prevent data redundancy
Max. database size	10 – 50 GB (3)	Depending on the size of the individual tables, hardware and difficulty of queries
Meta information	None (5)	By itself, a normalized database does not generate meta information about its content.
Textual data	Good (4)	Textual searches are always slower then numeric searches, but due to low/no redundancy, the amount of text to be searched is limited.
Side effects	None (5)	By normalizing a data warehouse, no relevant side effects appear.
Maintainability	Good (4)	It has a simple database schema and there is good tool support for maintenance.

3.3 OLAP

OLAP stands for **O**n**l**ine **A**nalytic **P**rocessing and is an approach to handle multi-dimensional analytic queries. OLAP systems generally are used for business intelligence reporting tools and data mining where data cubes are used to retrieve the desired information. OLAP is a sort of layer that runs on top of a normal database. There are three types of OLAP architectures, namely, MOLAP, ROLAP and HOLAP. The biggest weakness is that OLAP products are not suitable for handling data with multi-hierarchical and unbalanced structures. The strength lies in retrieving analytical and statistical information from data sets.

Several comparisons have been made between the different types of OLAP. We use a comparison performed at the Uppsala University [16]. From their results and observations we can conclude that for our case, MOLAP is not suited, since the data sets we want to handle exceed the limit of MOLAP, which is said to be around 1-2 Gb. Therefore, it is better to go into the direction of ROLAP or HOLAP. Considering the goal of our project, we feel ROLAP would be better than HOLAP. HOLAP aim more at analytic questions about the data. We aim at a generic solution, therefore we do not focus specifically on dealing with analytical questions but all kinds of queries. ROLAP has more efficient disk space usage because it can be optimized by normalization. HOLAP uses more disk space because of the multidimensional cubes, that are precomputed for analytic questions that contain a copy of the actual data. ROLAP can create and (optionally) store these cubes, but does not pre-compute them. ROLAP can be used using normal SQL tools, since it is a layer on top of a normal database, which makes it more generic for average use than having to study OLAP tools in order to work with it. Last, loggings generally contain quite some textual data. ROLAP is said to handle this kind of data pretty well. For all these reasons we select ROLAP as the OLAP representative for our project based on the conducted comparison in[16].

Evaluation

Table 2 shows the evaluation of the ROLAP architecture regarding the criteria defined in Section 3.1.

Table 2: Evaluation of the ROLAP architecture.

Criteria	**Evaluation (score)**	**Motivation**
Query performance	Good (4)	The architecture is designed to obtain information from large data sets
Analytic query performance	Very Good (5)	The multi dimensional cubes improve the performance of the analytic queries
Scalability	Good (4)	Tables could be divided over multiple servers and more memory increase cache size, thus performance
Data redundancy	Average (6)	When the multi dimensional cubes are stored they hold copies of the original data.
Max. database size	10 – 50 GB (3)	Depending on the size of the individual tables, hardware and difficulty of queries
Meta information	None/Little (4)	When multi cubes are calculated and stored, the cubes hold statistical/analytical information about the content.
Textual data	Good (4)	Textual searches are always slower than numeric searches, but due to low/no redundancy, the amount of text to be searched is limited.
Side effects	Yes (3)	Stored computed multi cubes might have to be recomputed from time to time.
Maintainability	Good (4)	It has a simple database schema and proper tool support for maintenance is available.

3.4 FluidDB

FluidDB, developed by FluidInfo [13], aims at becoming the ‘wikipedia of databases’. FluidDB is a form of cloud computing which is based on storing social data. Cloud computing is a service that provides data, software and hardware via the Internet and is location-independent. The cloud provider stores the data and makes sure it is accessible via the internet, usually via a browser interface. Social data is publicly accessible and can be questioned by anyone. For example, an application based on FluidDB, called tickery [17] (twitter query), collects twitter messages and stores it as data triples. FluidDB is the database to which everyone could add information to it or about data that is already in it. It is a publicly writable database so that all related information can be stored about some particular information. For example, by supplying the information about facebook friends, one could query who Person A follows on Twitter, as well as his Facebook friend.

FluidDB uses an architecture based on Entity-Attribute-Value (EAV) triples, or data triples. The architecture is implemented so that the data is stored in a relational database, and the actual content is stored as compressed XML. Via those triples, the location of the actual content in the compressed XML can be obtained. The EAV architecture is used as a lookup system for the actual content. In FluidDB, there are four key concepts, namely:

- Objects: Represents the actual content
- Tags: Labels attached to objects which define the attributes of objects.
- Namespaces: Organizes or groups tags
- Permissions: Handles access control

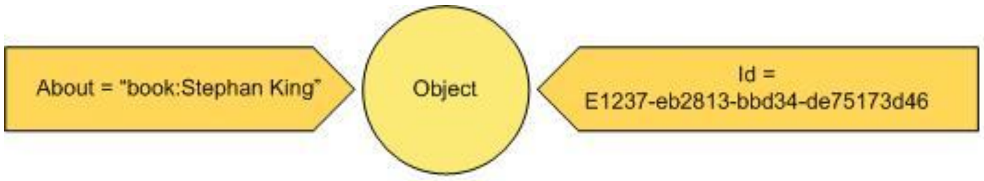


Figure 5: FluidDB data format.

The object represents the actual content, the 'about' tag is optional but should be unique and an id can be used to retrieve the actual content. Per object, additional tags can be attached. A tag consists of a namespace followed by the name of the tag, and a value. The data triples consist of an object, a tag and a value. An example of an object is represented in Figure 6:

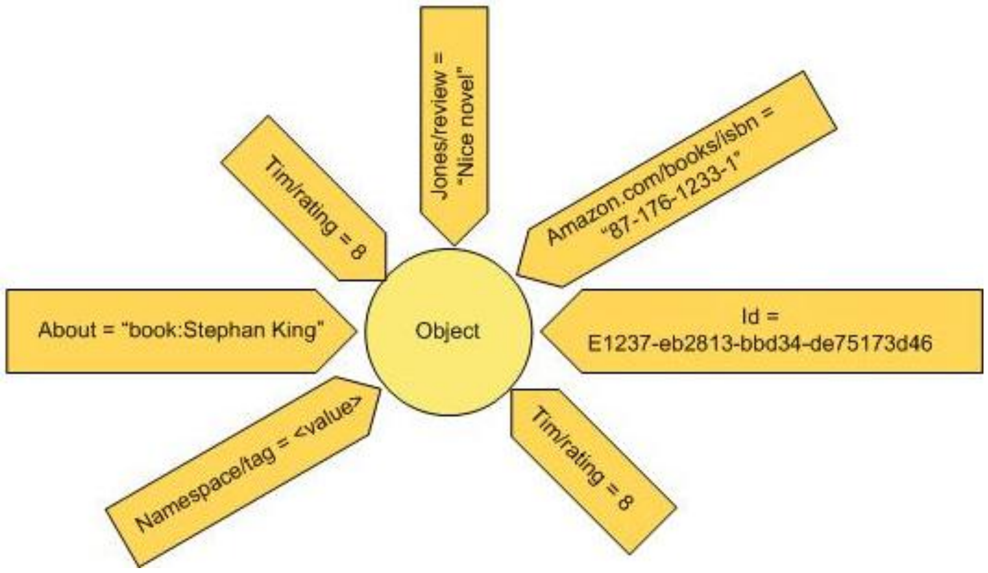


Figure 6: Visualisation of data triples for an object in FluidDB

FluidDB also supports the handling of permissions. Traditionally you would have permissions on an object. In FluidDB, the permissions are set on the namespaces, tags and tag-values. A permission consists of a namespace, tag or tag-value where the permission should be applied, a scope (e.g. see, create or read) and a list of exceptions. Exceptions mean that the permission restriction does not apply for the specified users. This approach keeps maximum flexibility for setting permissions but results in a (possibly) huge permission table. The permission information is stored as data triples as well.

In our own research [2], we concluded that data triples is not the best architecture for our problem. FluidDB made some improvements to the concept of data triples and showed it can be effective for large sets of data, like tickery. But, the use of cloud computing, makes FluidDB an unsuitable choice. Its architecture is tailored towards social data which is publicly accessible, which is something we do not want looking at the confidentiality of the audit data. Cloud computing is getting more secure for private use, but the client has to have faith in that security. The permission system on the other hand is simple and effective but might get out of hand when permissions need to be set for a lot of users and a huge tag set, especially when the tags can be defined by any user per object. Tag management, therefore, should be managed closely so that it does not get out of hand.

Evaluation

Table 3 shows the evaluation of the FluidDB data triple architecture regarding the criteria defined in Section 3.1.

Table 3: Evaluation of the FluidDB data triple architecture.

Criteria	**Evaluation (score)**	**Motivation**
Query performance	Good (4)	The architecture is designed to obtain small pieces of information from large datasets.
Analytic query performance	Poor (2)	Analyzing the data triples with possible different labels per object takes a lot of effort.
Scalability	Average (3)	Good scalability, horizontally due to cloud computing. Average scalability vertically.
Data redundancy	Little (8)	Not intended, but the architecture does not restrict it.
Max. database size	450 – 500 GB (4)	Data triple architectures are measured in number of triples rather than size. They can handle a few billion triples. A rough indication is 250 kb per triple * 2 billion = 450 – 500 GB
Meta information	A lot (1)	Tags attached to objects are considered meta information
Textual data	Good (4)	Has no limitations that restrict the performance on textual data lookups
Side effects	None (5)	The usage of data triples has no unintended side effects
Maintainability	Poor (2)	Due to the billion of triples, maintainability suffers. Tool support is limited.

3.5 InfoBright

Infobright [14] is a company that focuses on analytic data warehouse products. They have a commercial and open source variant of their product called Infobright, named after the company. Infobright gives support especially for very large datasets (up to 50 TB). It combines smart compressions with good performance and low installation and maintenance costs.

Infobright is easily deployed, is simply installed and configured and is self-contained, i.e. has no external dependencies. Infobright is easy to manage because no indexes, data partitioning, data partitioning or tuning is needed. It has a very small hardware footprint, which makes it possible to support databases up to 50

TeraByte on a single industrial standard database server. About performance and scalability they claim fast load speed, which remains constant as the size of the database grows, fast query speed on large volumes of data and they offer high data compression with a ratios between 10:1 to over 40:1. This results in less storage and less I/O requirements. Another important factor is that Infobright is column oriented and not row oriented as in traditional databases.

3.5.1 Layers

The architecture behind Infobright consists of 3 layers: Data Packs, Knowledge Grid, Optimizer.

Data Packs. The actual content is stored using efficient compression algorithms. Data is stored in Data Packs, so that the data of columns is divided in Data Packs of 64k elements. Each Data Pack is compressed individually and the compression method can vary according to the data type and repetitiveness of the data within a Data Pack. By doing so, the compression can be optimized by selecting the best fit per Data Pack.

Knowledge grid. The Knowledge grid consists of two parts, namely the Data Pack Nodes (DPN) and the Knowledge Nodes (KN) on top of the DPN. The DPN contains aggregated information about a Data Pack, such as MIN, MAX COUNT (# of rows, # of NULLS) and SUM information. For each Data Pack, there is a Data Pack Node. The Knowledge Nodes keep information about data packs, columns or table combinations. Unlike the indexes required for traditional databases, DPNs and KNs are not manually created, and require no ongoing care and maintenance. Instead, they are created and managed automatically by the system. In essence, the Knowledge Grid provides a high level view of the entire content of the database with a minimal overhead of approximately 1% of the original data.

Optimizer. The optimizer is the most intelligent part in the architecture. It uses the Knowledge Grid to determine the minimum set of Data Packs needed to be decompressed in order to satisfy a given query in the fastest possible time by identifying the relevant Data Packs. By looking into the summarized information in the Knowledge Grid, the optimizer groups the Data Packs in three categories:

- Relevant Packs: Data pack elements hold relevant values based on the DPN and KN statistics.
- Irrelevant packs: Data pack elements hold no relevant values based on the DPN and KN statistics
- Suspect Packs: Data pack elements hold some relevant elements within a certain range, but the Data Pack needs to be decompressed in order to determine the detailed values specific to the query.

The Relevant and Suspect packs are used in answering queries. In some cases, for example if we're asking for aggregates, only the Suspect packs need to be decompressed because the Relevant packs will have the aggregate value(s) pre-determined (in the DNs). However, if the query is asking for record details, then all Suspect and all Relevant packs have to be decompressed.

3.5.2 Data Manipulation Language

DML. Data Manipulation Language is a set of statements used to store, retrieve, modify, and erase data from a database. Infobright is a data warehouse that makes extensive use of compression and grouping ordered data. A big question that arises is how inserted, updated or deleted data is handled. Since all data is compressed, it is not practical to recompress all data after each DML statement.

DML functionality is not available in the Community Edition (ICE), only in the (commercial) Enterprise Edition (IEE). ICE makes use of a bulk import for the data.

Insert. On insertion, the data warehouse buffers multiple rows of incoming data and appends them to the final (partial) data pack. The pack is recompressed only after it is full, or the INSERT operation is complete. A lot of insertions result in two data sets, the original ordered data packs and the newly inserted data packs that contains random information. That results in an increase in the number of data packs that need to be decompressed per query. Therefore, a total recompression of the data is necessary in the case of a lot of insertions.

Delete. On deletion of a row, the data warehouse marks rows as deleted using a separate “delete mask.” This means that the data is not actually deleted, but will be ignored by queries. It impacts performance not that much as insertion, but when a lot of data gets deleted, the data packs get a lot of overhead from the deleted rows. The rows are deleted from disk when the data warehouse is recompressed.

Update. The update functionality is implemented as a deletion followed by an insertion of the new row with the updated information.

3.5.3 Example of query handling

By means of an example we explain how Infobright handles a query. Assume we have the following query:

```
SELECT COUNT(*)
FROM user
WHERE zipcode = ‘2468FG’
AND registrationdate > ‘1-1-2011’
AND gender = ‘M’
```

To evaluated the query, the optimizer uses the constraints of the query to identify the relevant data packs. The optimizer questions the knowledge grid to find out, which data packs are relevant or suspects, using the Data Pack Nodes. The Knowledge Nodes indicate which DPNs are relevant for the query (the user table). Then, the DPNs that belong to the zipcode column from the user table are found. Using the MIN and MAX values the DPNs hold about their data packs, the data packs can be identified that hold the value ‘2468FG’ for zipcode. If the data pack would purely hold the requested zipcode (and thus the MIN and MAX values are the same) the data pack is marked relevant. Next, the other restrictions from the query are evaluated in the same way, except that the only subset of rows that were identified by the previous restriction are evaluated. The selection of data packs is shown in Figure 7. Once all restrictions are evaluated and a subset of data packs is returned, those data packs are decompressed and the count can be performed.

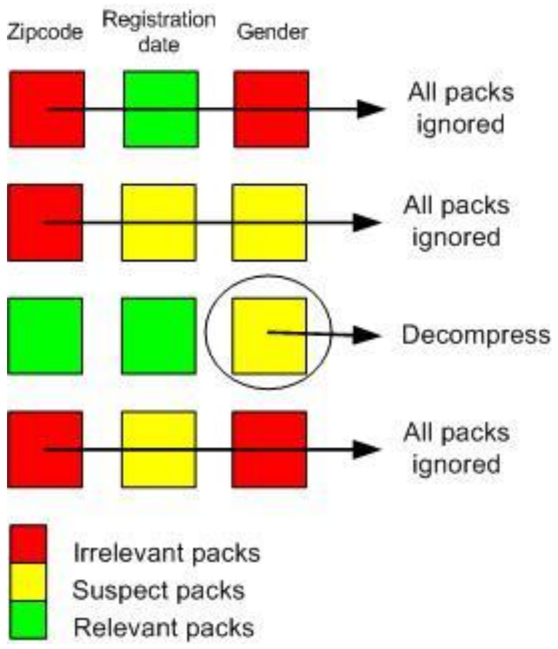


Figure 7: Selection of Data Packs in InfoBright (from [18])

Evaluation

Table 4 shows the evaluation of the Infobright data triple architecture regarding the criteria, which can be found in Section 3.1

Table 4: Evaluation of the InfoBright architecture.

Criteria	**Evaluation (score)**	**Motivation**
Query performance	Very Good (5)	Due to the power of the knowledge grid
Analytic query performance	Good (4)	Due to the power of the knowledge grid
Scalability	Very Good (5)	Scales well horizontally and vertically
Data redundancy	Little (6)	Does no effort to prevent redundant data but due to good compression redundant data does not take much disk space
Max. database size	=> 50TB (5)	Tailored for large datasets of several terabytes
Meta information	Average (3)	The DPN's and KN's in the knowledge grid hold meta information about the content
Textual data	Good (4)	Has no limitations that restrict the performance on textual data lookups
Side effects	Yes (2)	Due to compression of the content, the whole database has to be recompressed periodically to keep up the performance
Maintainability	Very good (5)	Choices for optimal compression algorithms and indexes are done by the system. Close to self-maintainable

3.6 Architecture Comparison

In this section we compare the different data warehouse architectures. All results are summarized in Table 5. The scores are determined based on literature and knowledge of experts.

From the comparison in Table 5 we see that the normalized database architecture seems to be the best fit for our business case. Its strongest point for our case is the removal of redundant data. Logging in general consist of a lot of repetitive information. Therefore we double the score for the '*data redundancy*' criteria. By removing the redundancy the disk size is decreased drastically and implicitly the performance should increase. Infobright seems a good alternative, but its architecture and compression starts paying off at extremely large databases. From the numbers we got about the audit trail growth, the audit trail should run a few years before a product like InfoBright should be considered.

Because of our generic approach we cannot argue whether our solution would be used for merely analytical questions. Therefore picking an OLAP solution seems illogical. FluidDB uses cloud computing and cannot be installed locally inside a company but is only accessible as a service via the internet. Since we deal with confidential data we don't want to add a third party and certainly not stream the data over the internet, although cloud computing becomes more secure. Nevertheless, we have rejected FluidDB as a possible solution.

Criteria	**Normalized Database**		**(R)OLAP**		**FluidDB**		**InfoBright**	
Query performance	Good	4	Good	4	Good	4	Very Good	5
Analytic Query performance	Good, due to the standard functionality to calculate multi cubes	4	Very Good, due to its calculation of the multi cubes	5	Poor, due to its storage as data triples which makes these type of queries expensive	2	Good, due to the power of the knowledge grid	4
Scalability	Good	4	Good	4	Average (horizontally good)	3	Very Good, horizontally and vertically	5
Data redundancy (double score due to importance)	None	10	The computed multi cubes that are stored in the database	6	Not intended, but can arise easily when its data comes from different sources	8	Does no effort to prevent redundant data but due to good compression redundant data does not take much disk space	6
Stable performance up to	10 – 50 GB	3	10 – 50 GB	3	450 – 500 GB	4	> 50 TB	5
Meta information	None	5	None / Little	4	A lot, tags attached to objects are to be considered as meta information	1	Average, the DPNs and KNs in the knowledge grid hold meta information about the content	3
Handling textual data	Good	4	Good	4	Good	4	Good	4
(Side) Effects due to inserts/updates/ deletion	No	5	Yes, Pre-computed cubes have to be recomputed	3	No, it is designed to do this without side effects but it might be necessary to change permission settings.	5	Yes. Due to compression of the content, the whole database has to be recompressed periodically to keep up performance.	2
Maintainability	Good, simple structure and a lot of supporting tools available	4	Good, simple structure and a lot of supporting tools available	4	Hard, due to the public accessibility and simple triple store structure	2	Very good. Choices for optimal compression algorithms and indexes are done by the system. Close to self-maintainable.	5
Total	43 points		37 points		33 points		39 points	

Table 5: Data warehouse architecture comparison

3.7 Conclusion

From Table 5, we can conclude that the normalized database architecture is the most suitable architecture for our problem. It has good query performance, no overhead by generated metadata, no relevant side effects, it is easy to maintain and most importantly, it removes redundancy. Logging generates a lot of data, but with a extremely high redundancy rate. By removing redundancy, the size of the log decreases. When dealing with logs up to 100 GB, every decrease in size, without losing information should contribute to faster retrieval times when questioning the data. When the data warehouse grows, with the normalized architecture, the amount of new values that arise should decrease over time. The growth of the data warehouse is expected to decrease exponentially.

The comparison in Table 5 also shows that InfoBright (commercial version) would be the best alternative. Nevertheless it would be an alternative with some consequences. It is a commercial product, so there are time and effort costs to install and learn the product before it can be used. Another reason not to choose InfoBright, at this point, is that it has been specially designed for very large data sets. The data is stored and compressed in sets of 64k records. When we would remove the redundant values from the test data, there won’t be many fields that have more than 64.000 unique values. If applied to the data we have, almost the whole data warehouse needs to be decompressed for every query, which takes time. Therefore we conclude that InfoBright is definitely not suited for the data set we have. We believe that InfoBright becomes a suitable alternative when the log size gets in the order of several TeraBytes.

The comparison is performed based on reports and statements from experts and other sources, since we are not expects in the field ourselves. The reliability of the comparison might therefore be questioned. Based on the reliability of the references, we believe the comparison is well-founded. On forehand, we expected the normalized data warehouse architecture to be the best architecture, especially for this business case. However, over time, when the content in the data warehouse increases, there must be a point where other architectures become a better choice, like InfoBright, as explained before. Also, there might be better architectures available, which are not included in the comparison. However, for this business case, we believe the normalized database architecture is the best one of the selected four.

4 Data warehouse design

Based on the comparison from the previous chapter, the normalized data warehouse architecture was selected. Normalizing a data warehouse removes redundancy and increases lookup speed, since the search tables become (relatively) small because the resulting tables only contain unique values. In this chapter we describe the design of the data warehouse schema, guidelines for transforming audit logs to the data warehouse and a test conversion we performed to check the size of the log files. The assumption is that, when the log size drastically decreases, the performance is positively influenced. Apart from that, the test conversion should helps us predict the growth of the data warehouse over time.

Organization of this chapter Section 4.1 describes the approach that we used while designing the data warehouse. Section 4.2 talks about a short analysis that we have done on the test data sets that Topicus provided. Section 4.3 covers the design of the data warehouse and the process of converting any audit log to a data warehouse schema, using guidelines. Section 4.4 discusses the results that we obtained while executing a test conversion in order to see the decrease in size between the log and the data warehouse. Section 4.5 holds a discussion on the test results and observations made during the test. Last, Section 4.6 holds the conclusion.

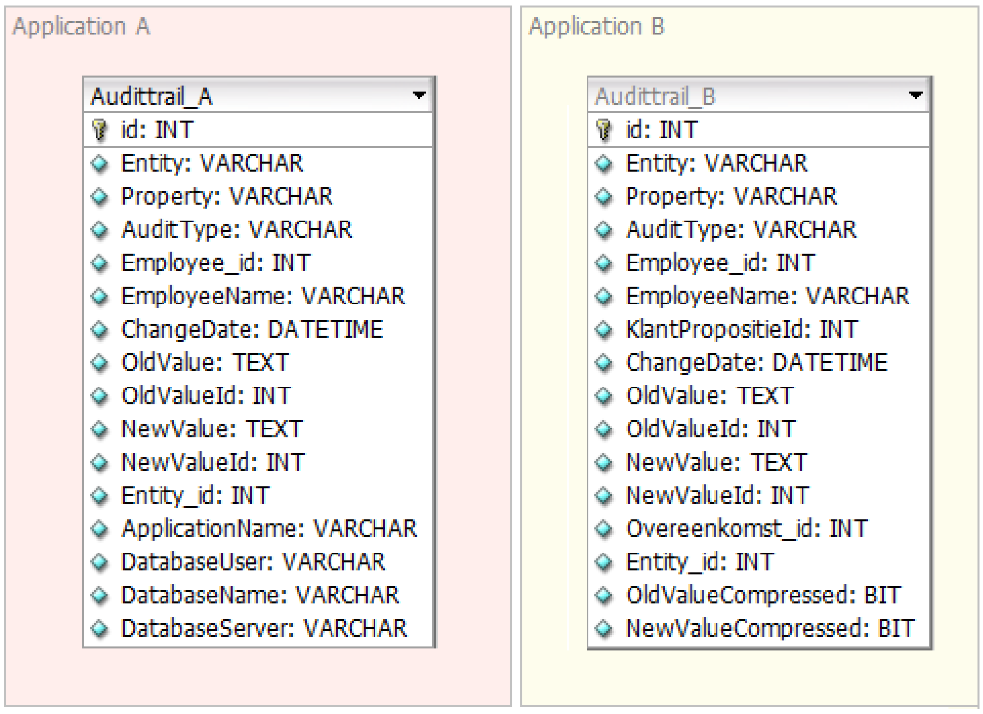
4.1 Approach

In this chapter we discuss the design of the data warehouse schema for the audit trail test set. From chapter 3, we concluded that the normalized database architecture is the best choice for our problem. The design of the data warehouse schema is based on the test data. Since we design a data warehouse that can contain any audit log, guidelines for the data warehouse schema must be defined. These guidelines ensure the data warehouse is always buil of certain constructions and that any audit log can be represented by a schema. Initially, the data warehouse schema is according to the normalization forms [2]. Based on these normalization forms, optimization guidelines are defined to improve the system performance and reduce the data size.

Once the schema has been designed, the data warehouse should be populated with the test data. In order to do that, the data has to be converted. By performing a test conversion, we get an indication of the decrease in size with respect to the original log size. When the size would decrease drastically, it will help boost the performance of querying the data, since there is less data to go through. With logs up to hundreds of Gigabytes, every reduction of data without losing information, is welcome. Also, by observing the results of the test conversion there will be a better understanding of by how much the data warehouse would grow over time.

4.2 Audit trail log analysis

In previous research [2], we looked into what kind of data the audit trail stores based on the design documentation for the audit trail. Based on that, we looked into the actual structure of the audit trail database of two applications. The schema's are shown in Figure 8:


Application A

Audittrail_A
- id: INT
- Entity: VARCHAR
- Property: VARCHAR
- AuditType: VARCHAR
- Employee_id: INT
- EmployeeName: VARCHAR
- ChangeDate: DATETIME
- OldValue: TEXT
- OldValueId: INT
- NewValue: TEXT
- NewValueId: INT
- Entity_id: INT
- ApplicationName: VARCHAR
- DatabaseUser: VARCHAR
- DatabaseName: VARCHAR
- DatabaseServer: VARCHAR

Application B

Audittrail_B
- id: INT
- Entity: VARCHAR
- Property: VARCHAR
- AuditType: VARCHAR
- Employee_id: INT
- EmployeeName: VARCHAR
- KlantPropositieId: INT
- ChangeDate: DATETIME
- OldValue: TEXT
- OldValueId: INT
- NewValue: TEXT
- NewValueId: INT
- Overeenkomst_id: INT
- Entity_id: INT
- OldValueCompressed: BIT
- NewValueCompressed: BIT

Figure 8: Fields of the audit trail from two applications

Figure 7 shows that there is quite a lot of overlap in what the different applications log. The basic information that is logged conforms with the specification as described in [2]. Apart from the basic information, the databases are extended with some application specific information. We think that the Aubittrail_B gives us more useful information then Audittrail_A. The Aubittrail_B trail logs information about 'klant propositie' / customer propositions and 'overeenkomst' / invoices. Based on that, we decided to take a data set of the audit trail from Application B as the input for our design. Due to legislations concerning privacy, we are not allowed to look into the actual database content and therefore we didn't analyze the audit trail any further than the database schema. For this research, a representative test set has been used. The test set contains two types of fields: obligatory fields, meaning every log record holds a value for that particular field, and optional fields, meaning that these fields can be empty or have NULL values.

4.3 Data warehouse Conversion

Based on the analysis of the available audit trail data test set, a data warehouse schema has to be defined. We have little knowledge and insight in the actual content of the data and the goal is to design a generic data warehouse which can handle any audit trail, the concept of labeling is introduced. The idea is based on annotated data provenance. Data provenance uses a form of labeling which adds meta information to data in order to group and relate data. By means of those labels, information gets meaning. Data then can be referred to via labels rather than actual content. As a result, the data warehouse has no knowledge of the data it holds, which means any log/data could be inserted into the data warehouse as long as the data warehouse schema for the audit log is conform certain guidelines.

Section 4.3.1, describes the process of obtaining a data warehouse schema for any audit log. By following the rules and guidelines, the data warehouse schema always has the same characteristics. Based on these characteristics, the architecture, that will be designed on top of the data warehouse, can use the generic structure of the database.

4.3.1 Conversion rules and exceptions

To get a generic data warehouse, we are looking for rules to convert a log that is stored as a flat table in a database, to a normalized data warehouse.

Initially, any log could be converted according to the known normalization algorithms[2]. By definition, the conversion produces a schema in the pattern as shown in Figure 9. The schema is an approved schema according to the normalization rules, but it is not an optimal solution. Every column of the log is placed in a separate table, and attached through a link table to the central LogRecord table. By introducing some exceptions, the schema produces a more optimal solution that still conforms to the normalization rules. The reason to add these exceptions is to decrease the database size and reduce the number of tables. The more tables, the slower the data warehouse becomes, since more joins are required.

When looking only at the column names, we see that some values are related and therefore should be stored together instead of in separate tables. We observed four main situations in which the data warehouse schema can be optimized without introducing new patterns in the schema.

- Exception 1: When the audit log contains a username and a field that already uniquely identifies a user (e.g., user_id), those two columns could be combined instead of storing separately in two columns in the data warehouse. So the table would become *{user_id, username}* instead of the default two columns *{user_id_id, user_id }* and *{username_id, username}* with two link tables.
- Exception 2: When there are columns in the audit log which are related to each other, such as, a 'firstname' and a 'lastname' column. Neither of the two uniquely identifies the other, but both are attributes of a person. Therefore these fields should end up in the same table with a unique identifier: *{person_id, firstname, lastname}*.
- Exception 3: Use enumerations for fields that have a fixed number of values, like a 'gender' field.
- Exception 4: Add all the columns that occur in every log entry (and can therefore be considered obligatory) to the central table (LogRecord as shown in Figure 9). The number of link tables are reduced and we are sure that, by doing so, that we do not introduce empty fields in the center table, which is against the database normalization rules.

Based on the exceptions, the schema can be optimized. When converting an audit log to the standard structure of the schema as shown in Figure 9, the data warehouse schema can be optimized by applying the exceptions to obtain a more optimal data warehouse schema.

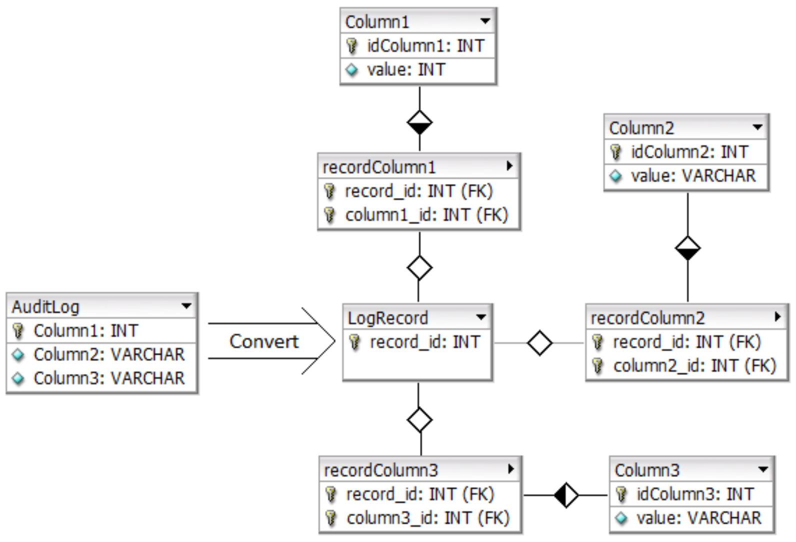


Figure 9: Initial conversion from log to data warehouse

4.3.2 Data warehouse schema

The design of the data warehouse schema is based on the two audit logs as shown in Section 4.2. We have chosen to take the log from application B as our business case, but a similar schema for another application can be obtained using the same conversion rules.

Our proposed data warehouse schema for the audit trail is shown in Figure 10. We call this the ONAT (Optimized Normalized Audit Trail schema) schema. This schema has been optimized using the 4 exceptions as described in Section 4.3.1. An exception has been made on exception 1. The fields Entity and Entity_id (shown in Figure 10) are not combined into one table. Although the Entity_id uniquely identifies the Entity, storing those fields in one table was shown, after analysis of the data, to be inefficient. The analysis showed that there are very few entities in comparison to the amount of ids. The same entity can occur with multiple ids. This results in a high redundancy ratio in the text field ‘EntityName’. These ratios ended up around roughly 5000 : 1, meaning every EntityName would appear around 5000 times in the table. Since the goal of normalizing a database is to reduce redundancy to a minimum and due to the fact that relational databases perform faster with numbers then text, we decided not to apply exception 1 on this particular case.

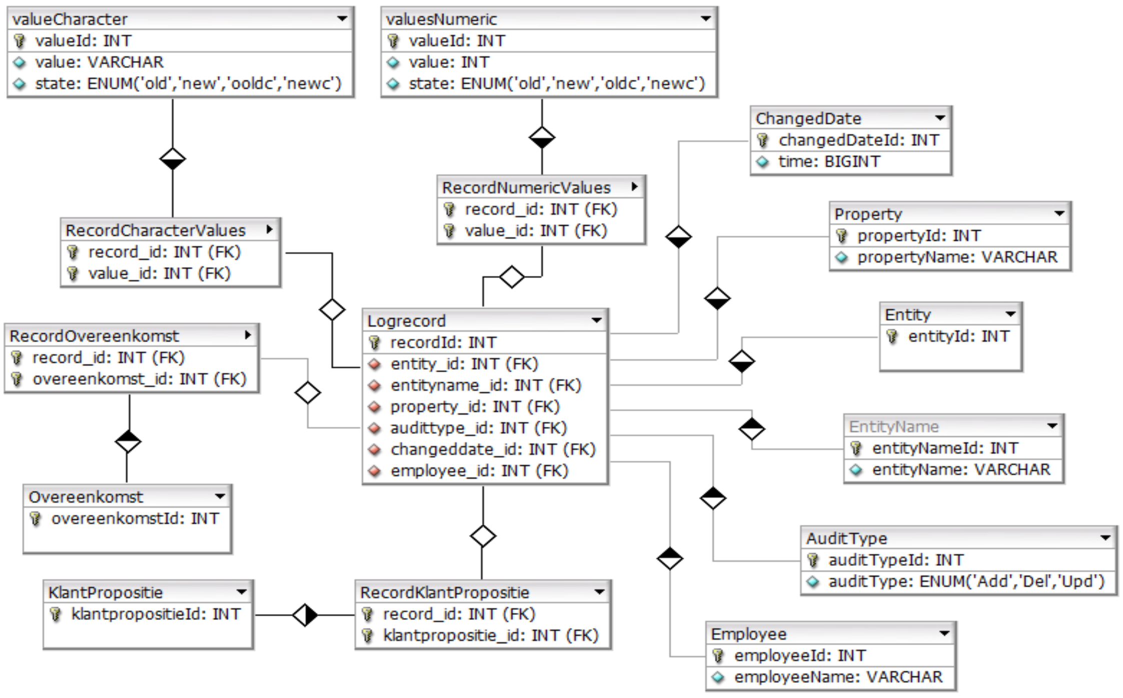


Figure 10: Optimized Normalized Audit Trail (ONAT) schema

The ONAT schema is the result of several test conversions that we have performed to discover areas for improvement of the original NAT schema, in terms of disk size. The NAT schema is the predecessor of the ONAT schema. The NAT was designed using the initial conversion principle (Figure 9) where the four exceptions, as explained above, were not applied.

4.4 Test conversion results

Test conversions are performed between the original audit trail log, the NAT schema and the ONAT schema. By means of the test conversions we evaluated how much the disk space would be reduced by both the NAT and ONAT schema in comparison to the original audit log as provided. During the test we converted 1 million records in batches of 100.000 records. The records were obtained randomly from the complete data set. The size on disk for the 1 million records is 224 Mb. Figure 11 and Figure 12 show the results of the test conversion.

Figure 11 shows the total size (in kb) of the data warehouses per batch. The original schema, our benchmark, increases the fastest, the NAT schema results in less disk space, but not as drastically as the ONAT schema. All schemas grow quite linearly as it can be seen in Figure 11. The decrease in disk size stays relatively constant over time. The data stored in the NAT schema reduces the disk size between 18 and 21 percent, where the ONAT schema reduces the disk size with 59 and 63 percent. The reason for this drastic difference between NAT and ONAT is the amount of obligatory fields. There are 7 fields identified as obligatory, which means 7 link tables can be removed saving 7 million records. This proves to be a drastic storage saver as well as a theoretical increase performance, due to the removal of several link tables.

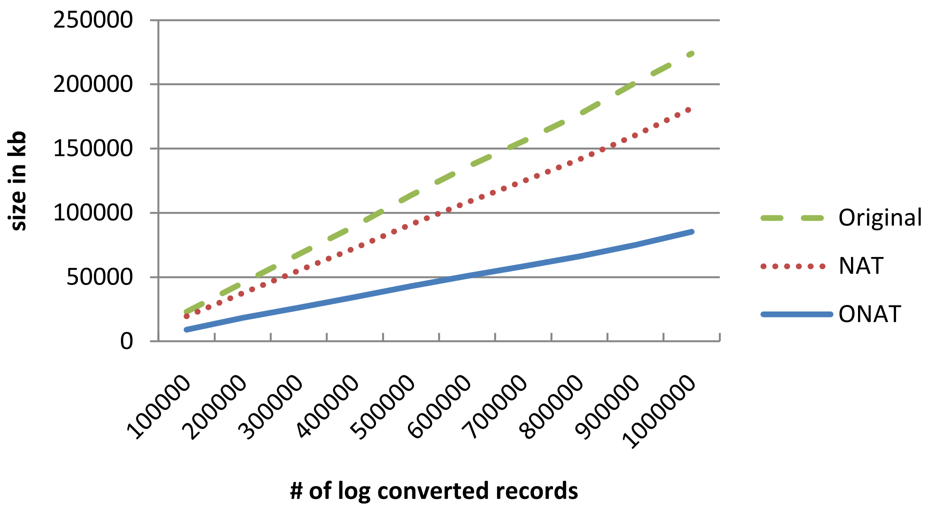


Figure 11: Total data warehouse size in kb per batch of 100k records

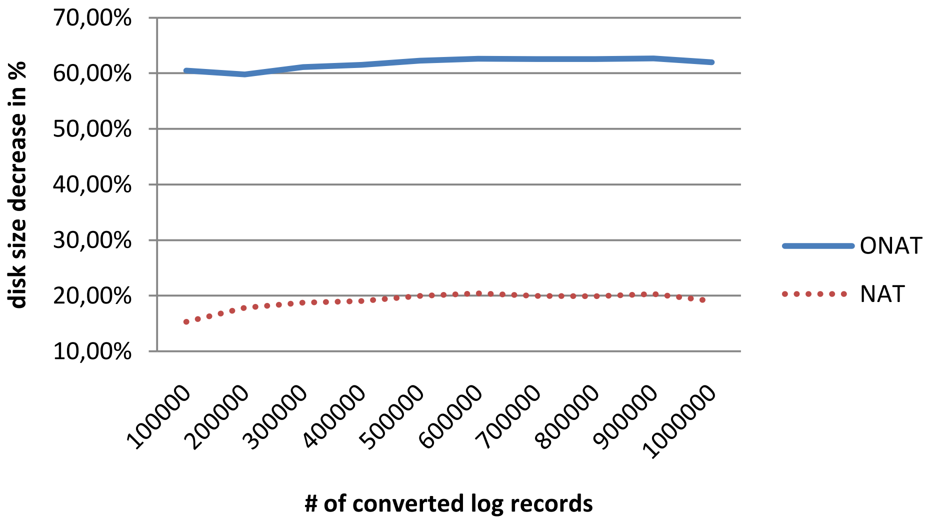


Figure 12: Total decrease in percentage per batch of 100k records

After converting 1 million log records, we see that the disk space for the ONAT schema is divided as shown in Table 6. A distinction is made between the link tables and the content tables. 96.23% of the data consist of link tables, which means only 3.77% of the data is considered as actual content. Some numbers on the disk size of the ONAT schema after conversion are shown in Table 6.

Table 6: Overview of the conversion results for the database schemas.

Disk Size of	Size in kb	Percentage w.r.t. original	Percentage w.r.t. ONAT schema
Original audit trail	224024	100 %	-
ONAT schema	85232	38.05 %	100 %
ONAT link tables	82019	36.61 %	96.23 %
ONAT content tables	3213	1.43 %	3.77 %

When logging an application for a long period of time, the chance that values from a new log record are not yet in the data warehouse generally gets smaller over time. From this assumption we predict that the percentage of content data decreases with respect to the size of the link tables. At some point, the data warehouse increases roughly linearly because the data warehouse always needs to store administrative information that links the values of a complete log record together.

4.5 Discussion

In the old situation, the audit trail produces roughly 130 Gb a year, in the original schema. Based on the test conversion we performed and the relatively constant results, we can now predict the total disk size for the data warehouse for the audit trail. If we take an average of 61 percent disk size decrease (the average of ONAT schema), we predict that the audit trail, after a year, would be around 50.7 Gb. The test log has, on average, 434,000 log records per day.

While performing the conversion test, we observed that the conversion takes longer each batch. We noticed that the conversion takes roughly lineary longer each batch. Although we have no facts to support this observation, we can reason about it to support these observations. When a new log record is converted, each value has to be checked for existence. When the data warehouse grows, the new value has to checked against more and more values. For the test data, 16 values have to be checked. We believe this is the main reason the conversion becomes slower over time. At start the roughly 400.000 records were converted in 30 minutes. At the end, the conversion handled about 100.000 in 30 minutes. These are indications through observation since it was not part of the tests. The 9.1 million records were converted in about 18 hours. Assuming the conversion performance goes down linear, we can conclude that the conversion handles 9.000 records less every 30 minutes. Assuming an application generates 450.000 records a day, the conversion reaches a point where it cannot process this amount in a day and from that point onwards, the conversion falls behind. Based on the observation, and linear increase in conversion time, the converter keeps up until the data warehouse holds around 4.5 billion records. That is, based on the average, 100 days of logging.

4.6 Conclusion

From the data warehouse schema design and the test conversion results we can conclude that the ONAT schema is a suitable schema for the audit log we have. Referring to the test conversion results in Table 6, the data warehouse holds less than 1,5% actual content (when normalized) in the data warehouse w.r.t. the disk usage of the original audit log. The rest of the data is administration for linking the values for each log record.

In case the data warehouse gets too large over time, or the conversion process cannot keep up the log generation speed, it is possible to backup only the link table. By doing so, the data warehouse loses about 96% of its size which means there is room for it to grow again. The link tables are merely used to be able to reconstruct a log record and can therefore be stored elsewhere without losing information, since the content remains. Another option is to investigate how the conversion process can be optimized.

Last, due to the simple structure of the data warehouse, the schema is generic which should facilitate interaction with the architecture that has been designed on top of the data warehouse.

From the tests on the test data we conclude the data warehouse already grows linearly after 100,000 records. Therefore we conclude the data warehouse is saturated in an early stage of the conversion. Once saturated, little new information is added by new log records that are added to the data warehouse. The data warehouse then grows linearly because of the administration costs to link a log record to the correct values. The linear growth is shown in Figure 11.

5 Audit Trail Question Model

The audit trail data is now stored in the ONAT data warehouse structure. To make the audit data questionable, an architecture is designed on top of the data warehouse. The architecture provides the functionality to let users ask *questions* about the data. The architecture then translates questions into queries. The architecture contains an internal model to represent the question that is translated into queries which provides the answer to the given question. In this chapter the model is introduced and explained together with the question language that is used.

Organization of this chapter Section 5.1 describes the approach that aims at answering the third research question. Section 5.2 is about the definition of the term '*generic model*' that is used throughout the chapter. In Section 5.3, a few possible questions, that could be asked about the audit data, are discussed. In section 5.4, we describe the language which we designed in order to ask questions about the audit data. Section 5.5 discusses the concept of labeling and how that concept is used in the architecture. Section 5.6 explains the meta model which is used as an internal model by the architecture. Finally, Section 5.7 holds the conclusion.

5.1 Approach

In order to answer the third research question (*"What is a generic architecture for handling audit trails?"*), the following has been done. We started by holding some interviews to understand the problem and to get a better insight in what information Topicus would like to get out of the audit trail logs. The information obtained from these interviews became the input for the designing an architecture that would solve the problem with the audit trail. We decided to define a model on top of the data warehouse that helps the user get information from the audit trail. Since the same problem can arise with other forms of logging, the goal is to come up with a generic model. Based on the gathered information, an architecture to support the questioning of the audit data is designed. Next to that, the architecture needs an internal model to represent the question. Last, we try to identify the weak points of the architecture and the model which are most vulnerable to change when the problem or domain changes.

5.2 Generic model

We designed a generic model to represent the questions throughout the architecture. To clarify, we define a definition that describes the term generic model. This definition is used throughout the rest of the chapter.

Definition 1*:* *Generic model*, A model which can be reused for the same purpose with different content in possibly a different domain. The model itself has no specific parts that point to the content it describes. Every equivalent problem can be represented by the same (generic) model.

The model we propose has been designed with this definition in mind. However, in order to realize that, the model needs to abstract from the actual content. To do this, we introduced labeling (from data provenance) and a specific language that is defined in which questions can be asked about the data.

5.3 Possible audit trail questions

From interviews with several stakeholders and supervisors within Topicus, we obtained possible questions. These questions give us an idea of the type of questions that the model should be able to handle. Some possible questions are:

- What is the current interest rate of invoice X?
- When was the last time property X was changed?
- When and by who was property X changed last?
- What is the change log for Object X?
- What are all interest modifications performed by "Jan"
- What are all properties from Object X that are changed more than 4 times for invoice Y.
- Who, and when changed the interest rate to value X for invoice Y?

After analyzing possible questions, we devised a structure in which all these questions could be represented.

5.4 Question Language

The language uses the concept of labeling from data provenance. Data is referred to by (user)definable labels and the language is based on the use of labels. The language asks questions about labels rather than actual the content. Therefore, a specific language is required. The questions will be asked in this so called *question language*. The language itself has two main goals: 1) To be intuitive and easy to use for the questioner. 2) To be relatively expressive without forcing the underlying model or architecture to lose their generic aspects.

The question language is mainly based on the notion of labels. Labels are translated into queries and refer to actual content. The labels themselves can be thought of as predefined sub queries that are represented by a label. These labels make the language more expressive than it might look at first glance. When comparing to SQL in general, with logging, queries that require a 'join' hardly occur because logs have a simple structure. Because of that, the question language can have less expressiveness then SQL for example. When defining the question language, the example questions are used to see if we can express then in terms of the question language.

The grammar for the question language is defined in EBNF. The complete grammar can be found in Appendix A. Below, the most relevant parts are shown to give an impression of the language. A question in the question language consists of two main parts: a *show* part and a *condition* part. The labels in the condition part describe the conditions that are applied to the data to select the correct subset of log records. The labels in the show part represent the information that will be shown about the records that were selected based on the provided conditions.

```
question          :     'show'! showLabels+ conditions*;
showLabels        :     optionShowLabel (showLabel)*;
optionShowLabel   :     IDENT;
showLabel         :     ','! IDENT;
conditions        :     'conditions'! leftSide (LOGOP
                  rightSide)*;
leftSide          :     condition;
rightSide         :     condition;
condition         :     '!'? IDENT (OPERATOR TIDENT)?;
```

```
LOGOP        :      'AND' | 'OR';
TIDENT       :      DOUBLEQUOTE IDENT WILDCARD? DOUBLEQUOTE;
IDENT        :      (LETTER | DIGIT)*;
```

A question can have multiple conditions connected to each other using the *'and'* and *'or'* keywords.

5.5 Labeling

In data provenance, labeling is used to label certain groups of data. Labels generally describe the data or some of its characteristics. Based on these groups, data can be easily looked up by referring to a label. In the proposed model, the idea of labeling is adopted. Before we can use labels in the questions, we devised a way to represent questions using labels. The goal is to distinguish different categories of labels. At first, two types of labels were defined, namely, *'simple labels'* and *'composed labels'*. Labels are mapped directly on a column in the underlying data warehouse. Composed labels are composed of other (composed) labels. Every composed label can be decomposed in, recursively, a set of simple labels.

After trying to apply these labels to the example questions, it became clear these two categories are not sufficient. A label can be used in two ways within a question. Generally when asking a question about data, you specify what you want to obtain from the result set and some conditions that affect the result. Based on that observation two main categories are defined, namely, *'showlabels'* and *'conditionlabels'*. Both categories can have a simple and a composed variant. Now it is possible to use the same label (name) in the show or condition context. For example, assume we have a label 'Administrators'. When the label is used as a *showlabel*, the employeeName and employeeId fields of the records in the result set are shown. If the label is used as a *conditionlabel*, the result set only contains log records for those employees who are administrator. The condition values for *conditionlabels* can be provided in two ways:

1) it is supplied inside the question by the questioner, e.g. Employee = "Jan".

2) The conditions are predefined.

Predefined means that the conditions are related to a specific label and therefore stored somewhere together with the labels. For example. the label 'Administrators' has the (predefined) condition that the employeeId's are '15' or '19'. The concept of a *'composed showlabels'* will be clarified below by means of an example:

Question 1.a: *Show* Employee *conditions* Administrators

The result of Question 1.a is the name and id of the employee's who are Administrator.

The same question can asked using 'Administrator' as a composed showlabel:

Question 1.b: *Show* Administrators

Now, the label Administrator consists of:

- A showLabel, which shows the name and id of employee's from the result set
- A condition part, which holds the condition that the employee's are administrator.

We have categorized the labels in which every example question can be represented. Now, these label categories must be defined in a model that can represent the structure of the question, in the question language, and also be able to represent the different label categories.

5.6 Question meta model

The question model is a generic model that is used in the proposed architecture as an internal data model. The question meta model is instantiated for each question. That model represents the structure of the question and the composition of the labels. The term 'model' here refers to any instantiation of the metamodel. The architecture that uses the model consists of several components which all contribute to populating the information for the model. The metamodel of the question model is shown in Figure 13.

The metamodel is explained by discussing the most important elements:

- *DataSource*: Holds information about the mapping on the database.
- *Condition*: Holds information about the characteristics of a condition, such as whether it is a negation, what condition values need to be applied, the type of values (range or exact values), the comparison operator ('=' '>' '=<' etc.) and to which DataSource needs to be applied.
- *SimpleConditionLabel*: A label with a name that is directly mapped on a field in the data warehouse. Optionally with a predefined condition for the particular label.
- *ComposedConditionlabel*: A label with a condition that consists of a set of other (Composed/Simple)ConditionLabels.
- *ConditionNode*: A node that either holds an 'And' or an 'Or' operator with a left and right side. The left and right sides can either be a (Composed/Simple) ConditionLabel or another ConditionNode. It forms a tree-like structure.
- *ShowLabel*: A label which holds a set of DataSources that represent the fields which are shown from the records in the result set.
- *ComposedShowLabel*: A combination of a ShowLabel and a set of ConditionLabels.
- *Question*: The 'root' element of every instantiation of the metamodel. Holds a set of Showlabels and the root node of the condition part tree.

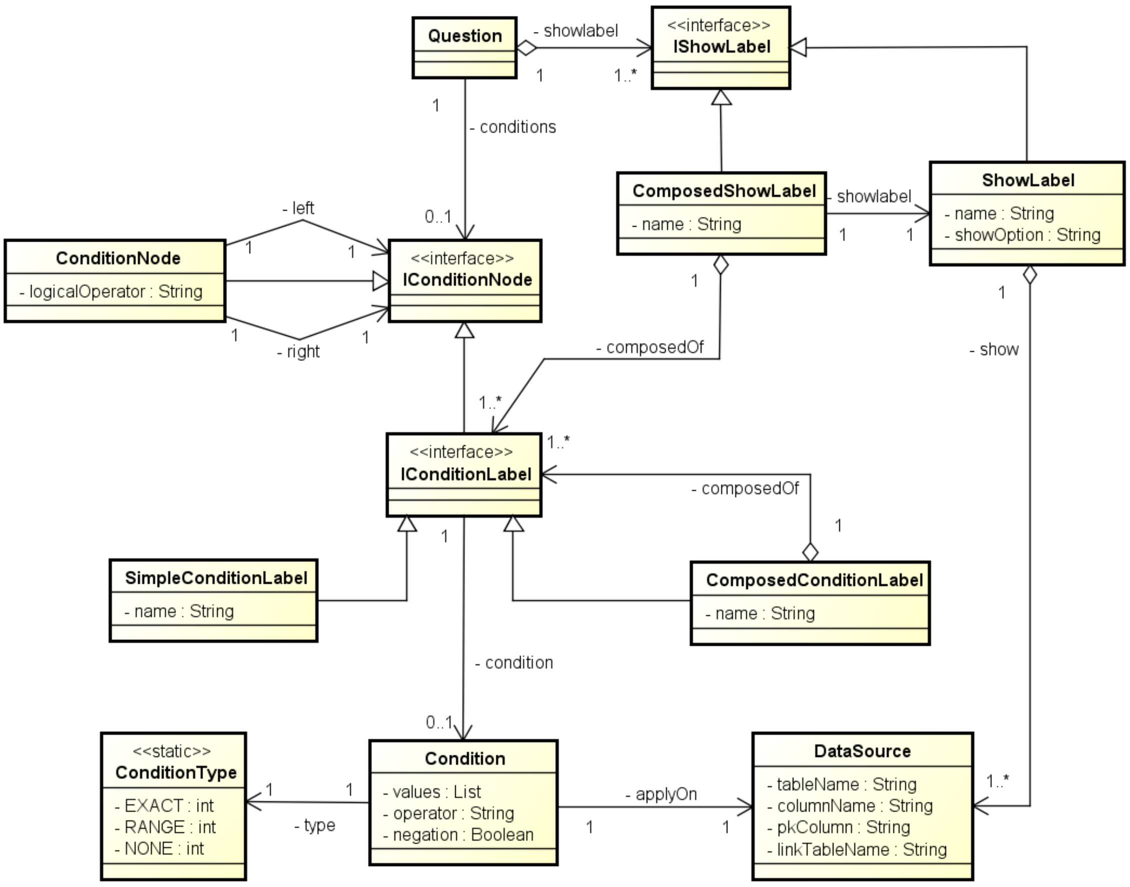


Figure 13: Question Metamodel

5.7 Conclusion

By comparing the definition of a generic model with the model that is defined we conclude that the model conforms to the definition and can be called generic and can be applied in many situations. The model has no specific knowledge about the data due to the concept of labeling. Labels make the model generic because all knowledge is attached to the labels which are configurable. The labels are the metadata the model needs to gain understanding of the data it handles. The (meta)model has knowledge about how to represent labels and how labels are structured (conditions and mappings). Further, the model has no knowledge specifically to the mortgage domain. Therefore, this model could be applied on any other domain

6 Audit Trail Architecture

The question model described in Chapter 5, is used in an architecture that supports the whole process from asking a question in a specific language to obtaining the correct results. In this chapter the architecture and its components are explained in more detail. The role of the model within the architecture is also discussed.

Organization of this chapter Section 6.1 describes the overall structure of the architecture, explaining what the function of each of the components is and how they interact with each other. Section 6.2 till Section 6.6 goes into detail for the components in the architecture and discusses their internal workflow. To clarify, the role of each components is explained using an example that is used throughout the chapter. Section 6.7 describes the areas of the architecture and the model which are most vulnerable to change. Further we discuss when the architecture is, and is not, not suited for a particular problem. Last, Section 6.8 holds a conclusion and some discussion about the architecture and the choices that were made while defining it.

6.1 Architecture

The architecture is responsible for the whole flow from accepting a question, analyzing the question and producing the correct answers. As described in the Chapter 5, questions are asked using labels in the question language. To get the correct answer to a question, four steps have to be taken:

1. Question must be parsed and analyzed.
2. Labels must be resolved to determine their meaning.
3. The question with the resolved labels must be translated into queries.
4. Possible post-processing and conversion to an output format has to take place.

The global architecture and its data flow are shown in Figure 14. Each step can be mapped onto a component in the architecture.

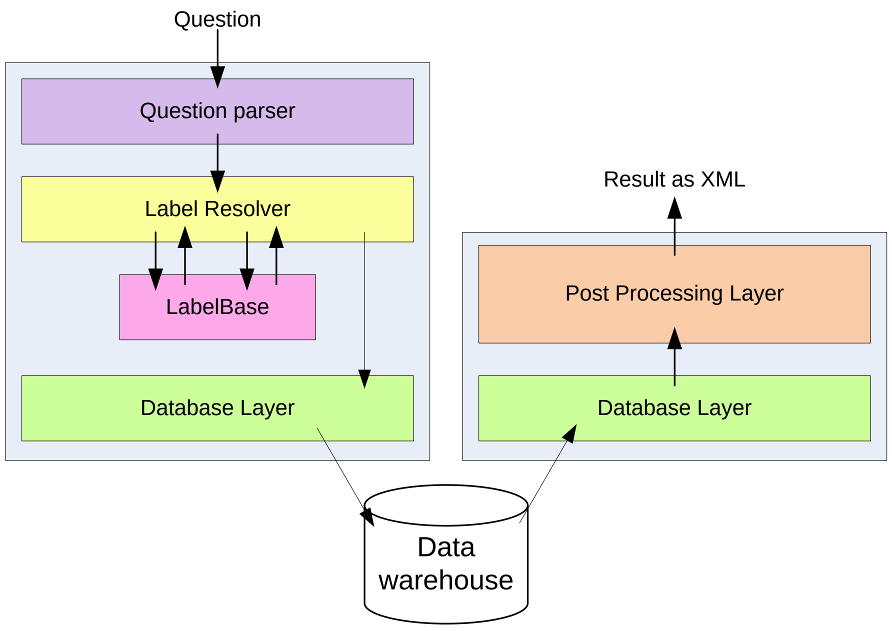


Figure 14: Components within the architecture

Each component in the architecture has its responsibilities supported by its functionality. The responsibilities of each components are outlined briefly below.

Question: The question is the input provided as plain text and conforming with to the specified question language.

Question Parser: The Question parser takes the textual input and parses the question using the defined grammar. The parsed question is then transformed to an instantiation of the question meta model that is used throughout the architecture. The model is filled with all available information, which mainly guarantees that the structure of the question stays unchanged. This internal model is called the Question model and is described in Section 5.6.

LabelBase: The LabelBase holds all knowledge about the labels. The mapping of the labels onto the content is stored inside the LabelBase. This mapping is; the mapping on the data warehouse, what conditions apply on a label, what fields need to be shown when a label is used as a *showlabel* and out of what labels *composedlabels* are composed. The input of the LabelBase is a textual description consisting of the label name and the type of the label (show or condition). The LabelBase returns all information that it knows about the label in a part of the Question Model.

Label Resolver: The Label Resolver takes the question model as input, and is responsible for identifying and resolving the labels which are used in the question. The LabelBase resolves the labels. The Label Resolver extracts all labels with their type (show or condition) and feeds that information to the LabelBase. The LabelBase returns the resolved labels in parts of the Question model. The Label Resolver collects all model parts of the resolved labels and merges that with the input model (which holds the structure of the question). The Question model is then complete and given as output.

Database Layer: The database layer is responsible for generating queries from the completely instantiated question model. In our case, the model is translated into MsSQL queries. The queries are executed onto the data warehouse and the results are obtained. The format of the result set depends on the implementation language. Every language has its own default format for storing results from the database. The result in a default format is given as output.

Post-Processing Layer: The post-processing layer provides the opportunity to insert modules for post processing of the result set, or other functionality, like a fraud analyzer or a converter to HTML or XML. When there is no post processing required, this layer does nothing other then converting the result set to some default standard like XML as output.

6.2 Question Parser

The internal workflow of the question parser is shown in Figure 15. The question parser gets a textual input that represents the question, in the question language. The question parser takes the grammar of the question language and parses, using ANTLR [19], the question into an Abstract Syntax Tree (AST). The AST holds the structure of the question and extracts its labels. The next step in the flow converts the AST into an initial model of the question using the question meta model. The result is an instance of a model for the specific question that is filled with all available information at this point, like the structure of the question and the label names. Here we assume that all labels are *simplelabels* because the Question parser is not responsible for the understanding of the labels.

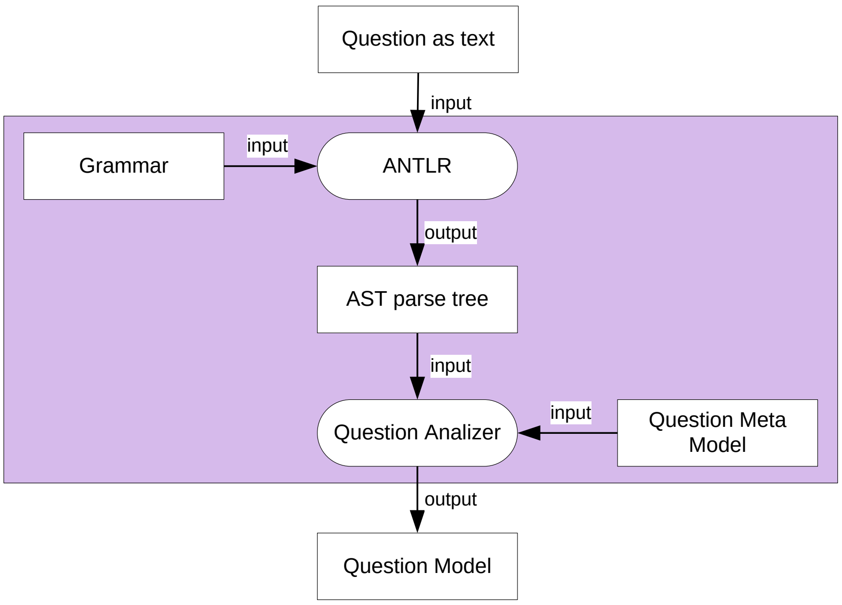


Figure 15: Workflow of the Question parser

Example:

To give an impression of how all the components in the architecture work together to produce a correct answer, an example is used throughout this section. After the explanation of each component, the output of that component is shown and explained. In this section the following example question is used:

"show Employee conditions Property = "Interest" AND inWeek5"

In natural language, we would like to see all employee's (id and name) that changed the property 'Interest' in the 5th week of 2011.

The object diagram in Figure 16 shows the instantiation of the model as produced and outputted by the question parser. It is a quite straight forward mapping from the AST, as displayed in Figure 17.

In the object model we see a clear distinction between the show and condition part. The question has a showlabel named '*Employee*'. The condition part has a root node, which is the AND- operator. On the left side of the AND-operator there is a condition label called '*Property*' that has a condition, namely that its value should be equal ('=') to '*Interest*'. On the right side, there is a label called '*inWeek5*'. At this point there is no information available regarding the condition. Now that the AST is converted, the model can serve as input for the next component, the Label Resolver.

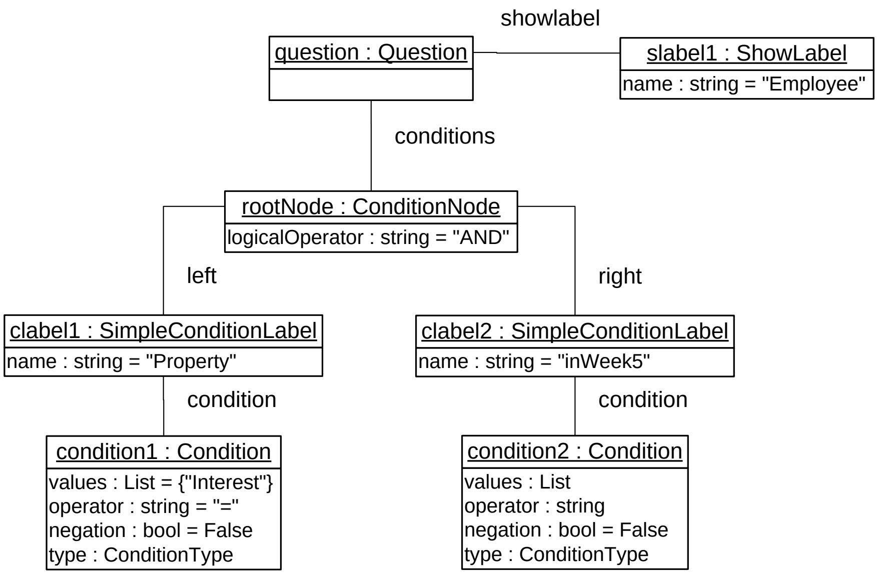
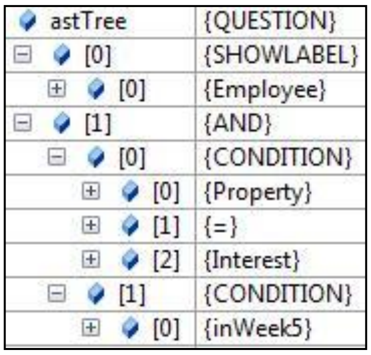


Figure 16: Resulting object diagram of the example question



astTree	{QUESTION}
[0]	{SHOWLABEL}
[0]	{Employee}
[1]	{AND}
[0]	{CONDITION}
[0]	{Property}
[1]	{=}
[2]	{Interest}
[1]	{CONDITION}
[0]	{inWeek5}

Figure 17: AST of the example question.

6.3 LabelBase

The meaning of labels have to be defined somewhere. This is done in the LabelBase. The LabelBase is responsible for the administration of the labels and their meaning. The LabelBase holds the following information about labels:

- The mapping of a label onto columns and records in the data warehouse.
- Predefined conditions attached to a label.
- Fields that have to be shown per label (in the case of show labels)
- Structure of composed labels.

The internal workflow of the LabelBase is shown in Figure 18. It accepts textual input which is the name of the label to be resolved. The output is a data structure for the resolved label, according to the Question meta model.

The LabelBase uses a cache in which it stores the already resolved labels.. There are three flows for the Labelbase are presented in Figure 18.

1. An event can trigger a process which resolves all labels from the LabelBase and puts it in the cache (dotted lines).
2. The label to be resolved, is already in the cache, so that the data structure already built for that label, is returned (solid lines).

3. The label is not in the cache, which means that the data structure has to be built using the question meta model and the information in the LabelBase database. Once resolved, the structure is placed in the cache (dashed lines).

The LabelBase is used by the Label Resolver component to obtain information about the meaning of the labels.

Within the LabelBase, the condition values are stored by its internal ids. In a normalized data warehouse, every value is uniquely identified by an id. When labels are added to the LabelBase, the exact condition values are resolved to their internal ids before being stored. By doing these lookups beforehand, these queries do not have to be executed each time a condition is used in a question, which yields faster results. For example, when a condition for Employee would be “Jan”, then “Jan” is resolved to its internal id (let’s say, 13). The LabelBase then holds the condition value 13 instead of “Jan”.

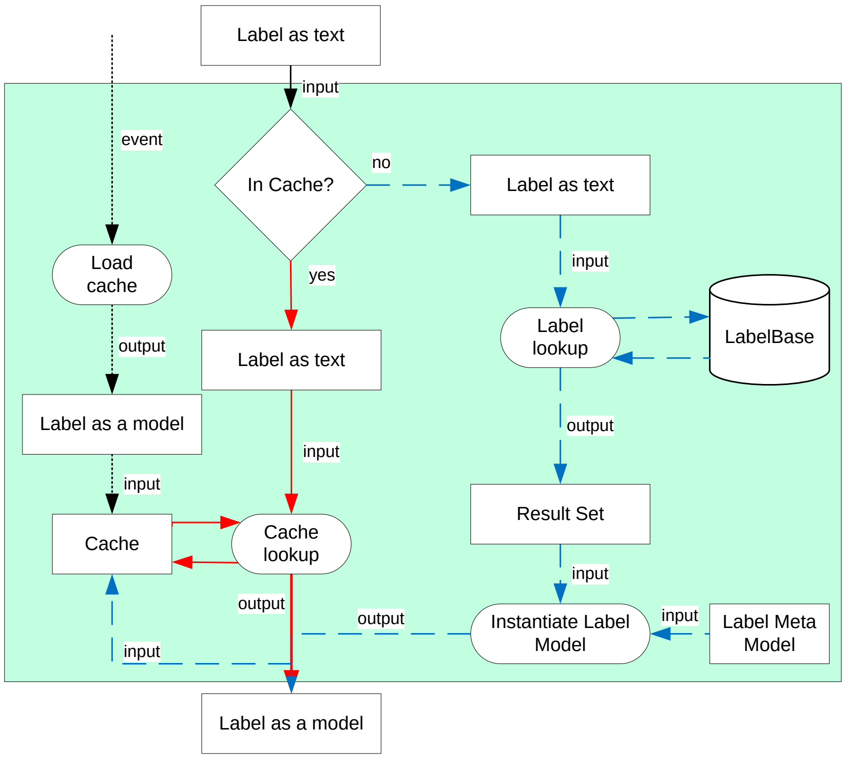


Figure 18: Workflow of the LabelBase

<u>Example:</u>

When the labels ‘*Property*’ and ‘*inWeek5*’ from the example question are given to the LabelBase to be resolved, The LabelBase produces a resulting data structure as is shown in Figure 19.

The Figure 19(a) shows the resolved ‘*Property*’ label. The property label seems to be a *SimpleConditionLabel* with no condition values, but the mapping onto the data warehouse is known. There was a condition value provided in the question, namely “Interest” and that value will be inserted later into the condition by the LabelResolver. The structure for a *SimpleConditionLabel* is the simplest data structure the LabelBase can return. Figure 19(b) shows the resolved label ‘*inWeek5*’. It turns out to be a *ComposedConditionLabel* composed of exactly one *SimpleConditionLabel* with a condition. The condition is an interval/range restriction on the '*ChangedDate*' label and has to be applied on the corresponding DataSource.

Once all labels are resolved using the LabelBase, the LabelResolver obtains all information that is required to complete the model instantiation.

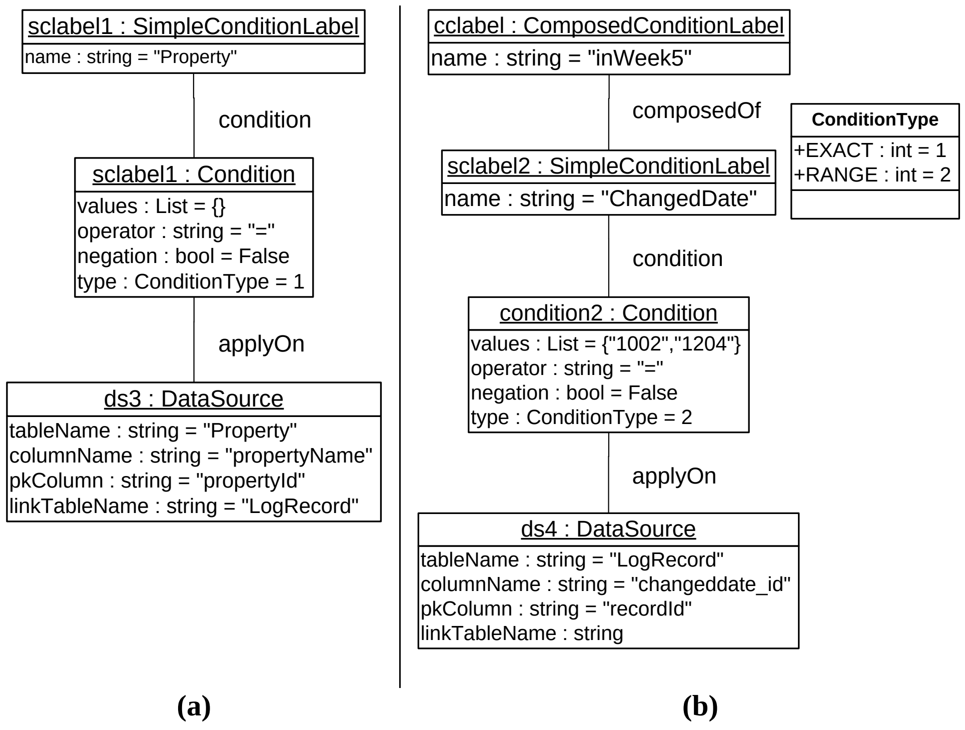

sclabel1 : SimpleConditionLabel
name : string = "Property"
condition
sclabel1 : Condition
values : List = {}
operator : string = "="
negation : bool = False
type : ConditionType = 1
applyOn
ds3 : DataSource
tableName : string = "Property"
columnName : string = "propertyName"
pkColumn : string = "propertyId"
linkTableName : string = "LogRecord"
(a)
cclabel : ComposedConditionLabel
name : string = "inWeek5"
composedOf
ConditionType
+EXACT : int = 1
+RANGE : int = 2
sclabel2 : SimpleConditionLabel
name : string = "ChangedDate"
condition
condition2 : Condition
values : List = {"1002","1204"}
operator : string = "="
negation : bool = False
type : ConditionType = 2
applyOn
ds4 : DataSource
tableName : string = "LogRecord"
columnName : string = "changeddate_id"
pkColumn : string = "recordId"
linkTableName : string
(b)

Figure 19: (a) Resolved label Property (b) Resolved label inWeek5

6.4 Label Resolver

The LabelResolver recieves a minimal instance of the question model as input and its responsibility is to complete the model using the LabelBase to resolve the labels and obtain the missing information. The workflow of the LabelResolver is shown in Figure 20.

The input is a minimal instantiation of the question model. First, the LabelResolver analyzes the model and extracts the labels. They are divided in two groups, namely *showlabels* and *conditionlabels*. Each of the names of the labels in these groups are sent to the LabelBase to be resolved. The result is a set with question model fragments, representing the data structure of the labels. The last step is to combine all those model fragments and merge them with the input model. The input model holds mainly the structure of the question. The meaning of the labels is then added resulting in a complete instantiation of the question model. The resulting model is the output for the LabelResolver component and is ready to be converted into queries by the Database layer.

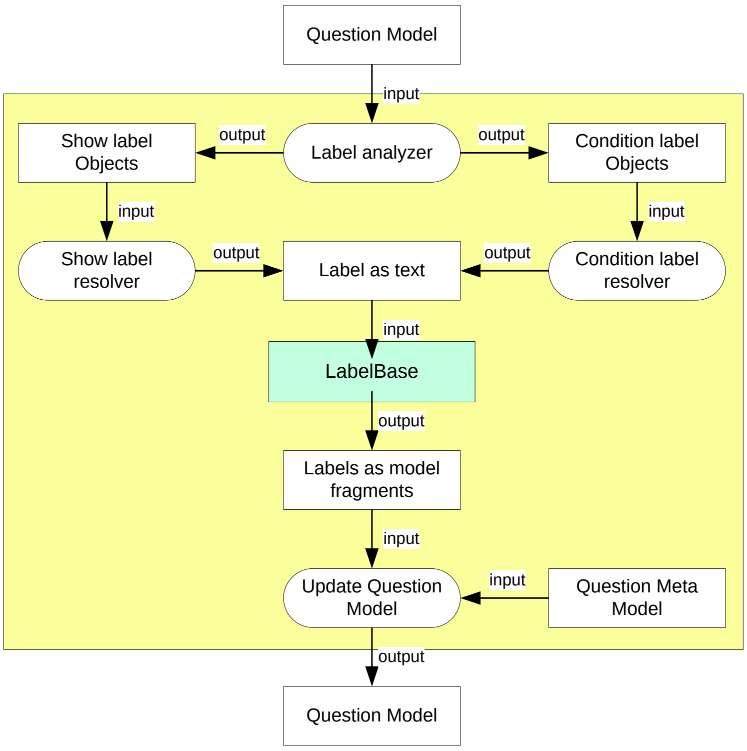


Figure 20: Workflow of the LabelResolver

<u>Example:</u>

The example question is structurally represented by the question parser, shown in Figure 16. After that, all the labels were resolved by the LabelResolver using the LabelBase, as shown in Figure 18, and then combined into a complete model by the last process of the LabelResolver, resulting in the model shown in Figure 21. From this model the queries can be generated and the result of the original question can be obtained.

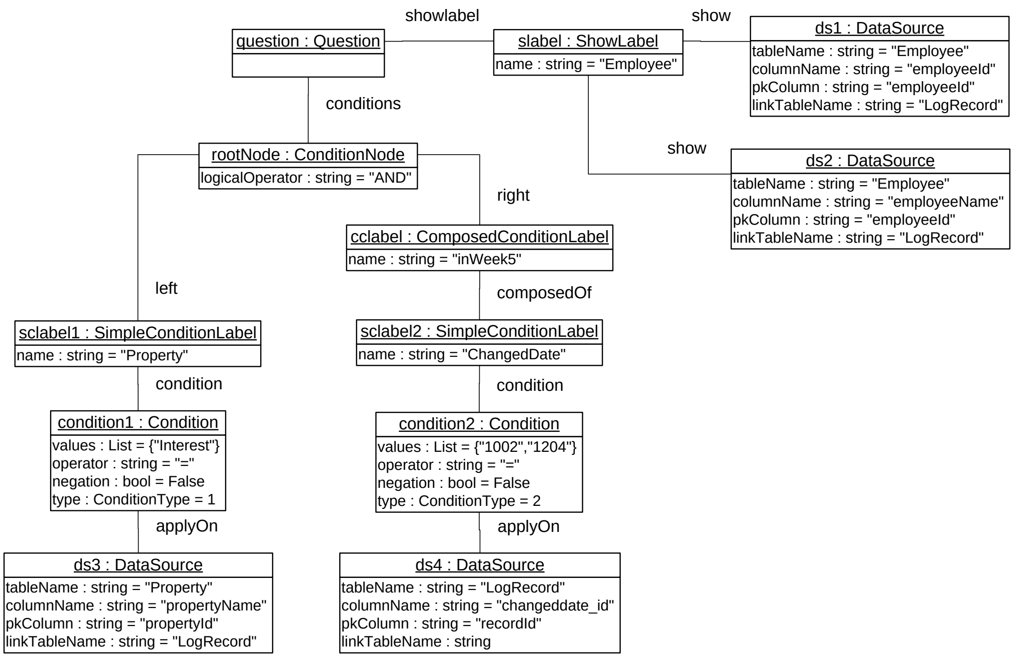

showlabel
show
question : Question
slabel : ShowLabel
name : string = "Employee"
ds1 : DataSource
tableName : string = "Employee"
columnName : string = "employeeId"
pkColumn : string = "employeeId"
linkTableName : string = "LogRecord"
conditions
show
rootNode : ConditionNode
logicalOperator : string = "AND"
ds2 : DataSource
tableName : string = "Employee"
columnName : string = "employeeName"
pkColumn : string = "employeeId"
linkTableName : string = "LogRecord"
right
cclabel : ComposedConditionLabel
name : string = "inWeek5"
left
composedOf
sclabel1 : SimpleConditionLabel
name : string = "Property"
sclabel2 : SimpleConditionLabel
name : string = "ChangedDate"
condition
condition
condition1 : Condition
values : List = {"Interest"}
operator : string = "="
negation : bool = False
type : ConditionType = 1
condition2 : Condition
values : List = {"1002","1204"}
operator : string = "="
negation : bool = False
type : ConditionType = 2
applyOn
applyOn
ds3 : DataSource
tableName : string = "Property"
columnName : string = "propertyName"
pkColumn : string = "propertyId"
linkTableName : string = "LogRecord"
ds4 : DataSource
tableName : string = "LogRecord"
columnName : string = "changeddate_id"
pkColumn : string = "recordId"
linkTableName : string

Figure 21: Complete model instantiation for the example question

6.5 Database Layer

The Database layer is responsible for generating queries from the complete model, which it gets as input. The dataflow for the database layer is shown in Figure 22. The database layer consists of two processes: The first is process translates the model to queries using mapping rules that are specific for the query language, in this case is MsSQL. The second process obtains and converts results from the data warehouse.

The mapping consists of a single complete query which, if executed, joins all tables together and returns all the log records of the whole data warehouse. The model-to-MsSQL process has a three steps:

- Generate queries from the condition labels of the model. This is mainly the 'where' part of an SQL query.
- Select the required parts of the query from the mapping, based on the show labels of the model. This is the 'select' part of an SQL query.
- Combine the generated condition queries with the selection queries, obtained from the mapping, creating the final query that answers the question.

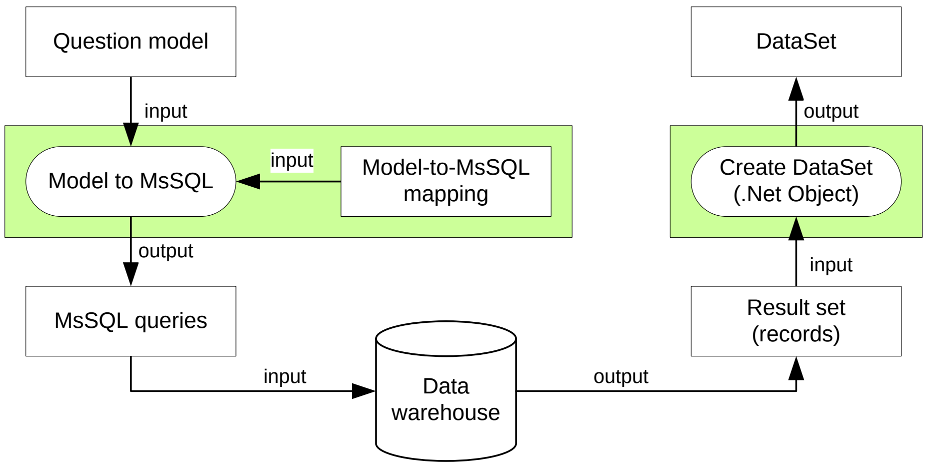


Figure 22: Workflow of the database layer

To convert an instantiated question model to MsSQL, a mapping is defined between the question meta model, Section 5.6, and MSSQL. Two processes are used to convert a model to MsSQL as explained before.

The 'select' part of the query is generated by looking at the DataSource objects of the *showlabels* in the model. The DataSource objects hold what fields need to be shown. Based on that information, the correct lines of MsSQL are selected from a main query. The main query returns all content in the data warehouse.

The condition (where) part of the query is responsible for collecting the record ids that meet the conditions specified in the original question. The show (select) part of the query is responsible of selecting the desired fields and corresponding values using the selected record ids.

The condition part of the query is generated from the model itself on runtime. Each condition part has the following syntax:

```
recordId IN (
      SELECT recordId
      FROM LogRecord
      WHERE ( ... )

)
```

Each *conditionlabel* in the instantiated model is evaluated. The *conditionlabel* obtains information from its condition object. The *condition* holds the condition values, the type (range restriction, static values) and whether or not the condition is negated (including or excluding the condition values). Using the *DataSource* objects, attached to conditions, we know to what fields in the data warehouse the restrictions need to be applied.

Once the final query is generated, the query is sent to the MsSQL server that hosts the data warehouse which executes the query. The result is then sent back to the database layer. The result is then converted into some data structure that is suitable for the specific programming language in which the architecture is written. In this case, a DataSet is a C# object holding all the returned records from the data warehouse. The DataSet is the output of the Database Layer.

<u>Example:</u>

In our running example, the database layer converts the instantiation of the model, shown in Figure 21 into queries. The show part and condition part contribute separately to the resulting query, as presented in Figure 23. The lines marked preceded with a dashed line pattern represent the lines of the query that are obtained via the show labels and the mapping of the database layer. The lines preceded by the solid line are generated from the condition labels of the model. They are combined, producing a single query that selects the log records that will answer the question.

```
SELECT
    lo.recordId as RecordId,
    em.employeeId as MedewerkerId,
    em.employeeName as MedewerkerNaam
FROM
    Property pr,
    Employee em,
    LogRecord lo
WHERE (
    lo.recordId IN (
        SELECT recordId
        FROM LogRecord
        WHERE EXISTS (
            SELECT propertyId
            FROM Property
            WHERE (propertyName = 'Interest')
            AND propertyId = property_id
        )
    )
    AND lo.recordId IN (
        SELECT recordId
        FROM LogRecord
        WHERE (changeddate_id BETWEEN '16' AND '20')
    ) .
)
AND pr.propertyId = lo.property_id
AND em.employeeId = lo.employee_id
ORDER BY lo.recordId;
```

Figure 23: The query generated from the model of the example question

6.6 Post Processing Layer

The workflow of the post processing layer is shown in Figure 24. This component is optional and provides room to add additional functionality. Some examples of additional functionality could be:

- Application of filters on the data as obtained from the database layer.
- Drawing statistics from the data.
- Adding plugins that connect the output to other systems like, for example, fraud detection systems.
- Output conversion, like conversions to XML or HTML, for displaying purposes.

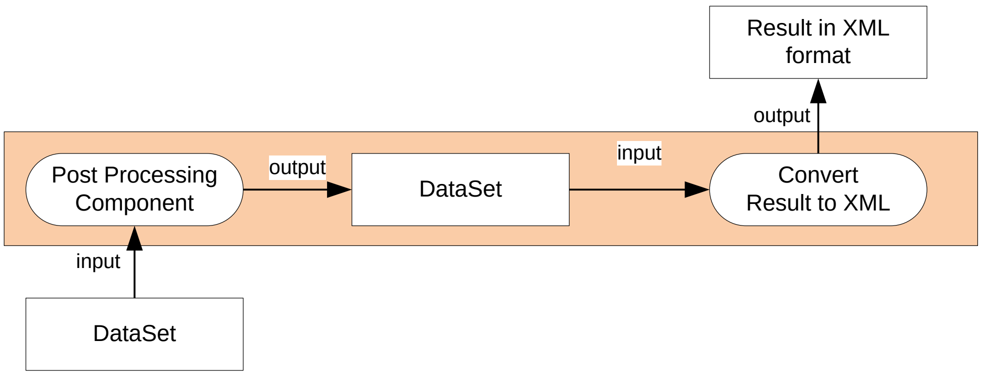


Figure 24: Workflow of the post processing layer

In case no additional functionality is added, the process converts the data to some standard format, like XML.

6.7 Possibilities and limitations beyond requirements

To check whether the architecture is applicable to different domains, scenarios are defined to identify the changes that may be required in order to apply the proposed architecture to another domain. Assume the following scenario:

Scenario: The architecture is used to question an access log of a company network. The logs are already stored in some MySQL database and a database conversion is not an option. The company would like to get statistics out of the data.

There are several changes in this scenario compared to our business case, namely:

- Different database language
- Different database schema
- Different content
- Analytic questions are asked

The different database language can be relatively easily solved by modifying the component which translates the question model to queries.

Under some conditions, the different database schema can be a problem. It relies on whether labels can be defined according the format of the architecture. Only in the case of complex table relations of multiple levels, the model cannot support the different database scheme. Depending on the database schema, the DataSource element of the metamodel might be modified, by adding information to realize the mapping between label and data. otherwise, the database needs a conversion or the architecture is not suited for the problem.

The difference in content is no issue, due to the fact that the architecture has no notion direct notion of content, because of the use of labels which hold the content/label mapping. The hard part lies in the definition of the labels. The labels should be defined by a domain expert together with someone who has knowledge about the structure of the logging/database.

The analytic questions can form an issue due to possible performance degration. The architecture is not specifically tailored to handle analytic questions. The problem can be covered by performing smaller queries and put the analytic functionality at a component in the post-processing part of the architecture. In theory it is possible, but performance might suffer because post processing is generally slower than letting a database perform analytic queries directly. If the performance becomes an issue, the question language needs to be extended. Based on the impact of the changes in the language, to architecture might need some change.

6.8 Conclusion

In this chapter we report on the architecture designed to support the process of asking questions about the audit trail data. The workflow of the individual components of the architecture are discussed and the responsibilities of the components are mentioned. To clarify the working of the components in the architecture, an example was used to show expected inputs and outputs of these components. From the example, we assert that the architecture works as intended. A question is asked in the question language, the architecture converts the question into the question model and all missing information is obtained from the Labelbase. Queries are generated and are executed on the data warehouse.

After identifying the weakness of the architecture and the limitations, we have a clear overview when the architecture can be applied, and when not, for a particular problem. From that, the conclusion is drawn that the architecture can be applied for the same problem in any domain without architectural changes. Most of the work will go in the definition of the labels.

7 Prototype

In order to evaluate the performance of the architecture, a prototype is built. Based on the test results we draw conclusions about the performance of the architecture. This chapter describes the prototype, the tests that are performed, the obtained results, their validity and conclusions we draw from the results.

Organization of this chapter: Section 7.1 describes the approach for answering the fourth and last research question: “*What is the performance increase (measured in units of time) of the proposed architecture?*”. Section 7.2 holds the requirements for the prototype, which were defined at the beginning of the project. Section 7.3 is about the design of the prototype and its scope. Section 7.4 describes some implementation choices that are made. Section 7.5 discusses the tests we defined, and the results are presented in Section 7.6. Section 7.7 concludes the chapter about the performance of the architecture based on the tests and their results.

7.1 Approach

In order to answer the fourth research question (“*What is the performance increase (measured in units of time) of the proposed architecture?”)*, the following has been done. After defining an architecture that theoretically solves our problems, we evaluate the performance of the architecture. To get information about the performance, we built a prototype to check the architecture would perform in practice. To test the theory, a prototype is built. The implementation of the prototype is kept to a minimum and captures only the mandatory and crucial parts of the architecture, as explained in Section 7.3.

The goal of the tests has been to assess how well the architecture performs in relation to the current audit trail implementation, which is a database. The tests consists of several questions about the data. First, the old database has been tested for performance against the new data warehouse. Then the architecture is tested, with the data warehouse underneath it, i.e., the architecture is built as a layer on top of the data warehouse. The tests give an indication of how much time the architecture takes on top of the execution times of the queries themselves.

Based on these tests, we estimated the performance increases with the new architecture and how the execution times are distributed over the architecture and the underlying data warehouse.

By analyzing the results of the tests, we draw some conclusions about the performance of our audit trail architecture.

7.2 Requirements

The requirements for the prototype were obtained by doing several interviews with employees of Topicus. The interviewed employees covered various roles in the organization like analysts, programmers, testers and management. The interviews were about the current implementation, current usage of its functionality, points for improvement and possible functionality that could be added. From the outcome of the interviews, the requirements were defined.

In this section we mention only the most important requirement regarding the prototype. The complete list of requirements can be found in Appendix C. The most important requirements are:

- *The prototype should retrieve information from the audit trail logs.* This is the main goal of the whole project. The prototype must be able to answer questions about the audit trail.
- *The prototype should increase performance for the retrieval of information without compromising the performance for logging records.* The storage performance does not change, since the old implementation stays intact. The data is transferred to the data warehouse on which the prototype runs. By implementing a prototype, we evaluated about the performance of the architecture of the proposed solution.
- *The prototype should have the functionality to filter and order information during or after retrieval.* Conditions can be defined using the question language. Based on the provided conditions, the data is filtered to present the correct information to the questioner. The results are always ordered by the record ids to get consecutive results in the order the changes occur.

7.3 Design

The design of the prototype has been based on the architecture, defined in Chapter 5. Due to limited time and resources, we kept the prototype to a minimum. This section describes the design of the prototype and discusses the scope of the prototype.

First, a grammar needed to be defined that defines the question language. This grammar was defined in ANTLR, and can be found in Appendix A. The grammar supports the complete language as described in Section 5.4. Only the bracket support is not included. The brackets can be used to group conditions like *((label a AND label b) OR label c)*. The support for brackets are left out, since the functionality is not relevant to the tests and has no impact on the test results.

Second, the prototype does not support adding, deleting or modifying labels via any form of interface or wizard. The database for the LabelBase is filled with a set of predefined labels. These are *simplelabels* that are directly mapped onto fields in the audit trail data warehouse. Other labels are used during the tests, which all use, or are composed of, simple labels. The LabelBase looks up every label that needs to be resolved and does not make use of caching. This indirectly means that the external event to fill the cache with all labels, as described in Section 6.3 is not supported either.

The LabelResolver component has been completely implemented as described in Section 6.4.

The Database layer needs to convert the model into queries. As described in Section 6.5, the process is divided into two phases:

1. The first phase generates the conditional part of the query, based on the conditions in the model. Since the condition part of the model is a tree-like structure, with a *ConditionNode* containing a left and a right side (see Figure 13). To realize that, the interpreter pattern[20] has been used.
2. The second phase uses a mapping to obtain the selection part of the query to be built. The mapping consists of a query that returns all data in the whole data warehouse. The query is manually defined, but could be generated, based on the data warehouse schema for the audit trail data (Figure 10). After the query is generated and executed on the data warehouse, the obtained results are stored in one of the standard data structures of the implementation language.

The post-processing layer in the prototype has two functions:

1. Convert the database results to HTML, so that it can be easily displayed.
2. Display unique log records from the result set.

Assume the example question: *"Show Employee conditions Administrator"*. The resultcontains employee information for all log records that meet the condition that the employee is an administrator. This means that if Employee A has 1 million records in the audit trail, his name ends up 1 million times in the result set. Intuitively, the questioner might not expect this behavior, but expects a list of the employees that are administrator, meaning that Employee A only occurs once in the result set. To overcome this situation, functionality has been added to the post-processing layer that filters the result from the database so that it will only show unique values to the questioner. Thus, for the example question, each employee occurs once in the result set.

7.4 Implementation

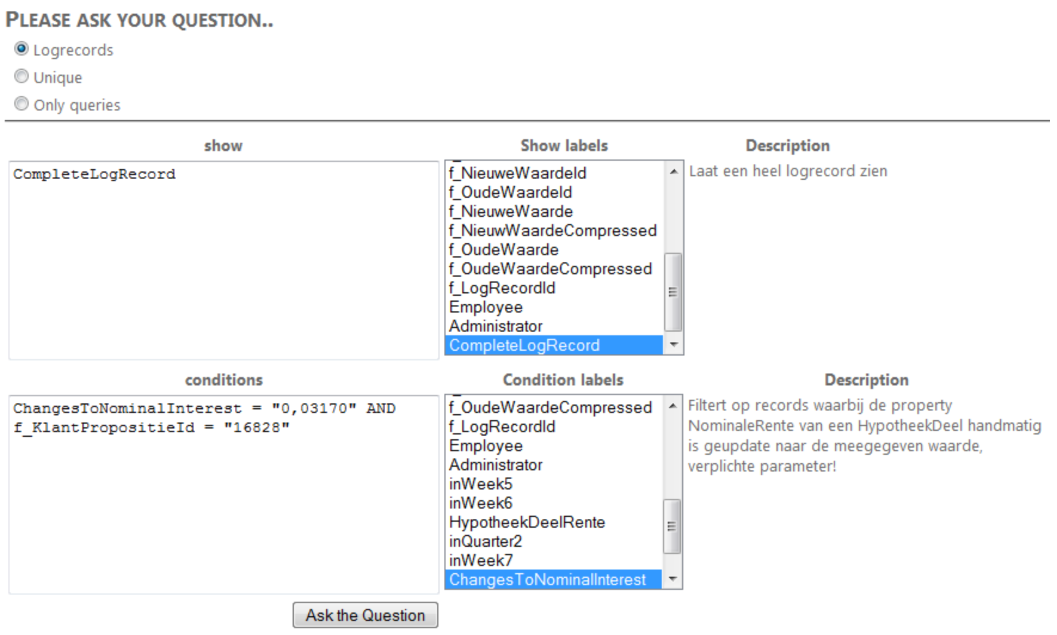
The implementation of the prototype is based on the original design as described in Chapter 6 but simplified as explained in the previous section (Section 7.3). During the implementation, some choices have been made and the most important ones are discussed in this section.

The main functionality of the prototype is to form questions about the audit trail data based on predefined labels. The prototype answers the questions and display the results to the questioner.

7.4.1 User Interface

For the prototype, the questions can be asked via a web interface. The interface is shown in Figure 25. On top, the questioner can choose how the results should be displayed. In the form of log records (each record that meets the conditions is shown in the result) or unique values (only unique results are shown).

In the textbox, labeled 'show', the *showlabels* are placed. In the list, the available *showlabels* are presented. If selected, a description is shown saying which columns of data the label is shown. The same goes for the conditions, where the description explains the conditions for a certain label. To form a question, the labels can be inserted in the textbox by double clicking on a label and by typing it in. When all labels are placed and possible condition values are entered, the question is stated and ready to be answered by the architecture.



PLEASE ASK YOUR QUESTION..

[✓ Logrecords]
Unique
Only queries

show	Show labels	Description
`CompleteLogRecord`	f_NieuweWaardeId f_OudeWaardeId f_NieuweWaarde f_NieuwWaardeCompressed f_OudeWaarde f_OudeWaardeCompressed f_LogRecordId Employee Administrator CompleteLogRecord	Laat een heel logrecord zien

conditions	Condition labels	Description
`ChangesToNominalInterest = "0,03170" AND f_KlantPropositieId = "16828"`	f_OudeWaardeCompressed f_LogRecordId Employee Administrator inWeek5 inWeek6 HypotheekDeelRente inQuarter2 inWeek7 ChangesToNominalInterest	Filtert op records waarbij de property NominaleRente van een HypotheekDeel handmatig is geupdate naar de meegegeven waarde, verplichte parameter!

Ask the Question

Figure 25: The prototype interface

7.4.2 Query generation

For the generation of the condition part of the query, a interpreter pattern is used. According to [20], the intention of the Interpreter pattern is: *To map a domain to a language, a language to a grammar, or a grammar to a hierarchical object-oriented design.*

The interpreter pattern for the prototype is implemented as shown in Figure 26. The interpreter pattern starts at the root with the Question object, and calls the interpret method. The question object calls the interpreter method of the root of the tree of the condition part of the model. From that point onwards it performs a recursive process until all leaves (labels) have been visited. The calls have a top-down approach, but the query generation is done bottom-up. The queries are built up while traversing back up the tree. Each parent node combines the queries generated by its leaves and passes it on to its parent in the tree. When the process gets back at the root, the condition queries are generated.

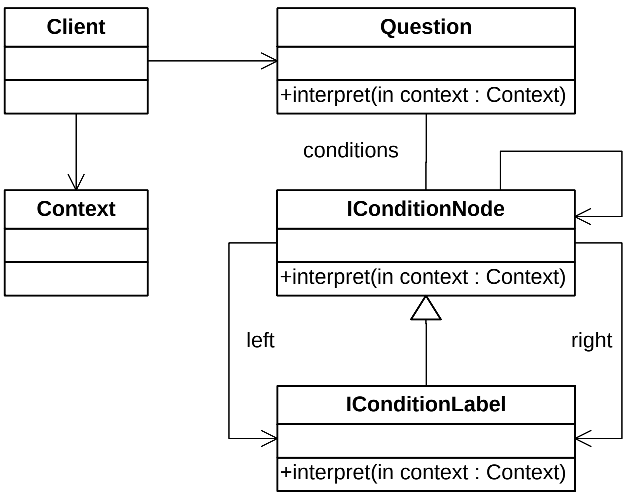

Client
Question
+interpret(in context : Context)
conditions
Context
IConditionNode
+interpret(in context : Context)
left
right
IConditionLabel
+interpret(in context : Context)

Figure 26 : Implementation of the interpreter pattern

For the show part of the model, the queries are generated differently. The database layer contains a model-to-MsSQL mapping. Within this mapping, one SQL query is defined that returns our complete data warehouse. In our case the query is defined manually, but it is possible to generate it using the data warehouse schema. This is a simple SQL query which is divided in five parts as shown in Table 7. The complete ‘main’ query can be found in Appendix B.

Table 7: Structure of the generated query.

Part #	Clause	Explanation
1	SELECT	The columns that can be displayed
2	FROM	The tables from which columns can be displayed
3	LEFT JOIN	The tables that need to be joined to the core (logrecord) table.
4	WHERE	Conditions that must be applied, these are generated and inserted later.
5	AND	Standard conditions that connect the tables to the core (logrecord) table that do not require a join.

The *Where*-clause part is generated using the interpreter pattern, as discussed before in this section. For the other four parts, the main query is separated per line and stored in a Dictionary. A Dictionary is a C# data structure (like a HashSet in Java), which is used as follows. Each record in the Dictionary has a key and a value. For the key we use the syntax “tablename.columname”. For the value, we used the corresponding line of SQL code.

The required information is obtained from the DataSource objects in the model. These DataSource objects hold information about the columns and tables onto which labels are mapped. By doing so, only the relevant parts of the initially main query are selected. Later, the conditional queries that were generated before are inserted, and the query is finished. By selecting only the relevant parts of the main query, unnecessary joins and selections are avoided, which improves the performance of the query.

After the query is executed, the results have to be shown via the webpage to the questioner. For simplicity, the results from the data warehouse are stored in a DataSet or DataTable, which are C# data structures. The content of these DataTables/DataSets are bound to a Grid on the webpage. Once the results from the data warehouse are obtained, the resulting records are rendered into the grid on the webpage, showing the answers to the question. If the questioner selected ‘Unique’ at the top of the page, the results go first through a filter. The filter eliminates all repetitive values before rendering the results into the grid that answers the question.

7.5 Performance test

After the implementation of the prototype, the tests suite that was designed up front, have been executed on the prototype. The test suite contains tests that give an indication about the performance of the architecture. The performance of the architecture is compared to the performance of the existing audit trail implementation. Identical questions have been asked to both onto the old database and the new data warehouse and via the architecture.

7.5.1 Test Approach

The goal of the tests is to evaluate the performance of the architecture and the data warehouse separately, as well as a whole. The results of the tests are compared to the performance of the original audit trail database using identical questions. The test questions are defined in a test suite (Section 7.5.2). The tests are performed on the following databases with the provided record count and size on disk shown in :

Table 8: Facts about the used databases.

Database	# of Records	Disk size
Original audit trail database	9.119.905	2.235.328 kb
Audit trail Data warehouse	9.119.905	848.520 kb

Audit trail logs can grow very large. The audit trails within Topicus produce currently about 130 GB a year, per application. Compared to these numbers, our test data set is rather small and not completely representative for the performance of an audit log in a year's time. Due to legislations concerning privacy we are not allowed to use the real audit logs. This forces us to make assumptions based on the test result that will be obtained with the test set that we have.

The tests have been performed on a laptop, because it is a stable environment and we can eliminate a few factors that might influence the results, like concurrent use of a server or network delay. After that, the tests have been repeated on a database test server. The server is faster than the local environment and has more memory. The server environment is still not a totally realistic environment, since the production environment has even better hardware but it is the most realistic environment available for testing. By performing the tests locally and afterwards on a server, the influence of more powerful hardware should reflect on the execution times and thus the performance. The interface application and architecture have ran on the laptop during all tests. Only the database is moved to the database server.

The tests will be conducted on systems with the specifications shown in Table 9:

Table 9: Specifications of the hardware in the test environment.

Specification	System1 - laptop	System2 – database server
CPU	Intel Core2Duo 2.1 GHz	Quad 2.83 GHz Intel Xeon
memory	2GB	8GB

The tests have been measured in units of milliseconds. For the database and data warehouse, the time is measured using a command from the MsSQL server. The following commands display the execution time of a query.

```
SET STATISTICS TIME ON
-- Query to be executed is placed here
SET STATISTICS TIME OFF
```

To ensure that caching does not influence the results, the cache is cleared before the execution of a query from the test suite. The cache is cleared by using the following MsSQL command.

```
dbcc dropcleanbuffers
```

For the architecture, the time is measured from the point that the architecture gets the question to the point the results are obtained from the data warehouse. The delay from the web interface is eliminated, because the rendering of browsers is usually inconsistent (time wise) and not relevant for the performance of the architecture itself. The architecture is responsible for the process that starts with receiving a question in the question language to delivering a result in its default format. Therefore only this process is measured.

The tests has been performed in the following order:

1. On the first system (laptop):
    a. Perform the tests on the original audit trail database.
    b. Perform the tests on the audit trail data warehouse
    c. Perform the tests on the architecture using the web interface.
2. On the second system (database server)
    a. Perform the tests on the original audit trail database.
    b. Perform the tests on the audit trail data warehouse
    c. Perform the tests on the architecture using the web interface.

Once the tests were performed, the results have been compared. We compared the difference in execution time between the old database and the data warehouse and how much time the architecture takes to convert a question, generate the models and generate queries. The same comparison is done for the tests on the database server. Last, the results on the different environments are used to evaluate the influence in the execution times by adding more powerful hardware. Based on the test results and observations on these comparisons, we draw conclusions about the performance of the architecture.

7.5.2 Test suite

The test suite is a collection of test cases, in this case a set of questions. Since the tests have been executed on two databases, the questions were translated for each system under test. The databases has a different structure, thus the queries have to be adapted to the specific database schema. The queries for the data warehouse are generated by the architecture. The queries for all systems had to be the same, so that the results can be compared. Once the results are identical, we can conclude that the same question has been asked for each system.

The queries for the test suite are selected based on the diversity in the number of results, question complexity (number of conditions etc.) and label complexity (composed labels, multiple predefined conditions etc).

The complete test suite can be found in Appendix D. The tests have been performed 12 times per batch. The highest and lowest times were eliminated and the average was calculated. The reliability of the results is determined by the standard deviation between the results. The results are found to be reliable when the standard deviation is less than 2.5% from the average. Whenever a test run has a higher standard deviation, the test should be done again.

The following questions are asked to each system under test:

1. Show the employee names and ids from the log records for which the employee is an administrator.
2. Show the unique employee names and ids that are administrator.
3. Show the employee names and ids from the log records from the seventh week of 2011, for which the employee is an administrator.
4. Same as question 3, but defined differently in the question language.
5. Show all the complete log records which have a Entity ‘HypotheekDeel’ and Property ‘RenteProduct’ or ‘VervolgRenteProduct’
6. Show all the unique properties of the Entity ‘HypotheekDeel’

7. Show the complete log records between 100400 and 100900.
8. Show all the employee names and ids of the employees who ‘changed’ the Property ‘NominaleRente’ from the Entity ‘HypotheekDeel’ into the value ‘0,03170’.
9. Show all the employee names and ids of the employees who ‘changed’ the Property ‘NominaleRente’ from the Entity ‘HypotheekDeel’ into the value ‘0,03170’ for invoice with number ‘16828’.
10. Show the complete log records for which the Property ‘NominaleRente’ from the Entity ‘HypotheekDeel’ ‘changed’ into the value ‘0,03170’ for invoice with number ‘16828’.

7.5.3 Test results

In this section, we present the results of the tests that have been performed on the original audit trail database, the data warehouse and via architecture.

The results are displayed in Table 10 and Table 11. Table 10, shows the test results obtained on the laptop environment, while Table 11 shows the test results obtained on the database server environment. The question numbers refer to the numbers in the test suite. The columns have the following meaning:

- The times in the ‘Original database’ column are the average execution times in ms for the queries on the original audit trail database.
- The percentages in the ‘std.dev original’ column represent the standard deviation for the execution times of the test run on the original database, based on the average.
- The times in the ‘Data warehouse’ column are the average execution times in ms for the queries on the data warehouse.
- The percentages in the ‘std.dev. Data warehouse’ column represent the standard deviation for the execution times of the test run on the data warehouse, based on the average.
- The times in the ‘architecture’ column are the average execution times in ms that the architecture needs to parse the question, create a model, convert the model into queries and possible postprocessing. These times do not include the data warehouse execution times.

The prototype runs, for all tests, on the same system. This means the prototype measurements are roughly the same for both systems. Only the databases ran on different systems. The standard deviation for the architecture is not mentioned because the execution times are so short, that standard deviation is relatively large. However, that does not mean that the results are unreliable. We are more concerned about the average times the architecture needs on top of the execution times of the queries.

	Original database (ms)	Std dev. original (in %)	Data warehouse (ms)	Std dev. Data warehouse (in %)	architecture (ms)
Q1	37143 ms	0,66 %	12724 ms	0,92 %	70 ms
Q2	37650 ms	1,95 %	12724 ms	0,92 %	127 ms
Q3	37627 ms	1,80 %	12019 ms	1,89 %	79 ms
Q4	37627 ms	1,80 %	12019 ms	1,89 %	42 ms
Q5	37537 ms	2,01 %	33445 ms	0,98 %	63 ms
Q6	37412 ms	1,13 %	13622 ms	1,33 %	364 ms
Q7	89 ms	7,16 %	20853 ms	0,77 %	28 ms
Q8	37552 ms	1,01 %	4531 ms	1,26 %	60 ms
Q9	38256 ms	2,07 %	14090 ms	1,20 %	76 ms
Q10	37418 ms	2,08 %	33335 ms	1,81 %	122 ms

Table 10: Test results on the laptop environment

	Original database (ms)	Std dev. original (in %)	Data warehouse (ms)	Std dev. Data warehouse (in %)	architecture (ms)
Q1	13830 ms	0,96 %	3242 ms	1,42 %	54 ms
Q2	13777 ms	1,00 %	3242 ms	1,42 %	151 ms
Q3	13768 ms	0,89 %	3004 ms	1,40 %	85 ms
Q4	13768 ms	0,89 %	3004 ms	1,40 %	41 ms
Q5	13791 ms	0,86 %	83664 ms	1,46 %	50 ms
Q6	13659 ms	0,84 %	5575 ms	1,56 %	359 ms
Q7	14 ms	13,58 %	5616 ms	1,18 %	28 ms
Q8	14070 ms	1,22 %	1611 ms	1,86 %	54 ms
Q9	13720 ms	0,59 %	3821 ms	1,87 %	82 ms
Q10	13766 ms	0,64 %	21363 ms	1,20 %	101 ms

Table 11: Test results on the database server environment

7.6 Discussion

From the test results, as shown in Table 10 and Table 11, there are a lot of comparisons possible. We will start with discussing the results on the laptop environment, shown in Table 10.

Laptop results

The original database performs rather steady, due to a full table scan for each query. Only Q7 is much faster. The obtained results for question 7 are 500 consecutive log records. This type of query seems to be really fast on a flat table.

The results for the data warehouse are quite constant, looking at the standard deviation, but the results between the questions differ. Question 5, 7, 8 and 10 stand out. Question 5, 7 and 10 produce complete log records as a result, which requires several joins to obtain all information. Joins are rather slow which shows in the execution times. Question 8 is by far the fastest. The question does not involve joins and has a very small result set (9 records), which gives a fast answer.

The times the architecture needs to parse the question, build up a model, generate queries and possibly do post processing on the obtained data seems to be quite fast. Most questions do not require more than a tenth of a second (0,1 sec). Question 2, 6 and 10 take longer. Question 2 and 6 need to do post-processing to filter unique results from the, possible, large result set. The filtering is performed by the architecture and therefore takes longer than other queries. Question 10 has a complex condition label, which takes a little bit more time. We expect that most of the time goes into obtaining the model for the label from the LabelBase.

Database server results

Looking at the results on the database server, Table 11, we observe, that the times are steady for all questions, again, with the exception of question 7 as explained before. For the data warehouse results, we make the same observations as the data warehouse results on the laptop. Again, Question 5, 8 and 10 stand out. The same goes for the results for the architecture. Apart from that, we observed nothing unusual.

Original database comparison

By analyzing the results of the original database execution times, we see that, by adding more powerful hardware, the performance roughly goes up by 63% (see Table 12). Only Question 7 is much faster, but this improvement is caused by the type of question which is more affected by the addition of better hardware.

	Laptop	Database Server	Improvements (in %)
Q1	37143 ms	13830 ms	62,77 %
Q2	37650 ms	13777 ms	63,41 %
Q3	37627 ms	13768 ms	63,41 %
Q4	37627 ms	13768 ms	63,41 %
Q5	37537 ms	13791 ms	63,26 %
Q6	37412 ms	13659 ms	63,49 %
Q7	89 ms	14 ms	84,42 %
Q8	37552 ms	14070 ms	62,53 %
Q9	38256 ms	13720 ms	64,14 %
Q10	37418 ms	13766 ms	63,21 %

Table 12: Original database comparison

Data warehouse comparison

By analyzing the results of the data warehouse execution times, we see improvements, as shown in the last column of Table 13. Most of the questions (1-4, 7 and 9) have an average improvement of around 73%. Question 6 and 8 have a little less improvement, but without a clear reason. The deviation is not that big to make it a big concern. Questions 5 and 10 deviate quite a lot from the average improvements. Question 10 requires 10 joins, which takes time. The results show that performing joins is less affected by the addition of more powerful hardware. Question 5 is slower on the database server than the laptop, which is a remarkable result. Question 5 also performs 10 joins, which could be the reason why this question was relatively slower than other questions, but this does not explain why more powerful hardware brings down the performance of the question. The question has been retested several times on different times, but the results stay the same. The question has no different characteristics compared to Question 10 and we could not find an explanation for this behavior.

	Laptop	Database Server	Improvements (in %)
Q1	12724 ms	3242 ms	74,52 %
Q2	12724 ms	3242 ms	74,52 %
Q3	12019 ms	3004 ms	75,01 %
Q4	12019 ms	3004 ms	75,01 %
Q5	33445 ms	83664 ms	-150,15 %
Q6	13622 ms	5575 ms	59,08 %
Q7	20853 ms	5616 ms	73,07 %
Q8	4531 ms	1611 ms	64,45 %
Q9	14090 ms	3821 ms	72,88 %
Q10	33335 ms	21363 ms	35,91 %

Table 13: Data warehouse comparison

Architecture comparison

From the results in Table 10 and Table 11, we do not see an improvement in architecture times. This is because the prototype always runs on the same system. Therefore the results are roughly the same. The architecture of the prototype does its job within two tenth of a second depending on how much post processing is required to filter results.

Original database versus the architecture

Looking at the previous comparisons, Table 14 shows the performance improvements between the original audit database and the architecture on the database server. The measurements as presented in the 'Architecture' column are the sum of the averages of the data warehouse plus the architecture measurements from Table 10 and Table 11. The comparison is done for both systems on which the tests have been performed to evaluate the influence of adding more hardware.

We observe an average improvement of 62 to 68% for the laptop environment and an average of 25 to 29% for the database server environment. Like in the preceding comparisons, the exceptions are Questions 5, 7, 8 and 10. These questions differ from the average for both test set results. The reasons why these questions differ has already been explained before.

An important observation is that the improvements on the database server are much lower than on the laptop environment. Based on that, we could conclude that adding more powerful hardware would reduce the degree of improvement, which sounds unlikely. From Table 12, we have seen that the addition of more hardware improves the measurements on the original database with 62 to 63%. From Table 13, we can observe that the measurements on the data warehouse increase with 73 to 74%. Since the data set is too small for a representative comparison, the measurements on the original database have a high percentage of improvement. The whole database does fit in memory, which speeds up the tests and gives faster results than when the database would not fit into memory. For this reason, the improvement percentages for the original database are too high to be really representative. Because of the unrealistic increase in performance, the performance improvements on the database server, shown in Table 14, are misleading. Based on the results of the data warehouse comparison, shown in Table 13, in which we see the performance improves generally with more than 70%, we can assume that with a larger data set the performance on the database server will improve 60% or more compared to the original database performance.

	Laptop			Database server		
	Original database	**Architecture**	**Improvements Laptop (in %)**	**Original database**	**Architecture**	**Improvements Database server (in %)**
Q1	37143 ms	12794 ms	65,55 %	13830 ms	3296 ms	28,36 %
Q2	37650 ms	12851 ms	65,87 %	13777 ms	3393 ms	27,96 %
Q3	37627 ms	12098 ms	67,85 %	13768 ms	3089 ms	28,38 %
Q4	37627 ms	12061 ms	67,95 %	13768 ms	3045 ms	28,50 %
Q5	37537 ms	33508 ms	10,73 %	13791 ms	83714 ms	-186,28 %
Q6	37412 ms	13986 ms	62,62 %	13659 ms	5934 ms	20,58 %
Q7	89 ms	20881 ms	-23361,80 %	14 ms	5644 ms	-6325,84 %
Q8	37552 ms	4591 ms	87,77 %	14070 ms	1665 ms	13938,20 %
Q9	38256 ms	14166 ms	62,97 %	13720 ms	3903 ms	25,66 %
Q10	37418 ms	33457 ms	10,59 %	13766 ms	21464 ms	-20,12 %

Table 14: Comparison original audit database and complete architecture

Scalability

Since we were not able to perform tests on a real-life environment, we make assumptions about the scalability of the architecture and the effects on the performance. We know that the audit log of 64 GB (the one that runs in the production environment of Topicus, obtained from 6 months of logging) cannot be queried at all. When converting the data to the data warehouse, the data would have a size of around 39 GB (61% decrease in size due to the obtained information about the test conversion in Section 4.4). Based on the disk size in the data warehouse (4% content, 96% link tables, shown in Table 6) there is about 1.6 GB content to go through when questioning the data, divided over several tables. From that we expect that the architecture can handle a log of at least 1 year without breaking down. The bottleneck will probably lie in the amount of link tables in the data warehouse and the number of joins that are required for a particular question. The 64 GB log will have roughly 275 million log records. In case several tables must be joined, the performance is expected to decrease, as seen in our own results. We expect that the link tables are too large to be able to perform all the joins.

7.7 Conclusion

Based on our tests and observations, we can conclude that the proposed solution improves the performance of questioning the data, on average between 70% and 75%. However, the time it takes to answer a question is closely related to the type of question: the more specific the question the faster the results are obtained. When a question involves a lot of joins with a lot of records in the result set, the performance decreases rather fast. From our tests and observations, we conclude that the strength of the architecture is to quickly identify the correct records that answer a question, but reconstructing a record and obtaining the correct values corresponding to the identified records is the bottleneck. The fastest results are obtained by asking really specific questions preferably with a small amount of records as result. Selecting columns to be displayed (showlabels) that are not relevant for the answer (or to the questioner) should be avoided.

We conclude that the architecture performs better than the original audit trail implementation. Apart from that, the architecture will be able to handle a lot more data before it breaks down or performance tweaks are required. Nevertheless, we cannot state when that point is reached.

8 Conclusion and Future Work

This chapter provides a final conclusion based on our research, the results of the tests and the observations made while performing the tests. We provide answers to the research questions which were formulated at the beginning of the project, and propose directions for future work.

Organization of this chapter: Section 8.1 provides answers to the research questions. First, we answer the sub questions and then we answer our main research question. In Section 8.2 proposes directions for future work.

8.1 Subquestions

Q1: What is an audit trail? The audit trail, as explained in Section 2.1 is a form of logging in which everything is stored chronologically. It can be seen as a change log. The audit trail is a very detailed log, usually it logs every change made to a database or data model of an application.

Q2: What data warehouse architecture is suited for storing audit trail logs? In order to answer this question, a theoretical comparison is done on several data warehouse architectures. Based on criteria that were defined, each architecture was evaluated and the architectures have been compared. From the comparison, it turned out that a normalized data warehouse architecture suits the problem best. By normalizing logs, which have a high redundancy rate in general, redundancy disappears. Without redundancy, records that need to meet certain conditions are obtained much faster, since there is less data to process. Also, after logging for a longer period, the amount of new data to the data warehouse is limited, since most data is redundant and already present in the data warehouse. Because of that, the data warehouse does not increase linearly, but decreases more exponentially over time.

Q3: What is a generic architecture for handling audit trails? This question has been answered by answering the following subquestions:

Q3.a: What meta data is required by the architecture to obtain understanding of the data? To create a generic architecture that could be used for any audit trail, we had to find a way to talk about the data. The data could be about everything. To cover this issue, the concept of labeling is introduced. Using labeling, the meaning of data can be captured by means of a label. The architecture does not know anything about the data, merely labels, which can occur in different forms. The main two types of labels are *showlabels* and *conditionlabels*. By making the the mapping between labels and data configurable, the architecture becomes more generic. All information the architecture needs to know is how labels are mapped onto data. To cover that, the LabelBase component has been introduced. The architecture uses LabelBase to resolve the labels to find out their meaning to gain understanding of their data.

Q3.b: What does a (generic) model, to represent audit trail data, look like? By using the concept of labeling, the meaning of data can be captured by means of a label. Labels are defined in two main categories, namely *showlabels* and *conditionlabels*. Since the audit data can be addressed by a label, a model is defined that describes labels rather than actual data. By doing so, the questions about the data is based on labels rather than knowledge about the actual content. This makes it easier for questioners to ask questions about data, since there is no knowledge required about the data or how it is stored in the data warehouse. The architecture needs a model to represent the different types of labels, the (structure of the) question and the mapping of the labels onto data after being resolved. We defined a model for this purpose and this model represents questions in the form of labels rather than actual data. The model (and architecture), can be reused for other audit trails as well. Our question model is described in Section 5.6.

Q3.c: What architectural changes to the model have to be made in order to make it domain specific? In order to use the architecture, as proposed in Chapter 6, some configurations are required in order to make it work for a different audit trail. First, the audit trail data must be prepared to fit in the structure of the data warehouse. The structure is in a rather straightforward normalized form explained in Section 3.2. Second, the mapping between labels and data must be defined in the LabelBase component. *Simplelabels* are one-to-one mappings between labels and columns in the data warehouse. These labels become the set of labels from which new labels with conditions can be derived. Third, extra functionality can be inserted in the post-processing component of the architecture. Apart from that, there are no structural changes required to the architecture that cannot be configured when the domain is changed.

Q4: What is the performance increase (measured in units of time) of the proposed architecture? To answer this question, tests have been performed on a prototype. From these tests, we conclude that the architecture speeds up the questioning process with generally 62 to 68%, depending on the type of question. The performance is affected mostly by the amount of columns the questioner requires. Obtaining a complete log record takes much more time than obtaining one or two columns, with the same conditions. The performance is also influenced by how specific a question is. A lot of conditions and possibly a small result help produce fast results. The time the architecture needs, excluding query execution times, is shown to be minimal. Without post processing, the architecture performs its tasks within a quarter of a second. Based on the test conversion (Section 4.4) we know that the data warehouse grows slower than the original audit trail database. It takes longer before the architecture, or data warehouse, breaks down and is not be able to answer questions. However, the exact moment when that point is reached cannot be determined based on the performed tests and obtained knowledge during this project.

8.2 Main question

What is a generic architecture for efficient questioning of audit trail logs?

Based on a our research using a business case from at Topicus, our goal was to define a model that could work with audit trail logs and improve the performance in data retrieval times. The audit logs contain a lot of detailed information. Getting desired information from the data required knowledge of the data as well as a lot of effort, time wise. Since all audit trail loggings have the same underlying general structure, we tried to abstract from the business case aiming to create a solution that is applicable to any audit trail log. During this research we have defined a model that could handle any audit trail log once stored in the data warehouse. To abstract from the actual content and increase the generic aspect of the architecture, labeling was introduced. Because the meaning of the data has to be defined in the LabelBase component, the architecture itself has no specific knowledge about the data for the business case. Therefore, we conclude that, once the audit data can be restructured to fit in the data warehouse as defined in Section 4.3.2, the architecture can handle any audit log. Using tests on the available data we got some insights about the time performance of the prototype. Based on the tests we can conclude that the architecture increases the performance drastically in comparison with the old implementation and thus improves the time-efficiency of questioning the data. Due to lack of more test data and legislations, we cannot perform tests on the most realistic environment to get an indication of how the performance scales over time, when the data increases. However, by extrapolating the execution speed of the architecture and little growth of the data warehouse when adding more log records, we predict that the architecture can handle a lot more data before the architecture reaches its limit. The limit of the architecture should be determined in future work.

8.3 Future Work

Based on the results of this research, we propose some directions for future work.

First, the performance of the architecture could be tested on a stable server environment with more log data to evaluated how the performance scales over time. Next to that, the queries which are generated by the architecture might not always produce the optimal and/or fastest queries. There might be room for improvement on the queries generation process to increase execution times.

Second, the architecture should be extended with functionality to facilitate the definition of the labels. A possible direction is, when a questioner defines a question with a complex condition part with several labels and conditions. With a simple click on a button, the condition could be saved as a new condition label in order to be reused later.

Third, possible applications for which the architecture can be used, could be researched such as, a system that detects fraud. Such a system could use the architecture to obtain data that can be used in the fraud detection process. Fraud is an important issue in the financial world. Other forms of analysis of the audit trail data can be researched. One could also investigate how the architecture could contribute in that process.

References

[1]. **American Institute of Certified Public Accountants (AICPA).** SAS70 Standard. 1992. http://www.aicpa.org/soc.

[2]. **Harleman, T.** *Research Topics - Audit trail.* University of Twente. 2011.

[3]. **Su, L.** *User Behaviour Based Access Control Decision.* Inner Mongolia University. Hohhot, China : s.n.

[4]. **Simmhan Y., Plale B., and Gannon D.** *A Survey of Data Provenance in E-Science.* SIGMOD Record. 2005. pp. 34:31–36.

[5]. **Buneman P., Khanna S., Wang-Chiew T.** *Why and Where: A Characterization of Data Provenance.* University of Pennsylvania. 2001.

[6]. **Cernosek G., Naiburg E.** *The value of modeling.* IBM. June, 2004.

[7]. **P.J., Best.** *Project Frodo Progess Report.* Queensland University of Technology. 2005.

[8]. **Mounji A., Le Charlier B., Habra N., Mathieu I.** *ASAX: Software Architecture and Rule-Based Language for Universal Audit Trail Analysis.* University of Namur. 1992.

[9]. **Mounji, A., Le Charlier, B., Zampunieris, D., Habra, N.** *Distributed Audit Trail Analysis.* Los Alamitos, CA : Proceedings of the ISOC'95 symposium on network and distributed systems security, 1995.

[10]. **van der Geest, T., Wieringa, G., Viegen, R.** *De Nederlandse hypotheekmarkt, De Hypotheekprofessional part 1.* Februari, 2008.

[11]. *AFM.* Autoriteit Financiële Markten. http://www.afm.nl/en/over-afm.aspx.

[12]. *BKR.* Registratie, Bureau Krediet. http://www.bkr.nl/.

[13]. *FluidDB.* http://fluidinfo.com/.

[14]. *InfoBright.* http://www.infobright.com/.

[15]. **Hightower, R.** *Data warehouse architecture.* University of Florida. September, 2009.

[16]. **S., Badiozamany.** *Microsoft SQL Server OLAP Solution – A Survey.* Department of Information Technology, Uppsala University. Sept 2010.

[17]. *Tickery.* FluidInfo. http://tickery.net/.

[18]. **InfoBright.** *High Performance Log Analytics: Database Considerations.* March 2010.

[19]. *ANTLR.* http://www.antlr.org/.

[20]. *SourceMaking.* http://sourcemaking.com/design_patterns/interpreter.

Appendix A

The Antlr Grammer for the Question language

```
tokens {
      QUESTION;
      SHOWLABEL;
      CONDITION;
      TOP;
      SHW          =     'show';
      COND         =     'conditions';
      DOUBLEQUOTE =     '"';
      COMMA        =     ',';
      WILDCARD     =     '%';
      NOT          =     '!';
      MIN          =     'MIN';
      MAX          =     'MAX';
      TOPTOKEN     =     'TOP';
}

public start       :     question -> ^(QUESTION question);
question           :     SHW! showLabels+ conditions*;
showLabels         :     optionShowLabel (showLabel)* ->
       ^(SHOWLABEL optionShowLabel) ^(SHOWLABEL showLabel)*;
optionShowLabel    :     opt? IDENT;
opt                :     MIN | MAX | top -> ^(TOP top);
top                :     TOPTOKEN TOPNUMBER;
showLabel          :     COMMA! IDENT;
conditions         :     COND! leftSide (LOGOP^ rightSide)*;
leftSide           :     condition -> ^(CONDITION condition);
rightSide          :     condition -> ^(CONDITION condition);
condition          :     NOT? IDENT (OPERATOR TIDENT)?;

LOGOP        :     'AND' | 'OR';
OPERATOR     :     '=' | '>' | '<' | '>=' | '<=';
LETTER       :     'a'..'z'|'A'..'Z' ;
DIGIT        :     '0'..'9' ;
TIDENT       :     DOUBLEQUOTE IDENT WILDCARD? DOUBLEQUOTE;
TOPNUMBER    :     (DIGIT)*;
IDENT        :     (LETTER | DIGIT)*;

WS           :   (' '|'\t')+ {Skip();} ;
```

Appendix B

The complete query that returns all the content of the data warehouse and which is used in the model-to-MsSQL mapping in the database layer component.

```
SELECT
      lo.recordId as Id,
      at.auditType as AuditType,
      cd.changeddate as Datumwijziging,
      en.entityName as Entiteit,
      kp.klantpropositieId as KlantpropositieId,
      em.employeeId as MedewerkerId,
      em.employeeName as MedewerkerNaam,
      vcn.value as NieuweWaarde,
      vnn.value as NieuweWaardeId,
      vco.value as OudeWaarde,
      vno.value as OudeWaardeId,
      pr.propertyName as Property,
      e.entityId as EntityId,
      vcn.compressed as IsNieuweWaardeCompressed,
      vco.compressed as IsOudeWaardeCompressed
FROM
      AuditType at, ChangedDate cd,
      EntityName en, Employee em,
      Property pr, Entity e,
      LogRecord lo
LEFT JOIN RecordKlantPropositie rkp ON lo.recordId =
rkp.record_id
LEFT JOIN KlantPropositie kp ON kp.klantpropositieId =
rkp.klantpropositie_id
LEFT JOIN RecordValueCharacterNew rvcn ON lo.recordId =
rvcn.record_id
LEFT JOIN ValueCharacterNew vcn ON vcn.valueId =
rvcn.value_id
LEFT JOIN RecordValueCharacterOld rvco ON lo.recordId =
rvco.record_id
LEFT JOIN ValueCharacterOld vco ON vco.valueId =
rvco.value_id
LEFT JOIN RecordValueNumericNew rvnn ON lo.recordId =
rvnn.record_id
LEFT JOIN ValueNumericNew vnn ON vnn.valueId = rvnn.value_id
LEFT JOIN RecordValueNumericOld rvno ON lo.recordId =
rvno.record_id
LEFT JOIN ValueNumericOld vno ON vno.valueId = rvno.value_id
WHERE lo.recordId IN
(
      --generated conditions are placed here
)
      AND at.audittypeId = lo.audittype_id
      AND cd.changeddateId = lo.changeddate_id
      AND en.entityNameId = lo.entityname_id
      AND em.employeeId = lo.employee_id
      AND pr.propertyId = lo.property_id
      AND e.entityId = lo.entity_id

ORDER BY lo.recordId
```

Appendix C

In this section, the requirements for the project are presented.

Do's

Req 1. *The system should increase performance for the retrieval of information without compromising the performance for logging information.*

The functionality to retrieve information from the audit log in the applications is disabled currently. The reason is that the performance is so low that the page gets a time out and crashes. The acceptable time to fulfill any request should not take longer than 10 seconds.

Req 1a. *The system should keep the current audit implementation as it is*.

As described before, the current implementation was designed to store information. We keep this component as it is and will use the log its produces as our source of input.

Req 2. *The system should retrieve information from the audit trail logs*

Either application changes and product definitions should be retrievable.

Req 2a. *The system should retrieve a change log of X over time, where X is*

i. a field
ii. an object
iii. an invoice

Important functionality is to be able to retrieve a change log of a field, object or product definition and how it's values evolved over time. In such changelog, information can be found about who changed a certain field, at what time and how many times.

Req 2b. *The system should reproduce a version of an invoice of a given point in time*

Once there is a log about how an invoice evolved over time, there is also the possibility to reconstruct an application. In this context, the word 'application' does not refer to a software application but to a client asking for an invoice. For example, an employee of the mortgage company would like to see how application 1001 looked like (thus with what values) at any point t in time. This application specific information (application numbers etc.) is not present in the current logs and has to be added somehow.

Req 3. *The system should have the functionality to filter and order information during or after retrieval*

The user that requests data should have the functionality to filter the data to get a good overview. This could be specified before the request is made to the system, and filters could be applied once the data is visible on screen. Filtering is used to create a sub-selection with a certain ordering.

Req 4. *The system should clearly display its information and within context for the viewer.*

Once a user gets data on his screen as a result from a request he made, it should be clear to the user what data he is looking at. With large datasets and large result sets data can be overwhelming. Therefore putting the data in context, the user should immediately know what kind of data he is looking at.

Optional:

Req 5. The system could extract scripts of data changes made in order to playback the changes to other databases

There are different databases on different environments which have to be merged from time to time. This is a time consuming activity. The audit logging from different test and setup environments can log all changes made. Once the changes should be merged with the production database, the audit log could be executed on a different database. This cannot cover a complete and clean merge of two databases since values can be changed on several environments. So conflicts still need to be solved. But detecting such conflicts can be done automatically.

Dont's:

These requirements are based on analysis of the data and the possibility to detect anomalies. We cannot fulfill such requirements since our input is a logfile, thus we have a process delay of unknown time depending on the circumstances like the speed of the conversion process, server load and current size of the logfile. Some other requirements were decided to be found out of scope.

Req 6. The system will not support fraud detection

By defining fraud scenario's and rules, analysis on the data could detect violation of those rules or matching scenario's. Detection can be done on a later point in time and does not need to be done real-time. Detection could be additional functionality to the system to be build, but the time needed to research and implement such functionality is time we don't have.

Req 7. The system will not support automatic analysis of data

Analysis on the logged data for the purpose of data mining of to draw statistics from. At this point there is no use for such knowledge and because of privacy violation we are not allowed to the data. Thus the degree of what kind of statistics and what sorts of data mining we are allowed to do have to be investigated.

Req 8. The system will not monitor data and alert users

The audit logging can be used together with self learning algorithms to predict abnormal behavior. For example in the workflow of an mortgage application. Once patterns are found that actions/changes are always in the sequence of 'ABC' then the algorithm that is analyzing the loggings can warn the user or a another person if a workflow of action sequence 'ACB' is detected. We could detect such situations, but cannot give feedback since the system is not looking real-time at the data. Therefore there is no interaction possible with the user.

Environmental requirements

In consultation with Topicus we have defined some requirements and restrictions, regarding the environment, for the solution and prototype:

Req 9. The prototype must be developed in .Net
Req 10. The interface of the prototype must be web-based
Req 11. Authentication is not part of the prototype
Req 12. The current audit trail implementation remains untouched

The current implementation remains because it works as intended and we see no room for improvement there.

Req 13. Preferably a MsSQL database should be used for the data warehouse.

For the data warehouse the request is made to take a MsSQL database. This is because this type of database is commonly used within the company. Of course, with strong arguments there is room to change the type of data storage.

Req 14. *The current audit trail log is our only input for the system*

The information that is present in the current log might not be sufficient to meet all the requirements. Nevertheless we treat the current log as our only source of input. external data could possibly be added as metadata to the data warehouse.

Req 15. *Due to legislations, all data has to be simulated*

All data we will use during the project must be fake or scrambled data. Due to legislations and violation of privacy rules we are not allowed to look into the content of the information that is logged by the audit trail.

Appendix D

The test suite that is used to test the performance of the old audit trail database, the new data warehouse and the architecture. The queries for the old audit trail database are different from the generated ones from the architecture, but they provide the exact same results and are therefore considered represent the same question. In this appendix, the queries and questions for the test suite are provided.

First, the questions, in a natural language form are listed. Second, the questions are translated into queries for the old audit trail database, Third, the questions as asked using the Question language are presented. Fourth and last, the MsSQL queries that are generated by the architecture for the corresponding questions.

The questions in natural language:

Questions	# results
1. Show the employee names and id's from the log records for which the employee is an administrator	27313
2. Show the unique employee names and id's that are administrator	2
3. Show the employee names and id's from the log records from the seventh week of 2011, for which the employee is an administrator	1012
4. Same as question 3, but defined differently in the question language later on	1012
5. Show all the complete log records which have a Entity '*HypotheekDeel*' and Property '*RenteProduct*' or '*VervolgRenteProduct*'	4138
6. Show all the unique properties of the Entity *'HypotheekDeel'*	39
7. Show the complete log records between 100400 and 100900	500
8. Show all the employee names and id's of the employees who '*changed*' the Property 'NominaleRente' from the Entity '*HypotheekDeel*' into the value '*0,03170*'	9
9. Show all the employee names and id's of the employees who '*changed*' the Property 'NominaleRente' from the Entity '*HypotheekDeel*' into the value '*0,03170*' for invoice with number '*16828*'	2
10. Show the complete log records for which the Property 'NominaleRente' from the Entity '*HypotheekDeel*' '*changed*' into the value '*0,03170*' for invoice with number '*16828*'	3

The queries for the old audit trail database corresponding to the questions.

1. `SELECT MedewerkerId, MedewerkerNaam FROM Audittrail WHERE MedewerkerNaam = 'Beheer' OR MedewerkerNaam = 'Beheer1'`
2. `SELECT DISTINCT(MedewerkerId), MedewerkerNaam FROM Audittrail WHERE MedewerkerNaam = 'Beheer' OR MedewerkerNaam = 'Beheer1'`
3. `SELECT MedewerkerId, MedewerkerNaam FROM Audittrail WHERE (MedewerkerNaam = 'Beheer' OR MedewerkerNaam = 'Beheer1') AND Datumwijziging BETWEEN '2011-02-08 16:21:00' AND '2011-02-08 16:32:00'`
4. `SELECT MedewerkerId, MedewerkerNaam FROM Audittrail WHERE (MedewerkerNaam = 'Beheer' OR MedewerkerNaam = 'Beheer1') AND Datumwijziging BETWEEN '2011-02-08 16:21:00' AND '2011-02-08 16:32:00'`

5. `SELECT * FROM Audittrail WHERE Entiteit = 'HypotheekDeel' AND (Property = 'RenteProduct' OR Property = 'VervolgRenteProduct')`
6. `SELECT DISTINCT(Property) FROM Audittrail WHERE Entiteit = 'HypotheekDeel'`
7. `SELECT * FROM Audittrail WHERE Id > 100400 AND Id <= 100900`
8. `SELECT MedewerkerId, MedewerkerNaam FROM Audittrail WHERE Audittype = 'Update' AND Entiteit = 'HypotheekDeel' AND NieuweWaarde = '0,03170' AND Property = 'NominaleRente' AND MedewerkerId != 1`
9. `SELECT MedewerkerId, MedewerkerNaam FROM Audittrail WHERE Audittype = 'Update' AND Entiteit = 'HypotheekDeel' AND Property = 'NominaleRente' AND MedewerkerId != 1 AND NieuweWaarde = '0,03170' AND Klantpropositield = 16828`
10. `SELECT * FROM Audittrail WHERE Audittype = 'Update' AND Entiteit = 'HypotheekDeel' AND Property = 'NominaleRente' AND MedewerkerId != 1 AND KlantpropositieId = 16828`

The questions in the question language corresponding to the questions:

1. show Employee conditions Administrator
2. show Employee conditions Administrator (with the ‘unique’ option activated)
3. show Employee conditions Administrator AND inWeek7
4. show Administrator conditions inWeek7
5. show CompleteLogRecord conditions HypotheekDeelRente
6. show f_Property conditions f_Entity = "HypotheekDeel" (with the ‘unique’ option activated)
7. show CompleteLogRecord conditions CompleteLogRecord > "100400" AND CompleteLogRecord <= "100900"
8. show Employee conditions ChangesToNominalInterest = "0,03170"
9. show Employee conditions ChangesToNominalInterest = "0,03170" AND f_KlantPropositieId = "16828"
10. show CompleteLogRecord conditions ChangesToNominalInterestForKlantPropositieId = "16828"

The queries as generated by the architecture corresponding to the questions in the question language:

1
```
SELECT
  em.employeeId as MedewerkerId,
  em.employeeName as MedewerkerNaam
FROM
  Employee em,
  LogRecord lo
WHERE
  lo.recordId IN (
    SELECT recordId
    FROM LogRecord
    WHERE EXISTS (
      SELECT employeeId
      FROM Employee
        WHERE
          (employeeName = 'Beheer1' OR employeeName = 'Beheer' )
          AND employeeId = employee_id
    )
  )
  AND em.employeeId = lo.employee_id
ORDER BY lo.recordId;
```

2
```
SELECT
  em.employeeId as MedewerkerId,
  em.employeeName as MedewerkerNaam
```

```
FROM
  Employee em,
  LogRecord lo
WHERE
  lo.recordId IN (
    SELECT recordId
    FROM LogRecord
    WHERE EXISTS (
      SELECT employeeId
      FROM Employee
        WHERE
          (employeeName = 'Beheer1' OR employeeName = 'Beheer' )
          AND employeeId = employee_id
    )
  )
  AND em.employeeId = lo.employee_id
ORDER BY lo.recordId;
```

3

```
SELECT
  em.employeeId as MedewerkerId,
  em.employeeName as MedewerkerNaam
FROM
  Employee em,
  LogRecord lo
WHERE (
  lo.recordId IN (
    SELECT recordId
    FROM LogRecord
    WHERE EXISTS (
      SELECT employeeId
      FROM Employee
      WHERE
        (employeeName = 'Beheer1' OR employeeName = 'Beheer' )
        AND employeeId = employee_id)
    )
  AND lo.recordId IN (
    SELECT recordId
      FROM LogRecord
      WHERE
        (changeddate_id BETWEEN '790' AND '800')
  )
)
AND em.employeeId = lo.employee_id
ORDER BY lo.recordId;
```

4

```
SELECT
  em.employeeId as MedewerkerId,
  em.employeeName as MedewerkerNaam
FROM
  Employee em,
  LogRecord lo
WHERE
  lo.recordId IN (
    SELECT recordId
    FROM LogRecord
    WHERE EXISTS (
      SELECT employeeId
      FROM Employee
      WHERE
        (employeeName = 'Beheer1' OR employeeName = 'Beheer' )
        AND employeeId = employee_id)
    )
  AND lo.recordId IN (
    SELECT recordId
      FROM LogRecord
      WHERE
        (changeddate_id BETWEEN '790' AND '800')
  )
AND em.employeeId = lo.employee_id
ORDER BY lo.recordId;
```

5

```
SELECT
  kp.klantpropositieId as KlantPropositie,
  DATEADD(MI,cd.changeddate,'1970-01-01') as DatumWijziging,
  pr.propertyName as Property,
  e.entityId as EntityId,
  en.entityName as Entiteit,
  at.auditType as AuditType,
  em.employeeId as MedewerkerId,
  em.employeeName as MedewerkerNaam,
  vnn.value as NieuweWaardeId,
  vno.value as OudeWaardeId,
  vcn.value as NieuweWaarde,
  vcn.compressed as IsNieuweWaardeCompressed,
  vco.value as OudeWaarde,
  vco.compressed as IsOudeWaardeCompressed
FROM
  ChangedDate cd,
  Property pr,
  Entity e,
  EntityName en,
  AuditType at,
  Employee em,
  LogRecord lo
  LEFT JOIN RecordKlantPropositie rkp ON lo.recordId =
rkp.record_id
  LEFT JOIN KlantPropositie kp ON kp.klantpropositieId =
rkp.klantpropositie_id
  LEFT JOIN RecordValueNumericNew rvnn ON lo.recordId =
rvnn.record_id
  LEFT JOIN ValueNumericNew vnn ON vnn.valueId = rvnn.value_id
  LEFT JOIN RecordValueNumericOld rvno ON lo.recordId =
rvno.record_id
  LEFT JOIN ValueNumericOld vno ON vno.valueId = rvno.value_id
  LEFT JOIN RecordValueCharacterNew rvcn ON lo.recordId =
rvcn.record_id
  LEFT JOIN ValueCharacterNew vcn ON vcn.valueId = rvcn.value_id
  LEFT JOIN RecordValueCharacterOld rvco ON lo.recordId =
rvco.record_id
  LEFT JOIN ValueCharacterOld vco ON vco.valueId = rvco.value_id
  WHERE lo.recordId IN (
    SELECT recordId
    FROM LogRecord
    WHERE
      (property_id = '45' OR property_id = '49' )
  )
  AND lo.recordId IN (
    SELECT recordId
    FROM LogRecord
    WHERE
      (entityname_id = '4' )
  )
  AND cd.changeddateId = lo.changeddate_id
  AND pr.propertyId = lo.property_id
  AND e.entityId = lo.entity_id
  AND en.entityNameId = lo.entityname_id
  AND at.audittypeId = lo.audittype_id
  AND em.employeeId = lo.employee_id
ORDER BY lo.recordId;
```

6

```
SELECT
  pr.propertyName as Property
FROM
  Property pr,
  LogRecord lo
WHERE lo.recordId IN (
  SELECT recordId
  FROM LogRecord
  WHERE EXISTS (
    SELECT entityNameId
    FROM EntityName
    WHERE
      (entityName = 'HypotheekDeel' )
      AND entityNameId = entityName_id
  )
)
AND pr.propertyId = lo.property_id
ORDER BY lo.recordId;
```

7

```
SELECT
  kp.klantpropositieId as KlantPropositie,
  DATEADD(MI,cd.changeddate,'1970-01-01') as DatumWijziging,
  pr.propertyName as Property,
  e.entityId as EntityId,
  en.entityName as Entiteit,
  at.auditType as AuditType,
  em.employeeId as MedewerkerId,
  em.employeeName as MedewerkerNaam,
  vnn.value as NieuweWaardeId,
  vno.value as OudeWaardeId,
  vcn.value as NieuweWaarde,
  vcn.compressed as IsNieuweWaardeCompressed,
  vco.value as OudeWaarde,
  vco.compressed as IsOudeWaardeCompressed
FROM
  ChangedDate cd,
  Property pr,
  Entity e,
  EntityName en,
  AuditType at,
  Employee em,
  LogRecord lo
  LEFT JOIN RecordKlantPropositie rkp ON lo.recordId =
rkp.record_id
  LEFT JOIN KlantPropositie kp ON kp.klantpropositieId =
rkp.klantpropositie_id
  LEFT JOIN RecordValueNumericNew rvnn ON lo.recordId =
rvnn.record_id
  LEFT JOIN ValueNumericNew vnn ON vnn.valueId = rvnn.value_id
  LEFT JOIN RecordValueNumericOld rvno ON lo.recordId =
rvno.record_id
  LEFT JOIN ValueNumericOld vno ON vno.valueId = rvno.value_id
  LEFT JOIN RecordValueCharacterNew rvcn ON lo.recordId =
rvcn.record_id
  LEFT JOIN ValueCharacterNew vcn ON vcn.valueId = rvcn.value_id
  LEFT JOIN RecordValueCharacterOld rvco ON lo.recordId =
rvco.record_id
  LEFT JOIN ValueCharacterOld vco ON vco.valueId = rvco.value_id
  WHERE (
    lo.recordId IN (
      SELECT recordId
      FROM LogRecord
      WHERE
        (recordId > '100400' )
    )
    AND lo.recordId IN (
      SELECT recordId
      FROM LogRecord
      WHERE
        (recordId <= '100900' )
    )
  )
  AND cd.changeddateId = lo.changeddate_id
  AND pr.propertyId = lo.property_id
  AND e.entityId = lo.entity_id
  AND en.entityNameId = lo.entityname_id
  AND at.audittypeId = lo.audittype_id
  AND em.employeeId = lo.employee_id
ORDER BY lo.recordId;
```

8

```
SELECT
  em.employeeId as MedewerkerId,
  em.employeeName as MedewerkerNaam
FROM
  Employee em,
  LogRecord lo
WHERE
  lo.recordId IN (
    SELECT record_id
    FROM RecordValueCharacterNew
    WHERE EXISTS (
      SELECT valueId
      FROM ValueCharacterNew
      WHERE
        (value = '0,03170' )
        AND valueId = value_id
    )
  )
  AND lo.recordId IN (
    SELECT recordId
    FROM LogRecord
    WHERE
      (property_id = '39' )
  )
  AND lo.recordId IN (
    SELECT recordId
    FROM LogRecord
    WHERE
      (entityname_id = '4' )
  )
  AND lo.recordId IN (
    SELECT recordId
    FROM LogRecord
    WHERE
      (audittype_id = '1' )
  )
  AND lo.recordId IN (
    SELECT recordId
    FROM LogRecord
    WHERE
        (employee_id != '1' )
  )
  AND em.employeeId = lo.employee_id
ORDER BY lo.recordId;
```

9

```
SELECT
  em.employeeId as MedewerkerId,
  em.employeeName as MedewerkerNaam
FROM
  Employee em,
  LogRecord lo
WHERE (
  lo.recordId IN (
    SELECT record_id
    FROM RecordValueCharacterNew
    WHERE EXISTS (
      SELECT
        valueId
      FROM
        ValueCharacterNew
      WHERE
        (value = '0,03170' )
        AND valueId = value_id)
    )
    AND lo.recordId IN (
      SELECT recordId
      FROM LogRecord
      WHERE (property_id = '39' )
    )
    AND lo.recordId IN (
      SELECT recordId
      FROM LogRecord
      WHERE
        (entityname_id = '4' )
    )
    AND lo.recordId IN (
      SELECT recordId
      FROM LogRecord
      WHERE
        (audittype_id = '1' )
    )
    AND lo.recordId IN (
      SELECT recordId
      FROM LogRecord
      WHERE
        (employee_id != '1' )
    )
    AND lo.recordId IN (
      SELECT record_id
      FROM RecordKlantPropositie
      WHERE EXISTS (
        SELECT klantPropositieId
        FROM KlantPropositie
        WHERE
          (klantPropositieId = '16828' )
          AND klantPropositieId = klantPropositie_id
      )
    )
  )
  AND em.employeeId = lo.employee_id
ORDER BY lo.recordId;
```

10

```
SELECT
  kp.klantpropositieId as KlantPropositie,
  DATEADD(MI,cd.changeddate,'1970-01-01') as DatumWijziging,
  pr.propertyName as Property,
  e.entityId as EntityId,
  en.entityName as Entiteit,
  at.auditType as AuditType,
  em.employeeId as MedewerkerId,
  em.employeeName as MedewerkerNaam,
  vnn.value as NieuweWaardeId,
  vno.value as OudeWaardeId,
  vcn.value as NieuweWaarde,
  vcn.compressed as IsNieuweWaardeCompressed,
  vco.value as OudeWaarde,
  vco.compressed as IsOudeWaardeCompressed
FROM
  ChangedDate cd,
  Property pr,
  Entity e,
  EntityName en,
  AuditType at,
  Employee em,
  LogRecord lo
  LEFT JOIN RecordKlantPropositie rkp ON lo.recordId =
rkp.record_id
  LEFT JOIN KlantPropositie kp ON kp.klantpropositieId =
rkp.klantpropositie_id
  LEFT JOIN RecordValueNumericNew rvnn ON lo.recordId =
rvnn.record_id
  LEFT JOIN ValueNumericNew vnn ON vnn.valueId = rvnn.value_id
  LEFT JOIN RecordValueNumericOld rvno ON lo.recordId =
rvno.record_id
  LEFT JOIN ValueNumericOld vno ON vno.valueId = rvno.value_id
  LEFT JOIN RecordValueCharacterNew rvcn ON lo.recordId =
rvcn.record_id
  LEFT JOIN ValueCharacterNew vcn ON vcn.valueId = rvcn.value_id
  LEFT JOIN RecordValueCharacterOld rvco ON lo.recordId =
rvco.record_id
  LEFT JOIN ValueCharacterOld vco ON vco.valueId = rvco.value_id
WHERE
  lo.recordId IN (
    SELECT recordId
    FROM LogRecord
    WHERE EXISTS (
      SELECT klantpropositie_id
      FROM RecordKlantPropositie
      WHERE
        (klantpropositie_id = '16828' )
        AND klantpropositie_id = klantpropositieId
    )
  )
  AND lo.recordId IN (
    SELECT recordId
    FROM LogRecord
    WHERE
      (property_id = '39' )
  )
  AND lo.recordId IN (
    SELECT recordId
    FROM LogRecord
    WHERE
      (entityname_id = '4' )
  )
  AND lo.recordId IN (
    SELECT recordId
    FROM LogRecord
    WHERE
      (audittype_id = '1' )
  )
  AND lo.recordId IN (
    SELECT recordId
    FROM LogRecord
    WHERE
      (employee_id != '1' )
  )
  AND cd.changeddateId = lo.changeddate_id
  AND pr.propertyId = lo.property_id
  AND e.entityId = lo.entity_id
  AND en.entityNameId = lo.entityname_id
  AND at.audittypeId = lo.audittype_id
  AND em.employeeId = lo.employee_id
ORDER BY lo.recordId;
```