Lappeenranta University of Technology
School of Business and Management
Degree Program in Computer Science

Niko Liukka

DEVELOPING AUDIT TRAIL FOR ESTABLISHED ERP SYSTEM

Examiners: Professor Jari Porras
Researcher Ossi Taipale

Supervisors: Professor Jari Porras

ABSTRACT

Lappeenranta University of Technology

School of Business and Management

Degree Program in Computer Science

Niko Liukka

DEVELOPING AUDIT TRAIL FOR ESTABLISHED ERP SYSTEM

Master's Thesis

2018

77 pages, 6 figures, 11 tables, 3 appendix

Examiners: Professor Jari Porras
Researcher Ossi Taipale

Keywords: Audit trail, software development, ERP, temporal tables

The term audit trail refers to the records which document the activity that has occured in entity, for example in software application or in business organization. Extensive audit trails are becoming more important when business is conducted more automatically with such tools as ERP systems. The goal of this study is to present concrete method for implementing audit trails in established ERP systems, which have large userbase and wide range of features. This is done by searching literature for audit trail implementation methods and comparing them. After this a case study is conducted. In the case study suitable audit trail implementation method is selected for large scale ERP system with following requirements: reliability, usability and performance. The chosen method is SQL temporal tables, which is shown to be the most reliable and ready made solution. In light of the case study the most important considerations for audit trail functionality development include: requirement gathering, audit trail format, deployment and monitoring as well as architecture of the main system. Additionally the audit trail should be designed to be part of the system early on.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto

School of Business and Management

Tietotekniikan koulutusohjelma

Niko Liukka

AUDIT TRAIL:N KEHITTÄMINEN VAKIINTUNEESEEN ERP-JÄRJESTELMÄÄN

Diplomityö

2018

77 sivua, 6 kuvaa, 11 taulukkoa, 3 liitettä

Työn tarkastajat: Professori Jari Porras
Tutkija Ossi Taipale

Hakusanat: Audit trail, ohjelmistokehitys, ERP, temporal tables
Keywords: Audit trail, software development, ERP, temporal tables

Termillä audit trail tarkoitetaan dokumentteja jotka kuvaavat mitä jonkin kokonaisuuden, kuten ohjelmiston tai yrityksen, sisällä on tapahtunut. Liiketoiminnan automatisoituminen esimerkiksi ERP-järjestelmien käytön myötä on tehnyt luotettavasta audit trail:sta entistä tärkeämmän. Tämän työn tavoitteena on esittää konkreettinen menetelmä, jota voidaan käyttää audit trail:n toteuttamiseen vakiintuneessa ERP-järjestelmässä, jolla on laaja käyttäjäkunta ja paljon ominaisuuksia. Aluksi esitellään kirjallisuuskatsaus, jossa etsitään ja verrataan erilaisia audit trail:n toteutustapoja. Tämän jälkeen suoritetaan tapaustutkimus, jossa suurelle ERP-järjestelmälle valitaan audit trail:n toteutustapa seuraavien vaatimusten pohjalta: luotettavuus, käytettävyys ja suorituskyky. Toteutustavaksi on valittu SQL temporal tables, jonka osoitetaan olevan luotettavin ja käyttökelpoisin ratkaisu. Tapaustutkimuksen valossa tärkeimpiä huomioitavia asioita audit trail:n kehittämisessä ovat vaatimusten määrittely, audit trail:n rakenne, käyttöönotto ja valvonta sekä varsinaisen järjestelmän arkkitehtuuri. Lisäksi audit trail tulisi suunnitella järjestelmän osaksi jo järjestelmän elinkaaren alkuvaiheessa.

ACKNOWLEDGEMENTS

This study has been made as a master’s thesis in Degree Program in Computer Science at Lappeenranta University of Technology. I would like to thank my supervisors from both the university and from the case company for whom this work was done to for their support and guidance. Furthermore thanks to everyone in the case company who shared their knowledge and ideas during the development process. Last but not least I would like to thank my fiancée for her support during this work and through out my studies.

Niko Liukka
Lappeenranta 15.1.2018

LIST OF SYMBOLS AND ABBREVIATIONS

API - Application Programming Interface

CA - Continuous Auditing

CDC - Change Data Capture

CMM - Continuous Management Monitoring

DDL - Data Definition Language

DML - Data Manipulation Language

ERP - Enterprise Resource Planning

ID - Identification

IT - Information technology

KPI - Key Performance Indicator

1 INTRODUCTION

“Audit trail” has two somewhat similar definitions which are both relevant in the context of Enterprise Resource Planning (ERP) systems. In accounting terms it refers to “documents and records that show the history of a company's financial activities, examined by someone who is doing an audit” and in information technology (IT) it refers to “a record of the activity on a computer or computer system”[1]. In other words the audit trail means keeping a record of the data which is or has been in a system or in a process as well as recording all the changes which have occurred in the data. So with complete audit trail it is possible to take a piece of data and follow the “trail” of changes to find out how the data has looked in the past. In addition to recording the actual changes it is also principal to record when and by whom the changes were made. This makes it possible to define who is accountable for the data. In practice audit trail is important because it makes it possible to verify that everything within an entity, be it a bookkeeping of the company or a software application, has gone as intended and that there has been no fraudulent actions.

In accounting the audit trail can mean for example being able to follow where the record in the ledger has originated. Furthermore even if the record seems to be valid, it is still beneficial to be able to inspect the whole trail of changes. In the literature there is classical example where fraudulent employee changes the bank account number from the information record of the vendor in the ERP system, pays invoice to this incorrect bank account and lastly changes the bank account back to original. (Singh, Best, Bojilov, & Blunt, 2014) These kinds of actions can go unnoticed if only the trail of records from invoice to the ledger is followed. For this reason it is important to have the changes recorded for every important piece of data, in this example to the vendor bank account information. The research has shown that keeping this kind of record has been made much more efficient by the use of software such as ERP systems. On the other hand the efficiency of software has also created more possibilities for frauds. (Debreceny, Gray, Ng,

1 Cambridge Dictionary. (2017). Retrieved 18 October 2017, from https://dictionary.cambridge.org/dictionary/english/audit-trail

Lee, & Yau, 2005) The use of ERP systems for bookkeeping purposes leads to the situation where the software audit trail of the ERP system forms the backbone for the accounting audit trail.

1.1 Goals and delimitations

The goal of this work was to present the necessary steps for developing audit trail functionality for the large scale ERP system. In order to reach this goal it was necessary to be able to answer to the following research question:

- How to implement audit trail in the large ERP system which has grown naturally over the time and what factors need to be considered before, during and after the implementation in this scenario?

These questions were further broken down to several parts which can be examined with defined research methodology. Firstly it was necessary to define the term audit trail (Q1) and the purpose it serves in the context of the ERP systems. The latter question was approached with case study. The case study consisted of the following steps: gathering the requirements for the audit trail from the various departments within the case organization (Q2), analyzing current audit trail of the case ERP system and its shortcomings (Q3), finding and comparing alternative methods for audit trail (Q4) and selecting the most suitable method (Q5). Based on this method a proof of concept audit trail was implemented and its suitability for system wide use was evaluated (Q6). The proof of concept included both the system for storing the audit trail information as well as functionality for history data maintenance.

The suitability of the proposed proof of concept was evaluated based on its ability to be scaled up to include all the information which needs to be audited in the system. This ability means fulfilling the requirements of reliability, usability and efficiency in processing time and storage space in a way that it can be deployed to system wide use. The requirements are described in more detail in the chapter 3.3. The proof of concept had to be

implemented in the manner which fulfills software quality requirements of the organization and is reviewed to be suitable for system wide use by other developers and system architects.

The system wide deployment of the developed audit trail solution was outside of the scope of this work because the significant size of the ERP system. Because of the size the system wide deployment would require excessive time and resources, especially in testing, and these were not available within the limited time frame of this work. Rather this work proposes comprehensive proof of concept for solving the problem which can then be later deployed system wide.

1.2 Research Methodology

Three distinct research methods were used in this work. The main methods were literature review and case study, which were complemented by small scale survey. Literature review was used as research method for providing background knowledge about the subject of the research. The review was conducted in two different parts. The first part was about gathering background knowledge and defining the most vital concepts for the research, namely audit trail and continuous auditing thus providing answers to the sub research question Q1. This clarified the goal of the work and gave initial guideline for choosing the final implementation method for the audit trail. The second part of the review was conducted by following the principles of systematic literature review (Kitchenham et al., 2009). Systematic literature review was performed in order to find and compare possible solutions for solving the problem of adding audit trail functionality into existing software system. This answers the sub question Q4.

After researching the literature a case study about audit trail development was performed. The case study was performed in Finnish software company which develops and maintains large scale and widely used ERP system, which is referenced in this work as “case ERP system” or “ERP system”. The case study consisted of development of the proof of concept audit trail solution. The goal was to use the gathered knowledge and utilize it to create auditing functionality to existing ERP system. The case study included the requirement

gathering, implementation and end result evaluation and thus it answers the sub questions Q2, Q3, Q5 and Q6. The outcome of the case study was proof of concept which could be further deployed to the system wide use. By evaluating the development process and the end result it is possible to define the advantages and disadvantages of the chosen implementation method as well as the factors which need to be considered at the different stages of the project. The identified considerations are general and technology-independent meaning that they can be applied in audit trail implementation processes regardless of the case specific details.

Lastly a survey was performed by sending limited questionnaire to other companies with similar ERP products in different markets. The goal was to get insights on how other similar ERP systems have dealt with the problem of auditing and thus get more input to be used on decision making when selecting the final implementation method. This gave further insights on the sub questions Q4 and Q5. The questionnaire was sent to three organizations and two of them gave their explicit permission to reference their responses in this thesis. The used research methodology is summarized in table 1.

1.3 Structure of the thesis

The rest of this report is structured as follows. Chapter 2 features literature review which gives background information about relevant terms and concepts as well as introduces audit trail implementation methods found in the literature. Chapter 3 describes the case study setting by introducing the case company and ERP system. Audit trail implementation methods which were considered in that setting and requirements for the new audit trail are discussed as well. The proposed solution for implementing the audit trail in the case study is presented in the chapter 4 and suitability of this solution is discussed in the chapter 5. Finally the chapter 6 summarizes the findings and gives answers to the research questions as well as discusses about the limitations of the results and future research directions.

Table 1. List of used research methods and their purposes

Research method	Purpose of the method
Literature review about audit trail	Define audit trail and related concepts.
Literature review about audit trail implementation methods	Find alternative methods for audit trail implementation.
Survey about audit trail implementation methods	Find alternative methods for audit trail implementation.
Case study • Requirement gathering • Method comparison • Implementation • Measurement • Evaluation of the process	• Gather audit trail requirements from real life case • Compare the previously found methods against found requirements • Create proof of concept audit trail • Verify the suitability of proof of concept against the initial requirements • Draw conclusions about the implementation process which can be generalized to other implementation process regardless of the used technologies

2 LITERATURE REVIEW

The literature review was used to gather knowledge and understanding which would then be used as a basis for the rest of the work. The review presented here will give a definition for the concept of the audit trail and list several different approaches for implementing it in software systems. The review was performed in two stages. In the first stage articles were searched from several sources with keyword combinations “audit trail” AND “development”, “audit trail” AND “ERP” as well as “continuous auditing” and by finding more related articles from the references of the found articles. Selected articles had to be peer reviewed and published in scientific publications. With this approach it was possible to efficiently gather knowledge about the concept of audit trail and its related concepts.

Secondly a systematic literature review was performed to find concrete implementation alternatives for the audit trail functionality. The systematic literature review method was chosen because of its ability to systematically cover broad field of research which could lead into insightful list of possible solutions including their strengths and weaknesses (Kitchenham et al., 2009). This was important for deciding if there was an existing solution or partial solutions for the problem or if the problem needed to be solved from the scratch. The main difference between systematic literature review and normal review is the use of review protocol and strategy. The protocol defines how the reviewed literature is found and strategy describes how the found literature is analyzed. The protocol and strategy, as well as the review process in which they are used, are documented to make the review more transparent and objective. (Kitchenham et al., 2009) This documentation can actually be seen as an audit trail of the review process. In the review literature was searched from databases listed in the appendix 1 with searchterms: “audit trail” and “audit trails”. The term had to appear in the abstract of the article. The goal was to identify detailed technical methods and technologies used for implementing auditing functionality. The searchterms were broad because preliminary queries revealed that material could not be found with more accurate search terms, for example with “audit trail development”. Several additional parameters were used to narrow down the number of the results. Firstly the publication year had to be 2010 or later. The accepted time period was relatively short but it was

decided based on the fact that technologies change quickly. Especially the more detailed methods are likely to become outdated in few years. Furthermore the selected articles had to be peer reviewed and available free of charge for the student of Lappeenranta University of Technology. Also the articles had to be written in English.

2.1 Audit trail

Audit trail has been well known concept for several decades especially in the field of accounting. In its essence the term audit trail boils down to the logs which describe the workings of the system. With these logs it becomes possible to audit the system, in other words to verify that everything has gone as intended. Furthermore it is possible to reconstruct the state of the system at the certain point in the past based on the audit trail. In order to be effective the audit trail logs must have well defined format which is machine readable and not tied to specific platform (Bishop, 1995). In accounting the most important actions for auditing are the different approvals (e.g. approving an invoice), postings (e.g. sending a payment) and deletions of all kinds of data (Li, Huang, & Lin, 2007). These actions are present in most transactions, for example processing of the invoice. For transactions the audit trail should include information about the existence (e.g. the approval for invoice exists), wholeness (e.g. invoice contains all the necessary information and each step of the processing is saved), accuracy (e.g. correct sums in invoice and other related records), classification, time and inclusion in the balance sheet (Li et al., 2007). On the other hand in terms of database systems the audit trail should contain: list of events which altered data; timestamp of the event; reason for the alteration; user who altered data; users role and location. In order to effectively gather this information the audit trail should be designed to be part of the database structure from the get go. (Flores & Jhumka, 2017) Adding this functionality afterwards could be expensive task in terms of time and effort.

The existence of the audit trail is a security question and audit trails are most commonly used in such fields as accounting and IT (Bishop, 1995). They are also commonly used for example in research and medicine when the integrity of the data is important for the security reasons (for example Cruz-Correia et al., 2013). This highlights the main purpose of the audit trail which is to verify the integrity of the data. The integrity is achieved when

it is possible to tell who has created, edited or deleted the data and thus is accountable for it. With audit trail the persons who have done incorrect actions with data, be it by accident or by fraudulent operations, can be held liable for their actions. The fraud detection has become increasingly more important because of the digitalization and automation of business and other activities. For example there are less and less physical logs about business transactions when everything is stored digitally. Furthermore the research suggests that automation increases the volumes of transactions which makes it harder to detect the few fraudulent operations within the whole mass. In order to prohibit the increased potential for frauds the digital systems must have proper audit trail functionality in place. Typical organization loses on average approximately 5% of its annual revenue to frauds. In 2011 this would have translated to the loss of more than $3.5 trillion worldwide. (Singh et al., 2014) In comparison Gross Domestic Product of Finland was $0.27 trillion during the same year [2]. The monetary losses are especially significant because in most cases the victims will not get their assets returned and thus face the possibility of bankruptcy. For this reason it is important to be able to prevent and detected frauds as efficiently as possible (Singh et al., 2014). Either way the frauds are always problematic for the image of the organization which can generate even more monetary losses.

The study of audit trails suggests that frauds are abnormalities in the normal operation of the organization and thus they can be detected by searching for anomalies in the audit trail. This process can largely be automated which makes the fraud detection greatly effective. The suspicious anomalies are numerous and they change from domain to domain, but in accounting some common anomalies include for example even sums in payments, duplicate payments, duplicate customers, customers with multiple bank accounts and repeatedly changing bank account numbers especially for vendors. (Singh et al., 2014) Classical fraud example is the case where employee changes the bank account number of vendor from the information system to bank account where he has access to, then pays

2 TradingEconomics. (2017). Finland GDP 1960-2017. Retrieved 2 November 2017, from tradingeconomics.com/finland/gdp

invoice for the vendor (to the wrong account) and then changes the bank account number back to original. If the vendor information changes are recorded then this kind of activity can easily be detected.

The research has shown that often the digitalization of the business processes involves the utilization of the ERP systems which integrates and unifies the core functionalities of the business. This makes the ERP system the central point of the operations and thus makes it also natural place for audit trail implementation, since most if not all transactions go through the system at some point. (Debreceny et al., 2005; Singh et al., 2014) However the ERP systems have implemented audit trails with varying interest. According to the study about embedded audit modules in the ERP systems, at the start of the millennium the reason for not implementing audit trail functionality in ERP systems was the lack of interest from the customers who felt that audit modules were not worth the effort because they could only detect frauds that had already happened. (Debreceny et al., 2005) Since then the attitudes are likely to have changed, however the lack of found audit trail solutions in the systematic literature review as well as the answers to the questionnaire which was sent to other ERP developers (chapters 2.3 and 3.2) suggests that even today there is no critical need for complete audit trail solutions among ERP customers.

2.2 Continuous Auditing

The concept of continuous auditing (CA) is closely related to the audit trail. CA is defined as an comprehensive electronic audit process which provides the auditor with relative certainty about the integrity of the accounting information in real time (Rezaee, Sharbatoghlie, Elam, & McMickle, 2002). CA can utilize the audit trail by analyzing it in real time or by making decisions based directly on the events in the system. This is cheaper, more efficient and more timely than traditional auditing (Alles, Kogan, & Vasarhelyi, 2008; Shin, Lee, & Park, 2013). At present the efficiency of CA is increasingly more important because the digitalization and automation of business processes increases the volume of transactions and lessens the human intervention. For this reason the auditing too has to be automated to keep up with the increasing numbers of transactions. In the

changing environment the CA can be used to provide outwards transparency about the business, direct attention internally and for legal reasons (Shin et al., 2013).

When CA is used to direct attention internally it is a part of internal control framework. Internal control is defined as a process managed by the board of the organization. The goal is to provide reasonable assurance about achievements of objectives and about effectiveness of the operations. (Chang, Yen, Chang, & Jan, 2014) According to other definition the CA is also a part of the broader Continuous Management Monitoring (CMM). CMM is about providing real time statistics and key performance indicators (KPI) for the company management to make data driven decision about the business. (Alles et al., 2008) From the technological perspective the collection and utilization of the data is similar between CA and CMM (Rezaee et al., 2002). Since CMM is implemented to increase the profits of the company the CA process can often be implemented alongside with it with the same effort. This makes the adaption of the CA more cost efficient but on the other hand it can also hinder the independence of the auditors. (Alles et al., 2008)

The rise of the CA has meant that auditor role has changed from reactively discovering frauds to more proactive detection and even prevention (Shin et al., 2013). This should make the implementation of the audit modules more desirable even to users who feel that reactive audit trail analysis is not effective enough. Great example about proactive monitoring is detecting violation of the segregation of duties. By definition the segregation of duties means that certain operations in organizations should be divided to different persons. For example same person should not be able to create or modify customer information and create or modify invoices for that customer. (Shin et al., 2013) Detection of these kinds of violations can be automated and when violations are found it is possible to direct more attention to the persons who have these kinds of privileges within the organization before they manage to exploit the violations.

2.3 Audit Trail Implementation Methods

Implementation methods were searched with the systematic literature review with the research protocol and strategy described at the beginning of this chapter. The initial search resulted in the list of 175 articles. However because of the broad searchterms, most of the articles dealt with subjects other than technical development of the audit trail. For this reason the results were further narrowed down by reviewing the titles and abstracts of the articles. The goal was to find articles which describe auditing methods from technological perspective for example by describing algorithms or process flows of auditing functionality or by discussing technologies which are used for providing auditing functionality. If the title and abstract of the article did not mention any of these the article was not fully reviewed. For example many of the articles were concerned with achieving and using the audit trail from organizational point of view in research and medical institutes. The final review material included 12 articles which were fully reviewed for the possible solutions for audit trail development. The reviewed articles are listed in the table 2 and found solutions are listed in the table 3.

Table 2. The list of fully reviewed articles.

1. Management of a Large Qualitative Data Set: Establishing Trustworthiness of the Data (White, Oelke, & Friesen, 2012)
2. Improved Security of Audit Trail Logs in Multi-Tenant Cloud Using ABE Schemes (Prakash & Nalini, 2014)
3. Forensic accounting in the fraud auditing case (Simeunovic, Grubor, & Ristic, 2016)
4. A Risk-Based Approach to Data Integrity (Albon, Davis, & Brooks, 2015)
5. Security and Audit Trail Capabilities of a Facilitated Interface Used to Populate a Database System with Text and Graphical Data Using Widely Available Software (Beland et al., 2014)
6. Using XBRL Global Ledger to Enhance the Audit Trail and Internal Control
7. Security information in production and operations: a study on audit trails in database systems (Bizarro & Garcia, 2011)
8. Analysis of the quality of hospital information systems audit trails (Cruz-Correia et al., 2013)
9. 3 Steps to Simplify Audits, Demonstrate Compliance and Manage Risk Across the Enterprise (Anonymous, 2011)
10. Compliance and Data Access Tracking (Mullins, 2011)
11. Automating Vendor Fraud Detection in Enterprise Systems (Singh, Best, & Mula, 2013)
12. A review and future research directions of secure and trustworthy mobile agent-based e-marketplace systems (Patel, Qi, & Wills, 2010)

Table 3. List of audit trail implementation methods.

Title of the article	Audit trail solutions	Category
A Risk-Based Approach to Data Integrity (Albon et al., 2015)	System logs	System logs
Using XBRL Global Ledger to Enhance the Audit Trail and Internal Control (Bizarro & Garcia, 2011)	eXtensible Business Reporting Language (XBRL)	Other
Security information in production and operations: a study on audit trails in database systems (Roratto & Dias, 2014)	Database triggers Database logs	Database Database
Compliance and Data Access Tracking (Mullins, 2011)	Database auditing software Capturing database requests Database logs	Database/audit module Database Database
Automating Vendor Fraud Detection in Enterprise Systems (Singh et al., 2013)	Embedded Audit Modules Monitoring and Control Layer Operating system (logs) Database (logs)	Audit module Implementation in application System logs Database
A review and future research directions of secure and trustworthy mobile agent-based e-marketplace systems (Patel et al., 2010)	Recording and analyzing the network traffic	Network traffic
Improved Security of Audit Trail Logs in Multi-Tenant Cloud Using ABE Schemes (Prakash & Nalini, 2014)	reverse proxy logs	Network traffic
Security and Audit Trail Capabilities of a Facilitated Interface Used to Populate a Database System with Text and Graphical Data Using Widely Available Software (Beland et al., 2014)	Domain layer	Implementation in application

The solutions are of varying levels of abstraction. Some of them are very detailed, for example database triggers, while others are broader concepts, for example recording and analyzing the network traffic. Because of this variation and low number of items it is impossible to draw concrete conclusions about popularity or suitability of the solutions from the data. However the found items can easily be categorized into few main groups: Database (5.5 items), Network traffic (2), System Logs (2), Implementation in application (2), Audit module (1.5) and Other (1) (note “Database auditing software” is categorized into both Databases and Audit module, hence the 0.5 items). The higher number of items in

the database category seems relevant for the audit trail development because most of the modern applications store the data in databases and that data is the subject of the audit trail. This makes the database seem like natural place for the audit trail functionality as well. On the other hand the network traffic, system logs, implementation in application and audit module categories focus on capturing events in the system and deriving the audit trail from them.

In the context of the case study all of the found solutions were viable in theory. But once the available time and resources were considered many of them could be discarded. Firstly the use of XBRL and embedded audit modules could be discarded because XBRL is not currently supported in the system and its implementation would require great effort which would not make sense from the business perspective because there is no demand for the XBRL. Likewise the adaption of third party audit module could be discarded because developing the functionality in-house was seen economically more desirable.

Furthermore the system logs and analysis of network traffic could be discarded because they were not accurate enough for solving the problem at hand. The goal was to audit the business data which is processed within the system and both (operating) system logs and recording of the network traffic would produce significant amount of data which is irrelevant for this purpose. The data could be filtered but the remaining solutions, namely database and implementation in the application would be adjusted by default to only process the necessary data.

3 CASE ERP SYSTEM

The case ERP system is developed by medium-sized Finnish software company. The system is cloud based meaning that customers use it over the Internet via browser rather than running it on premises. It is relatively large in terms of its functionality and user base which comprises of thousands of direct and accounting office customers who generate hundreds of thousands transactions monthly. The system also has hundreds of integrations to other software systems. The development of the system is continuous and has been ongoing for several years. During this time few different software technologies and frameworks have been used and multiple frameworks are still in use. This fact has to be kept in mind when implementing system wide functionality such as audit trail so that all different frameworks and system parts are considered.

There are various types of data processed in the system and the audit trail is most important for the information associated with financial transactions, for example invoices, vendor and supplier information, salary data, bank transfers etc. The daily number of these kinds of transactions is in tens of thousands. The second important set of information for auditing is the user information, for example personal information of the employees and users as well as user access information. The retention times for audited data can range from months to over ten years. The update frequency of the data is impossible to predict accurately, however generally the audited data changes require action from user and thus the data is more slowly changing than data which is altered by the system itself. For example user can in practice do only limited number of actions in certain time whereas the system could perform thousands of similar actions in the same time if they are automated. If the actions are performed by user they should be audited because they can be attributed to the user. On the other hand if the actions are automated the individual actions will not necessarily require auditing but rather the action of activating the automation should be audited because it can be attributed to the user.

3.1 Current state of the audit trail in the ERP

Currently in the system the audit trail exists for the most important sets of data for example invoices and bank account information. The backbone of the current solution is formed by database triggers which are created for necessary tables. These triggers capture and store all the data manipulation language (DML) events alongside with the old and new values of the changed data entries. In addition to this audit solution there are also other supporting solutions. For example some tables which are not supposed to be updated have more lightweight auditing where only the deleted entries are recorded. In addition to auditing the data changes there are also extensive logs for recording the requests made through both the user interface and application programming interface (API). This log can then be used for tracing who has done and what in the system.

Currently all of the important data changes are recorded and most of the other changes can be traced through the request log. However the current solution still has quite a few significant drawbacks. First of all the performance of the trigger based solution is not seen as good enough for the system wide use. There have been cases where the current audit solution could not be used because large scale operations would have taken too long to execute with audit logging and this would have had negative impact on usability and user experience. The main reason for the poor performance is the way the change history is stored which is not scalable. Meaning that when the amount of the history data increases the performance of the functionality decreases comparatively. This is a result of first introducing the audit log only as an ad hoc solution for limited part of the system from where it has then grown over the years without re-evaluation of the design. Considering this the new audit solution should have more thoroughly considered architecture and better scalability.

In addition to the less than desirable performance, the trigger based solution also has some issues with usability, maintainability and reliability. Currently there is no user interface for inspecting the audit data within the ERP system, except in few rare cases. This means that the inspection of audit data requires effort from the technical support team because the customers are not able to view the data themselves. In addition to the lack of the interface

the querying of history data could take too long to be widely usable by customers. Furthermore the lack of user interface also means that history data interpretation requires additional effort from the technical support. When changes have to be tracked from the data which is not part of the current audit solution the amount of investigative work increases further and for that reason a wider audit solution is desirable. The amount of required effort is multiplied when we consider that often the customers will not contact the technical support directly but rather they deal with customer support or for example invoicing department. The easy to use interface for inspecting the history data would decrease the amount of work required for audit data investigation through the company. With proper interface the users could even be allowed to view the history data themselves which could decrease the number of support cases where the main question is who has done and what within the customer's environment.

The problems with maintainability arise from the fact that currently every audited table requires specially created trigger which creates maintenance overhead. Also over the years triggers have been created with varying designs which makes the maintenance work more challenging. The last issue is about reliability. The current solution handles the updater information like any other information meaning that in database level the updates are allowed without explicitly defining the updater. This further means that the updater must be defined at the application level everywhere where updates are made which creates possibility for errors. Since there is at least a theoretical possibility of updates occurring without updater information the functionality could be made more robust.

3.2 Audit trail survey in similar ERP systems

As a part of the case study a questionnaire about audit trail was sent to three other organizations with similar ERP products. All of them responded but only two gave their explicit permission to reference their responses in this thesis. The goal of the questionnaire was to find out if other similar ERP systems had implemented the audit trail and if so what kind of design are they using. The questions and summarized answers of the questionnaire are listed in the table 4.

Table 4. Questions and summarized answers of the audit trail questionnaire.

Question	Summarized answers
Have you implemented audit trail/data versioning within your system? Are you keeping record on all the data changes that happen within the system or just on some changes?	1. Yes, ad-hoc logging for specific changes. 2. Yes, with possibility to select the stored data by end customer and administrator.
Describe the current solution for versioning: - Is it implemented in database (e.g. triggers, system versioned tables...) or in application (e.g. in data access layer, within the object-relational mappings...)? - Describe the general architecture if possible. - Are you able to record all the chosen changes reliably?	1. The change, timestamp and user information are saved in the data layer of the application. 2. The following information is stored: operation (*update*, *insert*, *delete*), table name, value and modified fields. The auditing can be activated for individual columns. The logging is done in application level.
Who is using the collected history data? Customers/support/development etc.	1. Any one who has access to the user who has made the changes. 2. Customers
What are the biggest advantages and disadvantages of your solution?	1. - 2. Simple and flexible solution which achieves its purpose.

Both of the participating organizations had implemented the audit trail to some degree. In other only the most sensitive pieces of data were versioned with changed values, user information and timestamps and in other the user could choose which data was included in the audit trail, but only the fact that data was changed alongside with the timestamp and user information were recorded. In this implementation the flexibility of choosing which parts of the system are audited was seen as a great advantage. Both of the organizations had implemented the auditing logic in the application level which writes the auditing log to the database. In both cases the auditing data was mainly used by the customers.

3.3 Requirements

The discussions with stake holders in the case company and the literature review suggest that the most important requirement for the audit trail is the ability to reliably proof who has changed data in the system. In terms of transactions the audit trail should be able to proof the existence, wholeness and accuracy of the data (Li et al., 2007). In other words the audit trail should be able to prove the integrity of the data which is a fundamental security question (Bishop, 1995). In order to be able to account the actions performed within the system to someone there has to be reliable way of recording the actions as they are performed. For example by recording all the changes made to the data in the system. In addition to the actual change the id of the person who changed the data as well as timestamp of the event should also be recorded (Flores & Jhumka, 2017). These requirements are also present with the case ERP system. First of all from legal point of view the ERP system contains sensitive information which has to be reliably accounted to someone: who has created this information and thus is responsible for it. Secondly some of the customers have stricter requirements for the internal control of their organization and for this they require functioning audit trail. Occasionally other customers require information about data changes as well and this generates work for the support department of the application which can be decreased with easy to use audit trail information. Audit trail can also support the bug detection and verification because possible bugs can be verified to have happened because of the malfunction of the software and not because incorrect actions of the user.

The other possible questions of “why” and “where”, in other words reason for the change and physical location of the changer, were excluded from the requirements (Flores & Jhumka, 2017). They were not seen as important as the “what”, “when” and “who”, because gathering the reasons for changes would require additional work through the system and in most cases the reason would be of little value. Same restrictions are associated with the location of the changer.

Aforementioned basic requirement of traceability and the shortcomings of the existing solution offer the basis for the more detailed requirements. Most importantly the audit trail should be reliable, meaning that when it is activated every change should be recorded and there should be no caps within the trail nor records with inadequate metadata, for example the id of the user and timestamp. Also any alteration attempts towards the recorded history data should be prevented so that the integrity of the records is ensured. For effective implementation and maintenance of the system wide audit trail the ease of development and deployment need to be addressed as well. The proposed solution should be easy to understand for the developers and it should require only minor work per each part of the system that it is applied to. The possibility to automate the process is desirable. This would also support the maintainability of the solution since if it was to be altered the changes could be automatically applied system wide. In addition to the developers the ease of use should apply also to the end users of the audit trail, for example to customer support and customers themselves. In practice they should be able to get the information they need with minimal effort. Last major requirement is the cost of processing time and storage capacity. For the most part the audit trail functionality is very basic in ERP systems meaning that there is only limited sales potential related to it. In some cases it can remove the obstacles for sales, for example when customer requires enhanced audit trail capabilities for internal control but in most cases it is mainly basic feature which customers expect to be in place at least on some level. For this reason the improved audit trail is unlikely to bring any direct profits for the company meaning that recurring costs associated with it should be minimized. These are mainly the costs generated from processing power and storage. The new audit trail functionality should be able to run with current processing power without noticeable difference in system performance for the end user. Furthermore the required storage capacity for the history data should be as low as possible. This can be achieved with proper data compression. The requirements are summarized in the table 5.

Table 5. Summary of the requirements.

Requirement	Description
Reliability	Every change must be recorded with appropriate metadata and the recorded history must be immutable. Changing the data without audit record must not be possible.
Usability	The development, maintenance and use of the functionality should require as little work as possible. The developers should be able to add the auditing functionality to new data entity with one hour of work. The audit log should be usable for users without special technical knowledge.
Performance	The recording of the history data should require as little storage space as possible. The increase in storage space usage should be at most ten folds for each audited entity. The recording process should not have impact on the performance of the system which is noticeable to the user.

3.4 Comparison of the different methods

The literature review resulted in surprisingly low number of actual technical audit trail implementation examples. It seems that most of the research has focused on the implications which the audit trail has on continuous auditing and internal control of the organizations (for example Cruz-Correia et al., 2013). Database triggers were the only specific method which was found in the review. It is also the method which the current audit trail in the ERP system is based on. Since the literature did not provide any strong and concrete alternatives for the triggers, which were considered to be not optimal in the case of the ERP system, they had to be found from elsewhere.

First alternative came from the third party company who are consulted by the case company on more demanding database maintenance tasks. The ERP system was currently in the process of migrating to a 2016 version of SQL Server. This edition of the SQL Server has many additional features compared to the old version, one of the most interesting being the introduction of system versioned temporal tables which provide built-

in support for storing and querying data that is or has been stored in the table at any given time. [3] This behavior is ideal for storing the audit trail. Feature was introduced in ANSI SQL 2011 standard but was not included in the SQL Server until the 2016 version (Kulkarni & Michels, 2012). The two remaining methods, namely SQL Server Change Data Capture and implementation in the application layer were formed by conversations within the development team and by interviewing product managers and architects from other ERP product developers. The general advantages and disadvantages of the alternatives are discussed next and summary about these characteristics is presented in the table 6.

3.4.1 SQL Server Change Data Capture

Change Data Capture (CDC) is feature which enables tracking changes to the data in a database. It logs the DML actions of the database along with the actual data and stores them within the database making them relationally searchable. [4] The main purpose of the CDC is to get the data that has changed. Main use case is the data replication to multiple instances, for example into data warehouse. The warehouse contains copy of the actual data which is constantly changing and is stored to different location closer to the application. With CDC it becomes possible to sync the warehoused data with the actual data for example once a day. The main advantage is that if the actual data is changing rapidly, instead of directly mirroring every change from the actual data to the warehoused data only the current state of the changed data can be synced to the warehouse once a day because with CDC it is possible to identify which parts of the data have changed. This makes the replication more efficient because unnecessary syncs can be eliminated. Similar functionality could be achieved by utilizing database triggers, timestamp columns and additional metadata tables, but CDC automates the change capture making the development process faster and less error prone. With CDC there is no need for schema changes and it also contains automated cleanup mechanism for deleting the recorded data

3 Rabeler, C., Hamilton, B., Sauber, S., Milener, G., & Guyer, C. (2016). Temporal Tables. Retrieved 19 November 2017, from https://docs.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables

4 Rick, B., Bruce, H., & Craig, G. (2016). Track Data Changes (SQL Server). Retrieved 28 December 2017, from https://docs.microsoft.com/en-us/sql/relational-databases/track-changes/track-data-changes-sql-server

after certain time period. Furthermore the CDC is very efficient because it is not directly tied to the database events, rather it gathers the changes from the database logs which are automatically generated by the DML events. [5]

The CDC is not actually meant for providing data auditing functionalities, but it was included as a possible audit trail solution because of its ability to automatically and efficiently capture the changes of data. This is the starting point of the audit trail and while the CDC is not necessary designed for providing such a feature, it could in theory be used as a basis for audit trail. However the question of who has changed the data would have to be addressed some other way. Furthermore in audit trails the actual trail of changes is of utmost importance and in CDC the fact that something has changed along with the start and end states is more important than the full trail of changes. This difference is problematic from the auditing point of view.

3.4.2 Database Triggers

Database triggers are perhaps the most used method for implementing database audit trail which is supported by the fact that it was the only concrete method found in the systematic literature review (Roratto & Dias, 2014). In essence the triggers are stored procedures, in other words small pieces of code, which run automatically after certain database events. There are two main types of triggers Data Manipulation Language triggers (DML) and Data Definition Language triggers (DDL). DML triggers fire after data manipulation events: *insert*, *update* and *delete* and they can be used for example for enforcing business rules and data integrity at the database level or for writing logs about the changes. DML triggers have two types: *after* triggers which fire after the actual event and *instead of* triggers which fire instead of the actual event. DML triggers are executed in a single transaction with the event that fired it. Meaning that if the execution of the trigger fails the actual event will also be rolled back. This can be beneficial for logging purposes because no action can happen without successful logging. In contrast the DDL triggers fire after

5 Rick, B., Bruce, H., & Craig, G. (2016). Track Data Changes (SQL Server). Retrieved 28 December 2017, from https://docs.microsoft.com/en-us/sql/relational-databases/track-changes/track-data-changes-sql-server

events which change the database schema, for example *create*, *alter* and *drop*.[6] DML triggers are more tied to the normal end use of the database and DDL triggers to the development and maintenance. In theory they could both be used for auditing purposes by creating triggers which fire after audited events, be it data or scheme change, and write the event information (what changed, timestamp and user information) to the log table. However in this case only events which are important from the business logics point of view were considered important, meaning that only the actions committed by the end users needed to be audited. For this purpose only the DML triggers would be needed.

The biggest advantage of the triggers, namely their ability to run diverse code, is also a big weakness. Nowadays it is often preferred to include only as few logic to the database as possible. This is mostly because actual programming languages and frameworks provide better tools for development and debugging making the development and especially the maintenance of the application level logic easier than the database logic. Triggers transfer or duplicate the logic from the application to the database. From the point of view of the application this logic is hidden and it can create significant problems in maintenance and debugging. For example a case where a bug is caused by the combined effect of the application logic and logic hidden in the database triggers. In this case the developer has firstly be aware of the existing triggers, which is not always the case, and then has to debug both the application and the database. DML triggers create also a second kind of maintenance issues because they are table specific. Meaning that if one kind of trigger is needed to multiple tables it must be created and in future maintained for each table separately or alternatively additional functionality must be created for automating this process.

6 Byham, R., Hamilton, B., & Guyer, C. (2017, March 14). DML Triggers. Retrieved 14 November 2017, from https://docs.microsoft.com/en-us/sql/relational-databases/triggers/dml-triggers

3.4.3 SQL Server Temporal Tables

Temporal features were introduced to SQL standard in 2011 but SQL Server adopted them only in the 2016 version of the software [7] (Kulkarni & Michels, 2012). In essence a temporal table is a system versioned table which keeps record about all the changes which are made to the data contained in the table. This is done by adding period columns to the main table and creating additional history table which has identical structure to the main table. The period columns are normal SQL columns with *datetime* type. The other column marks the beginning of the validity of the row and the other the end of the validity. The period definition was implemented with two columns to make the adaption of the temporal features easier, since many of the tools which utilize SQL do not have support for the actual period data type. When data is inserted, updated or deleted from the main table the period columns are updated accordingly and row is added to the history table. When system versioning is turned on for the table the user cannot directly interact with the history table which enhances the integrity of the history data. (Kulkarni & Michels, 2012) However by turning the system versioning off it is possible to modify the history table like any other database table but once the versioning is turned back on the database can automatically detect the holes in the history data.

The querying of the history data is done with new *select* parameters which can return the state of the row at a specified point in time (*as of*) or all the states within certain time period (*between and*, *contained in*, *from to*). The default behavior is to return the current state of the row which provides perfect backwards compatibility with existing queries where additional parameters are not defined. (Kulkarni & Michels, 2012) The parameter values are compared to the period columns of the rows and in practice they are no more than specified *where* clauses to these columns.

7 Rabeler, C., Hamilton, B., Sauber, S., Milener, G., & Guyer, C. (2016). Temporal Tables. Retrieved 19 November 2017, from https://docs.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables

The greatest strength, namely the ability to record every change ever made to the database table, is also the biggest caveat of the temporal tables. Recording the full state of every change generates extensive amounts of data which has to be stored, processed and maintained. Activating the system versioning is an table operation and it cannot be configured to only include certain columns of the table. This means that database should be designed to support the system versioning from the ground up so that data which is not critical for versioning can be placed to separate tables which will not need system versioning.

3.4.4 Implementation in application

All of the previous methods are implemented in the database which is natural because most of the time the data which is the target of the auditing is stored in the database. However with suitable architecture the functionality can just as well be implemented in the application. This way the functionality is not dependent on any specific database product or on any database at all. Additionally the solution could be implemented in higher level programming language rather than in SQL, making it more maintainable. Also when the auditing is implemented in the application the user could be identified more easily than in the database.

In order to make the application level audit trail functional and reliable there has to be centralized way for interacting with the data storage, be it database or some other form of storage. In object oriented design this means for example that stored data must be retrieved, created and altered through classes which are inherited from common base class which declares and implements required methods for accessing data. If this kind of architecture is in place the audit trail functionality can be added to it by extending the base class to record auditing information when data access operations are performed. There are also various frameworks for object-relational mapping which handle the conversion from application objects to database entities and vice versa.

The biggest downside of application level implementation is the requirements which it imposes to the software architecture. They are especially problematic for existing software which might require extensive refactoring in order to achieve centralized data access

functionality. Secondly the application level implementation is by default slower than purely database level implementation due to overhead associated with communication between application and database. Often this is not a problem on applications with moderate transaction volumes but it can prove out to be the performance bottleneck in larger applications with high volumes of transactions.

Table 6: General advantages and disadvantages of the alternatives for audit trail implementation. Advantages are marked with + and disadvantages with -.

Method	Characteristics
SQL Server Change Data Capture	+ Asynchronous functionality which increases performance + No need for schema changes + Automated cleanup mechanism − By default will not record the trail of changes
Database Triggers	+ Diversity + Well documented because of their popularity − Diversity − Maintenance issues − Synchronous functionality has impact on performance
SQL Server Temporal Tables	+ Automatic functionality for recording the changes + Reliability and integrity are builtin in the functionality − No control over the recording functionality which increases the size of the recorded history data
Implementation in application	+ More advanced tools available for higher level programming languages. + User information is available in the application level by default. + The auditing logic is visible to the application. − Poses requirements for the architecture of the application − The performance is bound to be slower than in database implementations.

3.5 Comparison in the context of case ERP system

For choosing the final implementation method it was necessary to compare the strengths and weaknesses of the possible solutions in the actual context of the case ERP system. This meant taking into account the characteristics of the system: large user base, high volumes of transactions, complexity of the system, used technologies as well as the previously described requirements: reliability, usability and performance implications.

3.5.1 SQL Server Change Data Capture

While CDC is not actually designed for implementing audit functionality it still offers the main functions for auditing, namely it records the fact that data has changed and the changed values as well. All this happens automatically in the database once the feature is enabled and configured. Additionally the data is captured asynchronously from the transaction logs which minimizes the effect on database performance. This is important because the case ERP system processes tens of thousands of operations daily. Thus the biggest advantages of the CDC are the automatic functionality after configuration and minimal impact on the database performance.

However by default the change data is kept for relatively short period of time meaning that actual audit trail implementation would require additional functionality which would read the CDC data and store it in more permanent way. In practice this would have to happen every time data is changed, because CDC records only the initial and current states of the data and not the states which might have been valid in between these states. Also the initial configuration of the CDC was seen complicated especially because there was no previous experience about its usage within the personnel of the company. This meant that extensive preliminary research about the constraints and considerations associated with CDC would be required before selecting it as the solution. Furthermore the CDC would not record the user data by default meaning that additional functionality would need to be created in order to be able to associate users with the changes.

3.5.2 Database Triggers

The database triggers provide greater opportunities for customization than the other database level solutions. With triggers it becomes possible to select the audited information at column level, for example only record changes to specific columns within a table. The format of the recorded data could also be freely designed. However the freedom of the triggers comes with a price because the functionality has to be enabled table by table and be coded manually. This creates additional maintenance job which could be reduced by developing additional framework which could automatically create triggers for desired tables. This way the maintenance process could be more or less automated similarly to the CDC and Temporal Tables. However the development of the said framework would require significant effort and it would be more error prone than the out of the box functionalities of the CDC and Temporal Tables. Additionally the past experience with triggers had shown that their performance was not good enough in many case because of the synchronous insertion of the history data. Meaning that users would notice significantly longer loading times on features where the trigger based auditing was enabled. However the performance could be improved by redesigning the underlying database design.

3.5.3 SQL Server Temporal Tables

Temporal tables were new feature in SQL Server 2016 which was promoted to be designed for data auditing purposes, among with data analysis and point in time analysis [8]. It enabled the automatic recording of all the data changes, much like the CDC, but unlike CDC it did not require any customization to be suitable for auditing. The lack of initial customization work would also make the solution more reliable compared to the CDC and triggers and would also reduce the effort needed for maintenance. In addition to the change recording the temporal tables also offer a dedicated syntax for querying the change data. The existing SQL queries would return the current state of the data while the new types of queries could

8 Rabeler, C., Hamilton, B., Sauber, S., Milener, G., & Guyer, C. (2016). Temporal Tables. Retrieved 19 November 2017, from https://docs.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables

return the history from certain period or at certain point in time (Kulkarni & Michels, 2012). These queries would make the utilization of the history data more straightforward because there would be no need to create unified functionality for querying the data.

Because of its comprehensive and automatic nature the temporal tables feature leaves little room for customization. While this reduces the amount of work needed for its deployment and makes it less error prone it also means that it can only be enabled on the table level. Meaning that when the versioning is enabled changes to the all the columns in the table are recorded. This is problematic for the mature and naturally grown system where the tables often contain more columns than is optimal and redesigning of the database structure to be more manageable is costly. In addition to this rigidity the temporal tables also place additional constraints for the database. Most importantly they cannot be used with foreign keys with cascade rules. Example of such cascade rule could be the database constraint which connects invoice rows to the invoice. When the invoice is deleted its rows can also be deleted or their reference to the invoice can be set to *null* with the cascade rule. This is important for maintaining the referential integrity within the database. This is severe restriction with temporal tables which was fixed in the 2017 version of the SQL Server. However in this case the migration to this version was not currently possible meaning that referential integrity would have to be forced in some other way for example in the application level. Additional constraint was the incompatibility of the temporal tables and *instead of* triggers which would generate additional work for recording the user who made the change. Also the performance of the Temporal Tables seems questionable because the history data is inserted synchronously with the actual data meaning that the performance cost is similar to the triggers.

3.5.4 Implementation in application

The implementation in the application level would be the most manageable solution because it could be implemented with high level programming language with advanced tool support for example for debugging. The functionality of this solution would not be completely automatic like with temporal tables, but with better tool support and with possibility to utilize automated unit testing the likelihood of errors could be greatly

reduced. Additionally compared to the database level solutions the id of the user could be easily obtained in the application meaning that there would be no additional work for implementing such a functionality.

The most drastic drawback for this method is the fact that currently in the system there is no unified point for interacting with the database. In the newer parts of the application there is extensive data access layer which handles most, if not all, of the database interactions and in any case it could be extended to be comprehensive. However in the older parts of the application the comparable layer for data access is less comprehensive and database can be accessed in various ways from various locations. This means that there is no suitable place for implementing the auditing functionality in the application without major refactoring of the system. Additionally the performance of the solution is questionable because in addition to the synchronous insertion of the history data, which is comparable to the triggers and temporal tables, there is also overhead with communication between the application and the database. In theory the functionality could be implemented asynchronously but this would be complicated requiring more work and increasing the error risk.

4 IMPLEMENTING THE NEW AUDIT TRAIL

The possible implementation methods were evaluated and compared based on their suitability for fulfilling the previously defined requirements for audit trail. The results of the comparison are illustrated in table 7. In order to make the comparison more descriptive the requirements of the performance and usability were split down to storage space and processing time performances and ease of implementation and maintenance. The alternatives were ordered based on their alleged performance derived from the literature and experiences within the development team of the ERP system. Extensive measurements were not made at this stage and for this reason there are several cases within the comparison where two or more alternatives are seen as equal. This means that no meaningful difference was seen between these alternatives.

Table 7. Comparison of the different methods based on the audit trail requirements of ERP system.

Requirement	Alternatives listed from best to worst
Reliability	1. Temporal Tables 2. Change Data Capture 3. Triggers & Implementation in application
Performance (storage space)	1. Change Data Capture & Triggers & Implementation in application 2. Temporal Tables
Performance (processing time)	1. Change Data Capture 2. Temporal Tables & Triggers 3. Implementation in application
Ease of implementation	1. Temporal Tables 2. Triggers & Change Data Capture & Implementation in application
Ease of maintenance	1. Temporal Tables 2. Implementation in application 3. Change Data Capture & Triggers

Firstly the reliability of the audit trail could most fully be achieved with temporal tables because the recording process is fully automated and the integrity of the history is guaranteed by the database. With CDC the recording process could be automated as well but there are no similar guarantees for the integrity. Secondly storage space wise the performance is similar between all the methods except for temporal tables which has worst performance in this regard. The difference is that with other methods the recorded data can be configured column by column and only changed columns can be recorded, whereas with temporal tables the whole row is recorded when one column changes. This generates a lot of redundant data but on the other hand it also means that state of the row in the past can be reconstructed by retrieving just one history record. In the case where only the changes are recorded the reconstruction would require iterating over multiple changes. Processing time wise the best alternative is the CDC because of its asynchronous functionality. Temporal tables and triggers are seen as equals because of their synchronous functionality. Implementation in application is last because in addition to storing the history data, which is common for all the methods, it is associated with the additional cost of communicating between application and database. However this additional cost could be removed if the history is always stored with the same query as the actual change. The ease of implementation is greatest for the temporal tables because it requires only simple activation process for each table. The other alternatives are tied because they would require either significant effort for implementing the functionality in maintainable manner or significant changes to the architecture of the application. Because of the mostly automated functionality the temporal tables are viewed as the most maintainable alternative as well. The implementation in the application was second because if it could be put in place there could be centralized place for the auditing functionality, which is not true for database solutions.

Based on this comparison the temporal tables and CDC were the strongest alternatives. Out of these the temporal tables was chosen as the final solution. While the CDC has more advantages performance wise the usability of the temporal tables was seen as more important factor especially because the use of CDC would require extensive work for fully adapting the functionality to audit trail use case. Furthermore the discarded alternatives all had some critical issues associated with them. Even if the trigger performance could be

improved, extensive amount of work would still be required for implementing a framework which would make them maintainable. Likewise a lot of work would be needed for providing additional functionality for CDC to be suitable for storing the audit data. Although the CDC would have performance benefit compared to the temporal tables. The implementation in the application is not viable solution because there are numerous ways to communicate with the database in the system.

After choosing the temporal tables as the solution for implementing the audit trail, a proof of concept was developed to confirm the suitability of the solution and to highlight the temporal table features to stakeholders. The proof of concept contained all the necessary functionality for the audit trail but it was activated only on the limited part of the system. The part chosen for the proof of concept was the user rights of the system. They were chosen because they were seen as the part which required the most work from the customer support, even though they were part of the existing audit trail solution. In addition to the activation of the temporal tables the proof of concept also included solutions for storing the user information for changes and clean up functionality for history data. By default these were not part of the temporal tables functionality and in fact the implementation of these features turned out to be the most laborious part of the proof of concept. However it should be noted that this was not specific for the temporal tables and all the other considered alternatives would have required similar effort for the user information recording and clean up functionality.

In the context of temporal tables several easily confused terms are used. In this work t*emporal tables* is used to name the feature in the SQL Server 2016 and *temporal table* is a database table on which this feature is turned on. Synonyms for these terms are *system versioning* (the feature) and *system versioned table* (table on which the system versioning is turned on). When system versioning is turned on a *history table* is created which replicates the scheme of the actual table (which is now temporal or system versioned table).

4.1 Recording the user information

Crucial step for providing useful audit trail is the capturing of the “last updated by the user” information, which tells who has made the change to the data. This is challenging because temporal tables does not have built in support for identifying the user who has made the change. However this same challenge is present with other considered database level alternatives as well. The identification is done by adding new column for the user’s id to the system versioned tables. This way the system versioning would record also the changer information. This approach is similar to the existing auditing identification of the ERP system. However additional steps are taken to address the shortcomings of the current approach regarding the reliability of the identification information. In the new audit trail the database is responsible for creating the user information for *insert* statements through the column default value which is defined to be the id of the user who has inserted the row. This means that there is no need for the application to include this information in the *insert* statements. However the application is left with the responsibility of updating this information when *update* events are performed on the data. The value can be provided by the database but the application must include the user id column in the *update* statements so that it is updated with the new value. It would have been more preferable to leave the handling of the both *inserts* and *updates* completely to the database because in the system there is no completely centralized place for interacting with the database. Since the updater id would need to be included in everywhere where the *update* statements were made this was somewhat error prone. Especially considering that by default the missing id would not cause any exceptions or even runtime errors but rather it would result in the rows containing wrong information for the updater id. In theory the *update* could be handled with *instead of trigger* which would run instead of the *update* statement and perform the *update* with the updater id. However these triggers were not allowed in system versioned tables in SQL Server 2016[9]. On the other hand the *after trigger* which would update only the updater id was not suitable either because it would result in creating duplicate rows in

9 Rabeler, C., Hamilton, B., & Andzic Mladen Guyer, C. (2017). Temporal Table Considerations and Limitations. Retrieved 23 November 2017, from https://docs.microsoft.com/en-us/sql/relational-databases/tables/temporal-table-considerations-and-limitations

history for each *update*. The first history row would contain the new values with old updater id and the second would contain the updated updater id. In order to overcome these issues an *after trigger* was created which only checks that the updater information is included in the *update* statement. If the information is missing the trigger throws an error and rollbacks the transaction so that the *update* statement is reverted. This way the database will be responsible for forcing the application to include the mandatory information and thus the correctness of the user information can be trusted.

The problem of recording the user information was even more complicated in case of deletions. In this case there were the same problems as with the updates but additionally they could not be resolved with similar *after trigger* because after the deletion there would be no row left for the trigger to write the user id to. To solve this issue additional table was created to hold the deleter information for system versioned tables. This table contained the user id of the deleter as well as the id of the deleted row and a timestamp which would tie it to the row in the history table. Additionally reference to the original table was added so that deletions from all of the tables could be written to the same table. This means that this table would grow largely over time and for this reason an index which contained the columns needed in the temporal queries was added to improve the query performance. Despite the performance impact this approach was seen more desirable because it is more maintainable than creating separate deletion information tables for each of the actual tables.

4.2 Altering the foreign keys

The temporal tables have the significant drawback of not being compatible with the foreign keys with cascade actions. Cascade actions define the functionality which applies to cases where the parent entry of the foreign key relationship is deleted or updated. Possible actions are cascading the operation, which means deleting or updating the child entries as well, setting the foreign key column of the child entries to null or doing nothing which essentially means that parent entries can not be deleted as long as they have child entries. With temporal tables the only allowed action is the do nothing, which means that the cascade functionality must be implemented in the application layer where it is needed.

More precisely when the table is system versioned it can not be the child table of the cascade foreign key relationship. The limitation is not present if the parent table is system versioned. Examples of these are shown in figure 1 with example database structure with three entities: vendor, invoice and invoice line. Invoice line has foreign key to invoice which has foreign key to vendor. Initially both foreign keys have cascade action and the figure shows how these must be changed when each of the tables is versioned.

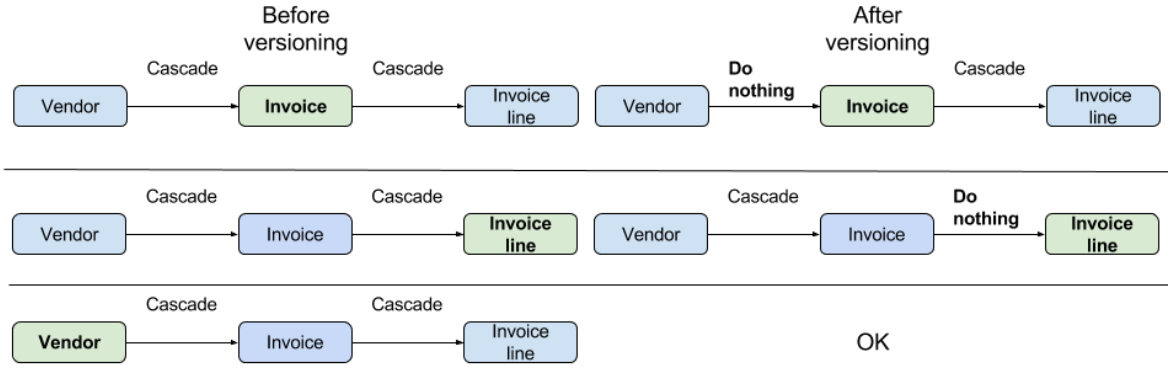

Figure 1. Examples of cascade constraints with temporal tables. On the left side is the initial state and on the right side the necessary changes are highlighted. The table to be versioned is marked with green background color.

The cascade logic is implemented in the application by adding the needed operations to the points where database interaction occurs. So for example a method which deletes entries in the parent table deletes first the possible child entries of the parent. This approach is possible when there are centralized modules in the application for interacting with the database objects. This was not completely true for the case ERP system which meant that for some entities the interaction points would have to be manually searched and the cascading functionality added. Since the limitation is only present when the versioned table is the child of the relationship the cascade logic must be added to the parent objects in the application. These are the only places where modifications are needed even though the cascade operation is recursive. So if the versioned table is part of the longer chain of cascade foreign key relationships only its parent objects, not parent's parents, need to be changed when versioning is activated for it.

4.3 Activating the versioning

The most elementary part of the database changes was the activation of system versioning. While this process could be scripted to happen automatically to multiple tables in the database at once, it was decided that it was more manageable to do it table by table especially at the proof of concept stage. The following code example in the figure 2 shows the example SQL script for adding system versioning to the existing table. The activation requires two *datetime2* columns to be added in the versioned table for indicating the period of validity of the record. The data in these rows is then automatically generated by the database system. The validity of the record starts when it is inserted or updated and it ends when the record is deleted or updated to something else. The optional *hidden* clause means that the columns will not be returned in the query results unless they are explicitly included in the query. This way the functionality of the existing queries is guaranteed to remain the same after the activation of the system versioning. After the columns are added they must be defined as *period for system_time* and then the system versioning can be turned on. (Kulkarni & Michels, 2012)

```
CREATE SCHEMA History;
ALTER TABLE InsurancePolicy
  ADD
    SysStartTime datetime2(0) GENERATED ALWAYS AS ROW START HIDDEN
        CONSTRAINT DF_SysStart DEFAULT SYSUTCDATETIME()
    , SysEndTime datetime2(0) GENERATED ALWAYS AS ROW END HIDDEN
        CONSTRAINT DF_SysEnd DEFAULT CONVERT(datetime2 (0), '9999-12-31 23:59:59'),
    PERIOD FOR SYSTEM_TIME (SysStartTime, SysEndTime);
GO
ALTER TABLE InsurancePolicy
  SET (SYSTEM_VERSIONING = ON (HISTORY_TABLE = History.InsurancePolicy))
;
```

Figure 2. Example of temporal table activation [10]

10 Rabeler, C., Hamilton, B., Sauber, S., Milener, G., & Guyer, C. (2016). Temporal Tables. Retrieved 19 November 2017, from https://docs.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables

In addition to the actual activation of the versioning two other steps had to be taken at this stage so that the system versioning would meet the requirements set for audit trail. First of all the history table had to be indexed for improving the query performance in potentially large history tables. Secondly a partition scheme and function needed to be created for enabling the maintenance of the history data.

Two indexes were added to the history table for improving the query performance and decreasing the needed storage capacity. Firstly a clustered columnstore index was created for improving both the query performance and data compression. Columnstore index is designed for data warehousing scenarios where the data in the tables is not updated which is the case with history tables. In such scenarios the indexing can achieve up to ten times better query performance as well as compression rates.[11] This was crucial for audit trail functionality since history tables could contain up to hundreds of thousands and potentially even millions of rows so tenfold performance improvement would be huge when measured in absolute terms. In addition to the column store index normal index containing the timestamp columns, user id and id in the actual table is created for further improving the query performance. Columns of this index are chosen based on the most likely use cases which are querying for all the changes made to specific row (id) and for all the changes made by the specific user (user id).

The second important step, namely the partitioning, means dividing the table into separate units which can be stored in different filegroups. The partitioning of the history table was necessary for being able to remove data from the table. This functionality is described more accurately in the chapter 4.4, but the initial partition scheme is presented here. When system versioning is turned on the user cannot directly interact with the history table meaning that it is not possible to delete rows from it. However it is possible to partition the history table and then switch out partitions from it to the *temporary* table (not to be confused with temporal table) which can then be *dropped*, essentially deleting the rows.

11 Barbara, K., Hamilton, B., & Guyer, C. (2016). Columnstore indexes - overview. Retrieved 27 November 2017, from https://docs.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-overview

The temporary table must have identical scheme compared to original table of the partition. Furthermore handling data in partitions increases the performance of the history data clean up process because, instead of individually deleting the rows which are old enough to be deleted, the whole partition can be deleted in bulk. In practice this means “moving” the partition from the history table to the temporary table and then dropping the whole table at once. This is not computationally expensive operation since it does not require actual data movement because the rows are not moved between tables. Rather the database is told that this partition is no longer part of the history table but it is now a part of the just created temporary table. Then the whole temporary table can be dropped at once.

The history tables were partitioned by the *SysEndTime* column, which presents the end time for validity of the entry, because it is constantly increasing column in the history table and that is requirement for the partitioning. It should be noted that *SysStartTime* column, which presents the start time of the validity, would not have been suitable for the partition function because its values are not constantly increasing as can be seen by simplified example: record X is created in 1.1.2000 and record Y in 2.1.2000, record Y is updated in 3.1.2000 resulting in creation of history table row with *SysStartTime* 2.1.2000, then record X is updated in 4.1.2000 resulting in creation of history table row with *SysStartTime* 1.1.2000 which is smaller than previously created history row for record Y and shows that *SysStartTime* is not constantly increasing.

The partitioning is phased so that each partition holds the history data from specific month and it is defined as RANGE LEFT partitioning meaning that the defined partition limits are the upper limits of the partition. Example of this is shown in figure 3. If row is added to this table with *SysEndTime* 1.12.2016 it will be located in the partition 1 because the value is smaller than the smallest partition limit. If the *SysEndTime* was 2.2.2017 the row would be in the partition 3. This shows that each partition represents a month except the first and last partitions which hold the values which are less or more than the outermost limits of the partitions. These are also the most vital partitions for the retention management process because both of them have to be empty when *merge* (partition 1) and *split* (partition 4) operations are performed during clean up process. If they are not empty the operations cause movement of the data which can be potentially expensive if the amount of the data is

large. For *merge* and partition 1 this is given because data in the partition is switched out before merging meaning that it will always be empty at this stage. However for the *split* and partition 4 there is no similar guarantee and thus the partition function must contain enough partitions in comparison to the retention time of history table so that there will always be empty partition for the *split* operation. The number of required partition limits is the retention time in months plus three. Additional limits, or in essence months, are needed for the following reasons. Firstly when the maintenance process is run there has to be empty partition at the end of the partition scheme so that it can be splitted without data movement hence the first additional limit. Secondly to ensure that the system has at least retention time's worth of history at all times the second additional month is needed. If there were no month dedicated for this then the system would have retention time's worth of history when the maintenance process is run but directly after that the system would hold the history data for one month less than the set retention time. These two additional limits would be enough to make the sliding window partitioning functional but third additional limit is added to give more time to react in error conditions. Here error condition means situation where the clean up process is unsuccessful for one reason or another. In such a scenario it is important to be able to fix the issue in time so that the last partition remains empty for the *split* operation. By adding additional limit there is always at least a month for repairing the issue regardless of the time of month the maintenance process is run. Without this month the time to fix the issue would be the time from the moment the process is run to the end of the current month. The functioning process is illustrated in the figure 5 in next chapter.

Figure 3. Example of the left range partitioning

4.4 Clean up process for history data

The last major step in the back-end functionality is the automated clean up functionality which periodically deletes the history data that is older than the chosen retention period for the data. This is important because the size of the history data will grow over time. This is

also the most complex part of the auditing system because implementing sliding window partitioning scenarios requires careful planning since once the functionality is deployed changing it could in worst case require the deletion of the previously recorded data. Such scenario can occur if the partition scheme is broken during the process, for example partition is deleted but new one is not created. Over time this would result in a case where there is no empty partition left at the end of the partition scheme and thus creating a new partition would require splitting the last partition which now contains data and this would make the split operation computationally expensive because data would need to be moved to temporary table row by row and then inserted back after the *split*. Solving this situation would have to be done in a way which will not cause blocking in the system or the versioning would need to be reseted and the existing history data would need to be deleted or moved elsewhere. Malfunction of the process could also mean performance issues in the database which are hard to notice and debug. The size of the ERP system places additional robustness requirements for the functionality since there are numerous databases which can have slight variations in structure. Lastly the partitioning by time meant that testing could not be done in 100% authentic way because it is not feasible to wait a month, which was the theoretical minimum retention time and test if the history is cleared correctly. Of course the actual retention times would be even longer. For this reason it was necessary to make the process not dependent on the current time so it could be run with mock timestamp.

The task of periodically deleting or archiving the old history data was divided into two main steps: first a stored procedure was created in each database that contains system versioned tables alongside a table which holds the information about system versioned tables such as name, retention time and information about constraint names of the table. This meant that database could contain system versioned tables with varying retention times. Secondly a SQL Server Agent Job was created to periodically run the procedure on all databases within the system.

The stored procedure would iterate over system versioned tables and it would delete the partitions which were older than the history data retention time. Every time a partition is deleted a new partition is added to the end of existing partition scheme to hold the new

history rows. Each table is processed inside a transaction so that the number of partitions is guaranteed to remain the same. The high level logic of the process is described in pseudo code in the figure 4. The actual logic was implemented in SQL.

```
timestamp = parameter1
historyTables[] = getHistoryTables()

For Each table In historyTables:
  oldPartitions[] = getPartitionsOlderThanRetentionTime(table.retentionTime,
                    table.partitionFunctionName, timestamp)

  For Each partition In oldPartitions:
    tempTable = createTempTable()
    SwitchOutPartitionTo(tempTable)
    Drop(tempTable)
    MergePartition(partition)
    newestPartition = getNewestPartition()
    SplitPartition(newestPartition)
  Next partition

Next table
```

Figure 4. High level presentation of the clean up logic

In the procedure the information of system versioned tables is retrieved from the documenting table, which holds information about all the versioned tables in current database, to the cursor which fetches the needed information (table name, retention time, partition function name) for the current iteration. The old and to be removed partitions are then fetched from the system tables of the database based on the partition function name, retention time and timestamp which was given to the procedure as a parameter. Normally the timestamp will be the current timestamp but for testing purposes an arbitrary timestamp can be given so the functionality can be tested for example as if the time was a year ahead of the actual current time. The temporary table for the *switch out* is created by selecting from the history table to the temporary table with *top 0* clause which essentially copies the history table scheme to the temporary table without copying any actual data. After this the oldest partition can be switched to the temporary table and then the table can be dropped.

The new partition is created by splitting the current newest partition which is found from system tables similarly to the oldest partitions. The whole process is then repeated for each partition which is older than the specified retention time. It should be noted that this process cannot be statically scripted because of the variables like table name and retention time. For this reason the actual procedure generates the queries dynamically from the static commands and the variables and then executes them. Since the commands are dynamically generated it is a good practice to address the risk of SQL injection, even though the variables can not be modified by end users. This was done by providing the variables as parameters to the procedure which executed the SQL commands and by enclosing the database object name variables with brackets rather than just appending the variables straight away to the command string. This way the variables will always be interpreted as data and not as SQL commands.

The process of merging and splitting partitions is illustrated in the figure 5 with adequate partition scheme for 6 months retention time. The process is run on 15th day of each month, but the actual date is irrelevant for the functionality of process. In the image each partition limit is shown as a date and each partition represents month expect the first and last partitions which are open ended and would hold all the records with smaller or greater values than the respective limits. The limits are the upper limits of the partitions and for this reason the limits seem the appear a month behind. For example at the initial state the row with date 10.1 will be placed to the partition with limit 1.2 and thus this limit is shown in the January column rather than in February. Each row represents a moment in time and the current month is marked with light brown background. Partitions to be merged are marked with red text and newly created partitions are marked with purple text. The purple background marks the splitted partition.

Row	*Dec*	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sept	Oct	Nov	Dec	*Jan*
1	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9					
...														
2	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9					
3		1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10				
4			1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10	1.11			
...	**Error in the process**													
5			1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10	1.11			
6					1.5	1.6	1.7	1.8	1.9	1.10	1.11	1.12	1.1	
7						1.6	1.7	1.8	1.9	1.10	1.11	1.12	1.1	1.2

Figure 5. Illustration of sliding window partitioning functionality.

The row 1 displays the initial state of the partitioning in December when the system versioning is activated, partition scheme created and sliding window maintenance process deployed. There are 9 partition limits, 6 months + 3 additional as described in the chapter 4.3, which results in 10 partitions. The row 2 shows the state in next year's July when the process deletes history data for the first time. When the process is run on 15.7 the partitions with limits before 15.1 are deleted, in this case the 1.1 partition. The last partition, which would contain values greater than 1.9, is splitted and new partition limit with value 1.10 is created as shown in row 3. The splitting will not cause data movement because the last partition will be empty. This is certain because current date is 15.7 and the last partition would contain records with timestamp greater than 1.9. The same process occurs in August (row 3) and partition with limit 1.2 is merged and new partition with limit 1.11 is created. The transition between rows 4 and 5 represents error condition where the maintenance process fails to complete on September (row 4). Assuming that the issue is fixed by the end of the October the process now finds two partition limits to be merged (1.3 and 1.4 on row 5). They are merged sequentially and after merging the 1.3 partition new partition is created by splitting the partition which holds the records with values greater than 1.11. This partition is still empty because it was specifically created for this scenario where the monthly execution of the process has failed. After this the second merge and split are performed as usual. Row 6 shows how the process has now recovered to its initial state with two partitions proceeding the current month. If the issue was not resolved in time the maintenance process would still be functional but it could require significantly more time

to execute. This is because the last partition would have started to fill up with the history data of current month and splitting it up would require inserting the rows to temporary table, deleting them from the partition and then inserting them back after the split. This is a size of data operation meaning that the more rows there are the longer the process takes. The actual impact on the performance of the system is hard to evaluate or test. It is possible that there would not be noticeable difference on the performance of the system even if this scenario realized. However because this could not be verified it was more desirable to try to prevent this from happening by adding the additional partition for giving more time to react on error conditions.

The previously described stored procedure is run with SQL Server Agent Job which is built in functionality that can be scheduled to run SQL scripts. The script iterates over all the databases and performs the stored maintenance procedure. It will also write information to the logging database which contains logs from other similar background tasks within the ERP system. This way the progress of the process can be observed and potential issues are located. The agent job is scheduled to run on the weekend in the middle of every month. The actual date has little significance for the maintenance processes functionality as long as it is not too close to the partition limits in other words turn of the month. This could create a situation where the number of deleted partitions changes during the processing and some databases would then have different partitions than the others. In theory this should not be a problem but since the scenario is complicated and hard to test it could lead to unexpected behavior and thus it should be avoided. From the business perspective it also makes sense to perform the maintenance in the middle of the month because on average the ERP system is used more heavily on the turn of the months and thus if the clean up process would fail in a way that would directly affect the users the harm would be lesser in the middle of the month. The same effect is strengthened by scheduling the process to run in the weekend.

4.5 Deployment and communication

The deployment of the auditing functionality needs to be carefully planned, because it consists of several modules which are mostly autonomous, but there are also some critical dependencies. This means that the deployment must be done in partially ordered fashion. The final functionality can be seen as compromising from modules with following task: user information recording, storing user information for updates, storing user information for deletes, enforcing the presence of the user information, modifying the database constraints, creating support structures, activating the versioning and partitioning and lastly maintenance plan. The dependencies and essentially the deployment order of these modules is presented in figure 6. The other possibility would be to deploy the whole functionality at once, but normally it is preferable to do the deployment in smaller sets so that the possible issues are easier to locate. Furthermore it is more preferable to activate the audit trail to the system gradually and not at once, which is what was done in the case study where all the functionality was developed but the audit trail was firstly deployed only to the limited part of the system.

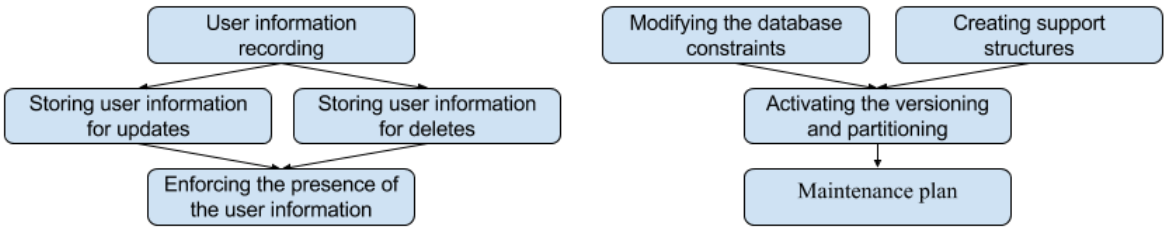

Figure 6. Dependencies between audit trail modules

As can be seen from the figure the modules form two separate processes which do not have dependencies and could be deployed simultaneously. The first is the activation of the user information recording, where the user information must first be recorded, then stored for updates and deletes and lastly the presence of the information can be enforced. The other process is the activation of the actual versioning, which firstly requires removing the possible constraints with cascade operations and creating support structures for the

auditing functionality, for example the documenting table which lists the audited tables. Then the versioning can be activated with partitioned history table and lastly the clean up functionality is created.

The knowledge sharing about the new audit trail functionality was done by providing extensive documentation about its functionality and usage. This was done by documenting the internal functionality of each of the above modules as well as providing higher level instructions about the steps which are needed when the audit trail is activated to the new part of the system. This information was stored to the intra-net of case company for future reference and it was presented to the development team in a scheduled knowledge sharing session. Furthermore instructions on how to monitor the new functionality, for example the growth of history data and execution of the maintenance process, needed to be communicated to the application management personnel.

5 RESULTS

The suitability of the proposed proof of concept was evaluated based on the previously described requirements: reliability, usability and performance which was divided into storage space use and computational performance. Based on these three factors the overall suitability of the solution was evaluated as well.

5.1 Reliability

The reliability of the audit trail was the most basic requirement for the new solution. With the temporal table based solution the reliability is for the most part built in to the functionality. When system versioning is turned on every change to the actual table will create row to the history table with appropriate timestamps. If for same reason the creation of the history row fails then the actual change is reverted as well. The integrity of the history data is also guaranteed by built in functionality since the history data can not be altered when the system versioning is active. Some of the alteration attempts involving history data tampering by turning the versioning off can also be automatically detected when the versioning is turned back on. However this applies only when history records are deleted or timestamps are altered and thus the reliability of the audit trail can be held questionable if malevolent user gains administrative access to the database. However this risk is valid for basically all imaginable audit trail solutions.

By default the temporal tables will not take into account the correctness of the identity of the user but in the proof of concept this problem is overcome by using triggers and other database functionality for providing and ensuring the presence of the user information. If the information is missing the database will throw an observable error so that the issue can be fixed. This way the user identification is robust against development errors and there are not any conceivable weak spots for the identification which was not the case with the existing solution. In conclusion the proposed audit trail solution manages to reliably answer the basic auditing questions what, who and when. This is achieved mostly with built in functionality with few manual additions. Despite its strengths the reliance on built in functionality creates potential vulnerability because in case of there being error in the

built in functionality the ERP system would be dependent on feature provider for fixing the issue. However because the feature is part of the widely used database system the vendor's reaction time to at least more critical issues is likely to be rapid.

5.2 Usability

The usability of the proposed solution is important for both developers and users. The ability of proposed solution to present the complete state of the data at any point in the past, without need to iterate over multiple history records like with methods that would record only the changed data, means that querying the history data is faster and thus the solution can support user interfaces with wide range of features. This enables the wider use of the audit trail. Previously it could only be used by the support of the case company but with improved functionality it can be used through out the case company and customers can even used it them selves. In this regard it fulfills the requirement of usability for users.

From the developer perspective usability means the amount of effort which is needed for enabling the audit trail for specific part of the system. Minimal effort is desirable because it decreases the required work time and the probability of errors. The proposed method has two mandatory steps and two additional steps which might be required depending on the case. The steps and their sub-steps are described in the table 8. The additional steps are written in cursive.

The first step is to add column to the versioned table for holding the id of the user who has last edited the data. After that the access points from where the versioned table is updated in the application need to be modified to include the user information. After this the triggers can be added for the versioned table for recording the deleter information and enforcing the presence of the user information in the *update* statements. It is important to ensure beforehand that the user information is always present because otherwise the trigger will prevent the data from being updated. The second step applies only to tables which have foreign key constraints with cascade actions. For example for deleting the child entries when parent entry is deleted. These types of foreign keys are not allowed with temporal tables if the temporal table is the child object of the relationship. For this reason

they must be modified to have no cascade actions. Before that the actions must be added to the application so that the cascade functionality is still present in the system. In practice the cascade functionality must be added to the access points of versioned table's parent tables and not to the access point of the versioned table. Furthermore implementing the actions in the application is mandatory for cases where entry with foreign key children must be deleted. Otherwise the foreign key without cascade actions in the database would prevent the parent entry from being deleted. Same rules apply to updates as well. The third step is the activation of the actual system versioning. Here the information of the newly versioned table must be added to the documenting table which is used in the clean up process. Then the predefined activation script can be altered to have the appropriate number of partitions, based on the retention time of table, and run. Last step is ensuring the backwards compatibility in tables which have been part of the existing auditing system. This is done by reading the history data from both the new and old audit trails in places where auditing data is used and by deleting the existing audit trigger.

Table 8. List of audit trail activation steps. Steps 1 and 3 are mandatory in each case and steps 2 and 4 are not needed in every case.

Step	Sub-step
1. Recording user information	1. Create column for user information 2. Update data access points 3. Create trigger for recording deleter information 4. Create trigger for enforcing updater information
2. Altering cascade constraints	*1. Add cascade rules to application* *2. Remove cascade rules from database*
3. Activating the versioning	1. Define the retention period and add versioned table to documenting table 2. Run the activation script with appropriate number of partitions
4. Ensuring the backwards compatibility	*1. Replace the old audit log with the new one* *2. Remove old audit triggers.*

The number of required steps is a bit greater than was initially hoped, but most of them are very generic requiring only small alterations to predefined scripts. For this reason they could be automated and this would greatly reduce the effort needed per table for activating the versioning. However the automation process was left outside of the scope of this work. The automation would also reduce the risk of errors. Currently most of the risks associated with activating the versioning have either small impact, for example they affect only the versioning process itself, or they are easy to identify, for example missing user information in updates which generates immediately observable error. The only conceivable exception is the defining of the partitions. If there are not enough partitions created for the history table, compared to the retention time, this can create an error which is only detected when it starts to cause slowness in the maintenance functionality, because there are no empty partitions to be split. This risk is reduced by instructing the developers on how the number of partitions is defined. Overall the usability of the proposed system is seen as adequate but there is room for improvements. Currently it will be challenging for developers to complete all the described steps within an hour, which was the desired time limit in requirements. However the partial automation of the process could greatly speed up the process as discussed.

5.3 Performance

The most critical factors from the performance point of view were the storage space and impact on the responsiveness of the system. Both of these factors are mostly affected by the number and variance of stored history rows which depend on the update frequency of the information. This is determined by the user behavior: how often users update the information, the scope of the possible values (boolean value has two possible values and string value has theoretically almost infinite number of possible values, but in practice the real values are much more limited) and the extent of the update (are all columns updated or only one). However it would have been too time consuming to try to simulate actual user behavior with these variables because they will vary greatly depending on the use case and the nature of the data. For this reason the storage space requirements were mainly evaluated by creating a worst case scenario where entries were updated directly in the database with randomized data. This is not completely realistic use case because in reality

the data is not randomized and thus the storage space requirements per history row found during testing are the worst case scenarios. This is because with real data the possible set of values is much more limited compared to the completely randomized values and thus same values are more likely to appear enabling the compression of the data. This can also be seen by comparing the test cases where every column of the table was randomized to the cases where only single column was randomized. In the later case the compression rates are up to ten times higher. By doing multiple updates it was possible to evaluate how much storage space would be required at most with certain update frequencies and how rapidly the storage size increased with the number of history rows.

The execution performance of audit trail was evaluated in the same manner by updating the data directly in the database and measuring the CPU time of the operations. However in these tests the updated data was not randomized. This was due to technical limitation which meant that the data generation was included in the duration of *update* operation. The initial tests with and without the randomization revealed that the CPU time consisted mostly of the randomization process and thus it hid the actual variations in the performance. The complete results of the measurements are shown in the appendix 2 and 3. The storage space tests were run only once with similar test data. The results are still representative because the measured variable of storage space has none or only minimal random variation, meaning that there is no need for averaging multiple test results. This was also confirmed by initially running few tests multiple times. The CPU time tests on the other hand were run 4 times to average out the small variances between tests.

5.3.1 Storage space requirements

The storage space tests had 4 variables: the usage of system versioning and columnstore index, the extent of the update (all columns and single column) and the number of existing history rows in other words the number of previous updates. The results are presented as space used per data row and as multipliers which describe how many times the actual data can be updated before the history data uses as much space as the actual data, in other words when the amount of required space doubles from the initial state where there is no history recorded. These measures were chosen over absolute measures because they give more generally applicable view on the performance. For example if the space requirement

doubles up after each data row is updated twice then this is true regardless of the database scheme or the number of actual rows. Using absolute measures, for example the actual space used after certain number of updates, is heavily case dependent and would not give as general results. The table 9 highlights the differences of storage space usage in different cases. The sizes shown here are measured after the operation has been run. The complete results with initial states are presented in the appendix 2.

Table 9. Storage space usage per row in different scenarios

Row	Number of previous updates	Operation	System Versioning	Columnstore index	**Size main/ row (KB)**	**Size history/ row (KB)**
1	0	Update every column with randomized data.	OFF	OFF	**1.099**	**0**
2	1	Update every column with randomized data.	OFF	OFF	**1.099**	**0**
3	0	Update every column with randomized data.	ON	OFF	**1.18**	**0.311**
4	1	Update every column with randomized data.	ON	OFF	**1.18**	**0.519**
5	38	Update every column with randomized data.	ON	OFF	**1.18**	**0.717**
6	0	Update every column with randomized data.	ON	ON	**1.18**	**0.02**
7	1	Update every column with randomized data.	ON	ON	**1.18**	**0.19**
8	38	Update every column with randomized data.	ON	ON	**1.18**	**0.354**
9	0	Update single column with randomized data.	OFF	OFF	**0.571**	**0**
10	1	Update single column with randomized data.	OFF	OFF	**0.571**	**0**
11	0	Update single column with randomized data.	ON	OFF	**0.572**	**0.311**
12	1	Update single column with randomized data.	ON	OFF	**0.572**	**0.328**
13	38	Update single column with randomized data.	ON	OFF	**0.572**	**0.343**
14	0	Update single column with randomized data.	ON	ON	**0.572**	**0.02**
15	1	Update single column with randomized data.	ON	ON	**0.572**	**0.033**
16	38	Update single column with randomized data.	ON	ON	**0.572**	**0.047**

The tests revealed that by default the compression rates seem to be greater in the history table and the effect is significantly increased with the use of columnstore index. The better compression rate in history tables by default is interesting because it seems to be undocumented. Some of the difference can be explained by the measurement method because initially the table was populated with not random data. This data would remain in the history table even if the further update iterations would generate only randomized data meaning that some of the data in the history table was not randomized and thus would have greater potentiality for compression while the data in the actual table was totally randomized. However this does not completely explain the difference because even with multiple update iterations the difference hinders only slightly. Furthermore the first update shows that the initial data used on average 0.311 KB (row 3 in table 9) of space in history table and 0.337 KB (initial state of row 1, which is shown in appendix 2) in actual table. This means that history table used 8% less space after first update.

Second conclusion which can be drawn from the results is that the usage of the columnstore index is highly beneficial for the compression rates. Even in the worst case scenario where every column was updated the storage space usage was 63% lower with the index compared to the versioning without index with single prior update (rows 7 and 4) and still 51% lower with 38 prior updates (rows 8 and 5). The difference is even bigger in the more favorable case where only single column was updated: 90% lower storage space usage with single prior update (rows 15 and 12) and 86% lower with 38 prior updates (rows 16 and 13). In the single column update case the advantage in compression rate is roughly tenfold as advertised by the database vendor[12]. Compared to the actual table the required storage space per row is 70% lower with 38 previous updates in the case where every column is updated (row 8) and 92% lower in the one column case (row 16). What this means in practice is that in the theoretical worst case scenario the storage space usage would double up compared to the current usage when every row is updated on average about 3.3 times during the history retention time. In the more favorable scenario, which is

12 Barbara, K., Hamilton, B., & Guyer, C. (2016). Columnstore indexes - overview. Retrieved 27 November 2017, from https://docs.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-overview

likely closer to the reality, the storage space usage is doubled after about 12.2 updates. The actual update frequency is impossible to predict accurately without field testing in the real production environment, which was not possible in this case. However it is unlikely that the average update count for data entries would be measured in hundreds rather than in dozens. This is because most of the data in the system has limited lifetime during which it can be updated, for example invoices. So even if the history data has long retention period the updates can happen only in the much more limited timespan. In any case the storage space requirements will not increase ten folds with any realistic update frequencies which was the minimum requirement for storage space usage. With the discovered factors the ten fold increase in storage use would require 33 updates for each row in worst case and in more favorable case 122 updates for each row.

5.3.2 Performance of the database operations

The execution performance was measured by comparing the CPU time of large update operations with varying audit mechanisms and history data sizes. In practice this meant updating every row in a table which contained about 36 000 rows. There were in total a baseline case, which did not have any auditing mechanism activated, and three test cases: trigger, temporal tables and temporal tables with columnstore index, which was the proposed method in this work. Each test case contained three steps: firstly without history data, secondly with history data equal to updating all the rows of the table five times and lastly history data equal to updating all the rows of the table 15 times. This was done to observe the possible performance impact the accumulation of the history data could have. Exception being the baseline case which did not have these steps because in this case there was no auditing which would accumulate history data. For other cases each of these steps were executed four times to eliminate any random variations in the results. The test cases were also split into two different scenarios: in the first only one column was updated and in the second three columns were updated. This highlights some differences between the auditing methods. This testing method resulted in the 80 different measurements which are shown in the appendix 3.

The results had two surprising discoveries. Firstly there seems to be minimal performance impact associated with size of the history data at least with tested history sizes, which was 543 585 rows of history data at largest. It is likely that the performance impact would have come more observable when the history size increases further but still the impact seems to be smaller than initially anticipated. The second surprising result was that the number of updated columns did not seem to affect the results except in case of the triggers. Because of these discoveries the results from different measurements were averaged for each method and these averages are compared to get a understanding of the differences. The values are presented in table 10.

Table 10. Average execution times of different auditing methods

Row	Trigger	Temporal	Columnstore	AVG CPU-time (s)	STDEV (s)
1	OFF	OFF	OFF	2.346	0.141
2	OFF	ON	OFF	2.426	0.253
3	OFF	ON	ON	2.676	0.18
4	ON	OFF	OFF	3.343	0.503

Naturally the best performance was achieved without logging. In this case the average execution time was 2.35 seconds. The execution time for the existing trigger based logging was the worst of the candidates with 3.34 seconds. It was also heavily affected by the increase in the number of updated columns. With one column the execution time was 2.87 seconds and with three columns it was 3.82 seconds. With other methods the difference was statistically insignificant. Temporal table performance was significantly better than trigger performance with average execution time of 2.43 seconds without the columnstore indexing and 2.68 with it. The columnstore indexing seems to have slight negative effect on the execution performance of the solution, but this is acceptable considering its advantages with query performance and compression. The overall difference between temporal tables and triggers is understandable because of poorer performance of the triggers with multicolumn case. However the difference in single column case is a bit surprising considering that theoretically both methods employ the same functionality of synchronously inserting history rows to separate table which was empty at the beginning of the test cases.

There were some random variation between test rounds and because the results for different methods are relatively close to each other this means that while the differences discovered by the measurements are likely to exists, any conclusions about their scale should be drawn with caution. Furthermore the tests were simplified cases from real life scenarios which are more complicated and have more variations. For example the measured CPU time of the database operation is only small part of the response time users experience while they use the system and it also fails to address the important IO time of the operations. Additionally the used operation, updating every single row in the table, is unlikely to happen in reality and the actual operations are normally much simpler with smaller execution times and absolute differences between the methods. For testing purposes a large scale operation was used to lessen the random variations between the measurements so the actual differences between the methods would be more observable. Likewise, the CPU time was the measure which is least affected by the other variables, like network delay and measurement inaccuracy, which would have been present if measurements were done through for example user interface of the system, even though this method would have been closer to the actual use cases. Despite the challenges in measuring the differences the result strongly suggest that proposed solution has better execution performance than existing trigger based solution and this performance is relatively close to the case where auditing method is not used. Additionally the difference between existing and proposed methods is likely to increase in the real use scenarios where the number of updated columns may vary. The small difference between no auditing and proposed auditing method also means that the impact on performance is not noticeable for users which was the requirement for the proposed solution. The average execution time in the test, which was heavy database operation, was only about 0.3 seconds slower with proposed auditing method which is hardly noticeable for users. With normal database operations and other processing times, for example network latency, the difference becomes insignificant.

5.4 Overall suitability

Overall the proposed proof of concept fulfills the requirements of reliability, usability and performance. In this regards the proposed proof of concept is suitable for system wide use which was the major issue for the existing auditing solution. However there are some issues regarding the usability and performance. For the developer usability there are more steps involved in the activation of the auditing than is desirable. Also since the process requires some manual work there is room for errors, some of which could be critical, for example the incorrect partitioning. However this issue could be resolved with further work by automating the activation process which would lessen the required work and leave less room for errors. Performance wise the test results were successful but since the auditing system could not be tested in the real production environment, which would be the only way to get accurate update frequency of the data, there is some uncertainty about the solutions functionality under real workloads. The uncertainty is mainly concerned with the storage space requirements of the history data. However the testing suggests that compared to the actual data the history data size doubles up when on average every row in the table is updated 3.3 - 12 times, depending on the scope (how many columns) and randomness (how varying the values are) of the updates. The doubling up of the storage space after on average 3.3 updates is the absolute worst case scenario where the data is completely random and even in this scenario the 3.3 factor means that the required storage space is unlikely to multiply with realistic update frequencies especially because the audited data should be rarely changing by its nature. If the update frequency is high it is likely caused by automatic processing which can not directly be attributed to any person and thus will not necessarily require auditing. However in complitely different case where the update frequency of the data would be significantly higher the proposed solution could generate excessive amounts of history data and thus it might not be suitable. In these scenarios a solution which would record only the actual changes would be more suitable.

6 DISCUSSION AND CONCLUSIONS

The research questions of this work was “How to implement audit trail in the large ERP system which has grown naturally over the time and what factors need to be considered before, during and after the implementation in this scenario?”. The example process of adding auditing functionality to existing ERP system is presented in the case study part of this work. The considerations discovered during the case study are presented in table 11. Before the implementation it is first of all important to gather the requirements for the auditing functionality. The requirements can be specific to auditing functionality itself, for example if it is necessary to log the same information as in the case study: “who”, “what”, “when”, or if it is necessary to include additional information for example “why” and “where” (Flores & Jhumka, 2017).

Table 11. Factors to consider when adding auditing functionality to existing ERP system

Phase relative to implementation	Considerations
Before	● Gather functional and nonfunctional requirements amongst stakeholders ● If possible plan the whole system with auditing in mind ● Consider demand and computational costs
During	● Should the format of the audit trail be static or dynamic ● Architecture of the system ● Maintenance of the history data ● Knowledge distribution
After	● Plan for deployment ● Monitor the functionality

In addition to these the system specific requirements like usability and performance need to be considered as well. If the audited system is small with low usage rates it is not as important to focus on the performance of the auditing. However if the system is likely to grow then special considerations should be given to both performance and design of the functionality so that it can later support the grown system. If possible the auditing

functionality should be designed and implemented as early as possible in the life cycle of system (Flores & Jhumka, 2017). This is evident in the case study as well because the major challenges were not about the auditing functionality itself. Rather they were about how to get the functionality to work with the existing system, for example implementing reliable recording of the user information and altering the incompatible database constraints. If the auditing had been considered from the start these issues could have been avoided. The demand for the audit trail must also be considered because it sets the limits on the price of the functionality, for example how much additional storage space can be used and how large the performance impact can be.

In the implementation phase several technical decision have to be made based on the previously defined requirements. Most importantly the format of the audit trail must be decided. The two main approaches are saving the full state of the data entry whenever it is changed and saving only the changed parts of the entry. For clarity the first method is referenced as static method and the second as dynamic method for the rest of this chapter. The names come from the fact that with static method the state of the data in the past can be generated by simply reading it from single history record, whereas with dynamic method the state has to be dynamically recreated by reading multiple history entries. The static method is used in the proposed solution with the temporal tables. The dynamic method was used by the existing auditing functionality. The main difference between the methods is that the static one generates more data, in terms of storage space, and the dynamic one generates more history data entries because each changing column is recorded to separate entry. However the actual storage space requirement is smaller because no redundant unchanged data is saved. It is also possible to combine the two methods by storing all the changes to single entry. This requires developing an additional structure for storing the changes which needs to be planned with care (Bishop, 1995). But it also combines the best features of the original methods, recording only the changed data with entry number equal to the static method. It is more efficient to get the past state of the data with the static method because the complete states are stored for each moment in time. With the dynamic method this requires reconstructing the state by iterating over several

changes. The choice between the methods boils down to the trade off between storage and processing time. The static method uses more storage while the dynamic method uses more processing power for maintaining and querying the higher number of entries.

Other considerations during the implementation are the architecture of the system, the maintenance of the history data and knowledge distribution. The architecture needs to be considered so that all the places where the audited data can be altered will include the auditing functionality. For database driven systems the most natural place for auditing is the database: the audit trail is completed when all the desired database operations are logged. However gathering the metadata for the log, for example user information can be more challenging, which was the case with the proposed proof of concept. In these kinds of situations the knowledge about the architecture is important as well. Maintenance of history data needs to be considered so that expired data can be deleted and storage space freed. Considerations must be done for two reasons. First of all simply deletion of the expired entries might not be possible, which is the case with temporal tables. Secondly even if the deletion is possible it might be computationally expensive operation especially in larger systems. In the proposed solution both of these concerns were addressed with the sliding window partitioning in history tables. Last important step during implementation is to share and receive information and feedback about the new functionality. Since the auditing is a feature which is likely to be deployed system wide it is important to gather feedback from other developers and stakeholders so that the final solution will be as efficient and easy to understand as possible. Furthermore sharing information about the new functionality is important so that once it is deployed others can start to utilize it as easily as possible.

Lastly once the implementation is completed it is necessary to plan for deployment of the new feature. Since the feature is likely to be system wide and it can potentially generate large amounts of history data once it is activated it is preferable to do the deployment in smaller sets rather than all at once. This can be done by utilizing proof of concept like was done in the case study of this work. Even then the deployment process must be planned carefully because the auditing functionality might have internal dependencies meaning that some parts of it must be deployed before others. For example in the proof of concept the

recording of user information must be deployed before its existence can be enforced. After deployment the functionality of new audit module must be monitored. This way the actual usage rates can be verified to be within the values which the auditing functionality was tested against. If usage rates, in practice the update frequency of data, are greater than expected then redesigning of the system might be required.

6.1 Limitations and future work

The greatest limitations regarding the results of this work was the exclusion of final deployment of the system from the scope of this work. This posed limitations to the evaluation of overall suitability of the proposed solution. The overall suitability was only evaluated based on the test scenarios rather than in functionality in production environment. The tests were planned to be as close to the real environment as possible, but since there is some uncertainty about the actual update frequency of data, final verification of the proposed solution could not be done until it is deployed. This limitation leads the way for the immediate future work. After deploying and monitoring the proof of concept, its functionality can be verified and the auditing can be extended to other parts of the system. Then user interface could be developed to get the full benefits of the auditing.

Additional limitation is that most of the conclusions on this work are drawn from the single case study and there is little reflection to the previous scientific research. However this is due to the apparent lack of previous study about this subject. This leads to the notion that research wise it would be beneficial to do further study about audit trail implementation methods. The systematic literature review of this work suggests that there is a lack of research about audit trail implementation methods from the software development perspective. Audit trail itself is widely researched concept, but most of the research is done from the perspective of accounting or organizational behavior. Even outside of the scientific literature there does not seem to be many thoroughly considered industry best practices about auditing. There are a few somewhat commonly used technologies, such as database triggers, but no broader frameworks which would address such real world challenges as large systems, maintainability and adding auditing to already mature

systems. For this reason the research should first focus on identifying auditing solutions and then comparing them so that framework for selecting suitable methods for varying real life scenarios could be generated.

REFERENCES

Albon, K. I., Davis, D., & Brooks, J. K. (2015). A Risk-Based Approach to Data Integrity. *Pharmaceutical Technology Europe; Cleveland*, *27*(7), 26–29. Retrieved from https://search-proquest-com.ezproxy.cc.lut.fi/docview/1707515184/abstract/D559E9F7C1E64186PQ/1

Alles, M. G., Kogan, A., & Vasarhelyi, M. A. (2008). Putting Continuous Auditing Theory into Practice: Lessons from Two Pilot Implementations. *Journal of Information Systems*, *22*(2), 195–214. https://doi.org/10.2308/jis.2008.22.2.195

Anonymous. (2011). 3 Steps to Simplify Audits, Demonstrate Compliance and Manage Risk Across the Enterprise. *Database Trends and Applications; Chatham*, *25*(4), 16. Retrieved from https://search-proquest-com.ezproxy.cc.lut.fi/docview/911809200/?pq-origsite=primo

Beland, K., Larson, K., Rowley, T., Mueller, M., Smith, C., Rizzo, A., … Rendell, M. (2014). Security and Audit Trail Capabilities of a Facilitated Interface Used to Populate a Database System with Text and Graphical Data Using Widely Available Software. *Journal of Software Engineering and Applications*, *7*(8), 713. Retrieved from https://search-proquest-com.ezproxy.cc.lut.fi/docview/1567063675/

Bishop, M. (1995). A standard audit trail format. *National Information Systems*, *18*. Retrieved from http://www.dtic.mil/get-tr-doc/pdf?AD=ADA302547#page=155

Bizarro, P. A., & Garcia, A. (2011). Using XBRL Global Ledger to Enhance the Audit Trail and Internal Control - ProQuest. *The CPA Journal*, *81*(5), 64–71. Retrieved from https://search-proquest-com.ezproxy.cc.lut.fi/docview/875640416/

Chang, S.-I., Yen, D. C., Chang, I.-C., & Jan, D. (2014). Internal control framework for a

compliant ERP system. *Information & Management*, *51*(2), 187–205. https://doi.org/10.1016/j.im.2013.11.002

Cruz-Correia, R., Boldt, I., Lapão, L., Santos-Pereira, C., Rodrigues, P. P., Ferreira, A. M., & Freitas, A. (2013). Analysis of the quality of hospital information systems audit trails. *BMC Medical Informatics and Decision Making*, *13*, 84. https://doi.org/10.1186/1472-6947-13-84

Debreceny, R. S., Gray, G. L., Ng, J. J., Lee, K. S., & Yau, W. (2005). Embedded Audit Modules in Enterprise Resource Planning Systems: Implementation and Functionality. *Journal of Information Systems*, *19*(2), 7–27. https://doi.org/10.2308/jis.2005.19.2.7

Flores, D. A., & Jhumka, A. (2017). Implementing Chain of Custody Requirements in Database Audit Records for Forensic Purposes. In *2017 IEEE Trustcom/BigDataSE/ICESS*. https://doi.org/10.1109/trustcom/bigdatase/icess.2017.299

Kulkarni, K., & Michels, J.-E. (2012). Temporal features in SQL:2011. *ACM SIGMOD Record*, *41*(3), 34. https://doi.org/10.1145/2380776.2380786

Li, S.-H., Huang, S.-M., & Lin, Y.-C. G. (2007). Developing a Continuous Auditing Assistance System Based on Information Process Models. *The Journal of Computer Information Systems; Stillwater*, *48*(1), 2–13. Retrieved from http://search.proquest.com.ezproxy.cc.lut.fi/docview/232574185/abstract/EEC9818776A44221PQ/1

Mullins, C. S. (2011). Compliance and Data Access Tracking. *Database Trends and Applications; Chatham*, *25*(4), 27. Retrieved from https://search-proquest-com.ezproxy.cc.lut.fi/docview/911809182/?pq-origsite=primo

Patel, A., Qi, W., & Wills, C. (2010). A review and future research directions of secure and trustworthy mobile agent-based e-marketplace systems. Information Management &

Computer Security, *18*(3), 144–161. https://doi.org/10.1108/09685221011064681

Prakash, B., & Nalini, N. (2014). *Improved Security of Audit Trail Logs in Multi-Tenant Cloud Using ABE Schemes* (Vol. 5).

Rezaee, Z., Sharbatoghlie, A., Elam, R., & McMickle, P. L. (2002). Continuous Auditing: Building Automated Auditing Capability. *AUDITING: A Journal of Practice & Theory*, *21*(1), 147–163. https://doi.org/10.2308/aud.2002.21.1.147

Roratto, R., & Dias, E. D. (2014). Security information in production and operations: a study on audit trails in database systems. *Journal of Information Systems and Technology Management*, *11*(3), 717–734. Retrieved from http://www.jistem.fea.usp.br/index.php/jistem/article/view/10.4301%252FS1807-17752014000300010

Shin, I., Lee, M., & Park, W. (2013). Implementation of the continuous auditing system in the ERP-based environment. *Managerial Auditing Journal*, *28*(7), 592–627. https://doi.org/10.1108/maj-11-2012-0775

Simeunovic, N., Grubor, G., & Ristic, N. (2016). Forensic accounting in the fraud auditing case. The European Journal of Applied Economics, *13*(2), 45–56. https://doi.org/10.5937/ejae13-10509

Singh, K., Best, P. J., Bojilov, M., & Blunt, C. (2014). Continuous Auditing and Continuous Monitoring in ERP Environments: Case Studies of Application Implementations. *Journal of Information Systems*, *28*(1), 287–310. https://doi.org/10.2308/isys-50679

Singh, K., Best, P., & Mula, J. (2013). Automating Vendor Fraud Detection in Enterprise Systems. *Journal of Digital Forensics, Security and Law*, *8*(2), 7–40. Retrieved from http://ojs.jdfsl.org/index.php/jdfsl/article/view/87

White, D. E., Oelke, N. D., & Friesen, S. (2012). Management of a large qualitative data

set: Establishing trustworthiness of the data. *International Journal of Qualitative Methods*, *11*(3), 244–258. Retrieved from http://journals.sagepub.com/doi/abs/10.1177/160940691201100305

APPENDIX 1. SYSTEMATIC LITERATURE REVIEW DATABASES

ACM - Association for Computing Machinery.
ACS Publications.
Aerospace Database (ProQuest - CSA).
Alma Talent Fokus.
Aluminium Industry Abstracts (ProQuest - CSA).
Amadeus - Analyse major databases from European sources.
ANTE: Abstracts in New Technologies and Engineering (CSA) (ProQuest XML).
ARTO - kotimainen artikkeliviitetietokanta.
ASFA Marine Biotechnology Abstracts (CSA - ProQuest).
Biotechnology & BioEngineering Abstracts (ProQuest - CSA).
Biotechnology Research Abstracts (ProQuest - CSA).
Cambridge Structural Databse CSD.
Ceramic Abstracts (ProQuest - CSA).
Chemical Abstracts SciFinder Scholar.
Chemoreception Abstracts (ProQuest - CSA).
Civil Engineering Abstracts (ProQuest - CSA).
Computer and Information Systems Abstracts (CSA) (ProQuest XML).
Conference Papers Index (CSA) (ProQuest XML).
Copper Technical Reference Library (ProQuest - CSA).
CORDIS Community Research and Development Information Service.
Corrosion Abstracts (ProQuest - CSA).
CRC Handbook of Chemistry and Physics.
Dawsonera.
Directory of Open Access Books (DOAB).
DOAJ Directory of Open Access Journals.
E-PRTR - The European Pollutant Release and Transfer Register.
Earthquake Engineering Abstracts (ProQuest - CSA).
Ebook Library (EBL).
EBSCO - Academic Search Elite.
EBSCO - Business Source Complete.
Edilex.
Electronics and Communications Abstracts (ProQuest - CSA).
Ellibs Library.
eMagz - sähköinen lehtikirjasto.
Emerald Journals.
Engineered Materials Abstracts (ProQuest - CSA).
Engineering Research Database (CSA) (ProQuest XML).
Environmental Engineering Abstracts (ProQuest - CSA).
ePress - kotimaisia sanomalehtiä.
EPRI - Electric Power Research Institute.
Espacenet (European Patent Office).
Facts databases on chemicals / Kemikaalien faktatietokannat.
FINLEX - ajantasainen lainsäädäntö.
FreePatentsOnline.
GreenFILE (EBSCO).
IEEE Xplore Digital Library.
INIS - International Nuclear Information System (IAEA).
IOPScience - Institute of Physics.
Journal and High Cited data JHCD (JCR+ESI).
JournalTOCs.
JSTOR Arts & Sciences I.
Kauppakamari Ammattikirjasto.
Kauppalehti Online.
KH Net -kiinteistötietopalvelu (KH-kortisto).
Kielitoimiston sanakirja.
KnowPap - Paperitekniikan ja tehtaan automaation oppimisjärjestelmä.
KnowPulp - Sellutekniikan ja tehtaan automaation oppimisjärjestelmä.
LUTPub / Doria.
LVI Net -tietopalvelu (LVI-kortisto).
MarketLine Advantage.
Materials Business File (ProQuest - CSA).
Materials Research Database with METADEX (CSA) (ProQuest XML).
Mechanical & Transportation Engineering Abstracts (ProQuest-CSA).
Melinda - kirjastojen yhteisluettelo.
METADEX (ProQuest - CSA).
MOT Dictionaries.
Nature weekly journal.
OAPEN.
OECD iLibrary.
OpenDOAR - The Directory of Open Access Repositories.
Oxford English Dictionary.
Palgrave encyclopedia of strategic management.
PROLA - American Physical Society.
ProQuest Central (ProQuest XML).
PSK Standardit.
PubMed. RefWorks (New).
RefWorks (Old).
Royal Society of Chemistry Journals.
RT Net (RT-kortisto).
Rusgate.
ScienceDirect - All Subscribed Content (Elsevier API).
SCOPUS (Elsevier).
SFS-standardit.
SFSedu- standardisoinnin oppilaitosportaali.
Social Science Research Network (SSRN) - eLibrary.
Solid State and Superconductivity Abstracts (ProQuest - CSA).
Springer eBooks.
Springer LINK.
SpringerLink eBooks - Business.
SpringerLink eBooks - Chemistry.
SpringerLink eBooks - Computer Science.
SpringerLink eBooks - Computing.
SpringerLink eBooks - Energy.
SpringerLink eBooks - Engineering.
SpringerLink eBooks - Environment.
SpringerLink eBooks - Mathematics.
SpringerLink Journals.
ST-Akatemia Online -palvelu.
Suomalais-Venäläinen kauppakamari.
Talentum lehtiarkisto.
Taylor & Francis Online Journal Library.
TEPA-termipankki.
Terveyskirjasto.
Thomson One Banker.
Ulrichsweb - Global Serials Directory.
Web of Science (WOS).
Wiley Online Library.

APPENDIX 2. RESULTS OF STORAGE SPACE USAGE TESTS

Before operation										After operation					
Number of previous updates	Number of rows	Size of main table (kb)	Size main/ row (kb)	Number of history rows	Size of history table (kb)	Size history/ row (kb)	Operation	System Versioning	Columnstore index	Number of rows	Size of main table (kb)	**Size main/ row (kb)**	Number of history rows	Size of history table (kb)	**Size history/ row (kb)**
0	136787	46112	0.337	0	0	0	Update every column with randomized data.	OFF	OFF	136787	150304	**1.099**	0	0	**0**
1	136787	150304	1.099	0	0	0	Update every column with randomized data.	OFF	OFF	136787	150304	**1.099**	0	0	**0**
0	136787	78032	0.57	0	0	0	Update every column with randomized data.	ON	OFF	136787	161432	**1.18**	136787	42600	**0.311**
1	136787	161432	1.18	136787	42600	0.311	Update every column with randomized data.	ON	OFF	136787	161432	**1.18**	273574	142080	**0.519**
38	136787	161432	1.18	5197906	3723416	0.716	Update every column with randomized data.	ON	OFF	136787	161432	**1.18**	5334693	3822896	**0.717**
0	136787	78032	0.57	0	16	0	Update every column with randomized data.	ON	ON	136787	161440	**1.18**	136787	2688	**0.02**
1	136787	161440	1.18	136787	2688	0.02	Update every column with randomized data.	ON	ON	136787	161440	**1.18**	273574	52032	**0.19**
38	136787	161440	1.18	5197906	1839080	0.354	Update every column with randomized data.	ON	ON	136787	161440	**1.18**	5334693	1888720	**0.354**
0	136787	46112	0.337	0	0	0	Update single column with randomized data.	OFF	OFF	136787	78144	**0.571**	0	0	**0**
1	136787	46112	0.337	0	0	0	Update single column with randomized data.	OFF	OFF	136787	78144	**0.571**	0	0	**0**
0	136787	78032	0.57	0	0	0	Update single column with randomized data.	ON	OFF	136787	78280	**0.572**	136787	42600	**0.311**
1	136787	78280	0.572	136787	42600	0.311	Update single column with randomized data.	ON	OFF	136787	78280	**0.572**	273574	89656	**0.328**
38	136787	78280	0.572	5197906	1783672	0.343	Update single column with randomized data.	ON	OFF	136787	78280	**0.572**	5334693	1830728	**0.343**
0	136787	78032	0.57	0	16	0	Update single column with randomized data.	ON	ON	136787	78280	**0.572**	136787	2696	**0.02**
1	136787	78280	0.572	136787	2696	0.02	Update single column with randomized data.	ON	ON	136787	78280	**0.572**	273574	9152	**0.033**
38	136787	78280	0.572	5197906	246512	0.047	Update single column with randomized data.	ON	ON	136787	78280	**0.572**	5334693	253184	**0.047**

APPENDIX 3. RESULTS OF EXECUTION TIME TESTS

Row	N° of rows in actual table	N° of previous updates	N° of updated columns	Trigger	Temporal	Columnstore	CPU-time (s)	CPU-time (s)	CPU-time (s)	CPU-time (s)	AVG (s)	STDEV (s)
1	36239	0	1	OFF	OFF	OFF	2.078	2.453	2.219	2.391	2.285	0.170
2	36239	0	3	OFF	OFF	OFF	2.375	2.454	2.297	2.500	2.407	0.089
1 - 2											**2.346**	**0.141**
3	36239	0	1	OFF	ON	OFF	2.688	2.578	2.094	2.609	2.492	0.270
4	36239	5	1	OFF	ON	OFF	2.281	2.469	2.719	2.422	2.473	0.183
5	36239	15	1	OFF	ON	OFF	2.313	2.500	2.140	2.234	2.297	0.153
3 - 5											**2.421**	**0.209**
6	36239	0	3	OFF	ON	OFF	2.297	2.922	2.344	2.172	2.434	0.333
7	36239	5	3	OFF	ON	OFF	2.281	2.250	2.453	3.063	2.512	0.378
8	36239	15	3	OFF	ON	OFF	2.359	2.203	2.156	2.687	2.351	0.240
6 - 8											**2.432**	**0.300**
3 - 8											**2.426**	**0.253**
9	36239	0	1	OFF	ON	ON	2.625	2.672	2.546	2.563	2.602	0.058
10	36239	5	1	OFF	ON	ON	2.797	2.469	3.046	2.766	2.770	0.236
11	36239	15	1	OFF	ON	ON	3.031	2.594	2.422	2.312	2.590	0.316
9 - 11											**2.654**	**0.225**
12	36239	0	3	OFF	ON	ON	2.688	2.828	2.813	2.657	2.747	0.087
13	36239	5	3	OFF	ON	ON	2.765	2.656	2.516	2.719	2.664	0.108
14	36239	15	3	OFF	ON	ON	2.782	2.547	2.516	2.890	2.684	0.182
12 - 14											**2.698**	**0.125**
9 - 14											**2.676**	**0.180**
15	36239	0	1	ON	OFF	OFF	3.047	2.984	2.657	2.906	2.899	0.171
16	36239	5	1	ON	OFF	OFF	2.750	2.875	2.938	2.797	2.840	0.083
17	36239	15	1	ON	OFF	OFF	2.719	2.765	2.797	3.204	2.871	0.224
15 - 17											**2.870**	**0.156**
18	36239	0	3	ON	OFF	OFF	3.813	3.922	3.938	3.734	3.852	0.096
19	36239	5	3	ON	OFF	OFF	3.703	3.625	4.047	3.891	3.817	0.190
20	36239	15	3	ON	OFF	OFF	3.782	3.875	3.797	3.672	3.782	0.084
18 - 20											**3.817**	**0.123**
15 - 20											**3.343**	**0.503**