St. Pölten University of Applied Sciences

/ informatik & security

/fh/// st.pölten

Datenbank Forensik in NoSQL

Audit-Trail für Dokumentenorientierte NoSQL Datenbanken

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

eingereicht von

Gruber Florian, BSc

is201803

im Rahmen des

Studienganges Information Security an der Fachhochschule St. Pölten

Betreuung

Betreuer/in: FH-Prof. Dipl.-Ing. Peter Kieseberg

Mitwirkung: -

St. Pölten, 24. August 2023

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Fachhochschule St. Pölten GmbH, Campus-Platz 1, A-3100 St. Pölten,T: +43 (2742) 313 228, F: +43 (2742) 313 228-339, E: office@fhstp.ac.at, I: www.fhstp.ac.at

Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Arbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich sonst keiner unerlaubten Hilfe bedient habe.
- ich das Thema dieser Arbeit bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.
- diese Arbeit mit der vom Begutachter/von der Begutachterin beurteilten Arbeit übereinstimmt.

Der Studierende/Absolvent räumt der FH St. Pölten das Recht ein, diese Arbeit für Lehre- und Forschungstätigkeiten zu verwenden und damit zu werben (z. B. bei der Projektevernissage, in Publikationen, auf der Homepage), wobei der Absolvent als Urheber zu nennen ist. Jegliche kommerzielle Verwertung/Nutzung bedarf einer weiteren Vereinbarung zwischen dem Studierenden/Absolventen und der FH St. Pölten.

Ort, Datum *Unterschrift*

Kurzfassung

Datenbanken haben in der heutigen Zeit einen großen Stellenwert, da diese in vielen Applikationen verwendet werden. Die Bereiche in denen solche Applikationen verwendet werden, erstrecken sich von einfachen Webshops bis hin zu ERP-Systemen (Enterprise Resource Planning), welche in weltweiten Konzernen eingesetzt werden.
Umso wichtiger ist es, dass sich die digitale Forensik mit diesem Thema auseinander setzt. Die Datenbank-Forensik ist jener Teilbereich der digitalen Forensik, der sich mit den in Datenbanken gefundenen Informationen befasst. [1]

Das Ziel dieser Diplomarbeit ist, eine Übersicht über den aktuellen Stand der Technik der Datenbank-Forensik im Bereich RDBMS (Relational Database Management System) und NoSQL (Not only Structured Query Language) zu schaffen. Des Weiteren soll eine aufgezeigte Technik aus dem Bereich RDBMS adaptiert und im Bereich NoSQL seine Anwendung finden.
Dazu wurden die folgenden Forschungsfragen gestellt:

- Welche Techniken im RDBMS-Bereich werden als Stand der Technik bezeichnet?
- Welche Techniken im NoSQL-Bereich werden als Stand der Technik bezeichnet?
- Welche Techniken aus dem Bereich RDBMS eignen sich dazu, um für den Bereich NoSQL adaptiert zu werden, unter der Berücksichtigung, dass diese bisher noch keine Anwendung in diesem Bereich gefunden haben?
- Mit welchen Mitteln, kann diese Technik für den NoSQL Bereich adaptiert und in welchem Umfang kann diese bereit gestellt werden?

Hierzu wurde ein Prototyp geplant, entwickelt und evaluiert. Das Ergebnis der Evaluierung ergab, dass die Audit-Trail Hinzufügung, Änderung und Löschung von Dokumenten erfolgreich detektieren und verifizieren kann. Eine Audit-Trail soll alle Änderungen in einer Datenbank protokollieren, sodass zu einem späteren Zeitpunkt ein zeitlicher Ablauf über alle Änderungen abgerufen werden kann und Manipulationen aufgedeckt werden können.

Abstract

Databases are of great importance in today´s world, as they are used in many applications. The areas in which such applications are used range from simple web shops to ERP (Enterprise Resource Planning) systems that are used in global corporations.
This makes it all more important for digital forensics to deal with this topic. Database forensics is that branch of digital forensics that deals with information found in databases. [1]

The aim of this thesis is to provide an overview of the current state of the art in database forensics in the field of RDBMS (Relational Database Management System) and NoSQL (Not only Structured Query Language). Furthermore, a demonstrated technique from the area of RDBMS is to be adapted and applied in the area of NoSQL.
To this end, the following research questions were posed:

- Which techniques in the RDBMS area are considered state of the art?
- Which techniques in the NoSQL area are considered state of the art?
- Which techniques from the RDBMS area are suitable to be adapted for the NoSQL area, taking into account that they have not yet been applied in this area?
- By what means can this technology be adapted for the NoSQL area and to what extent can it be made available?

For this purpose, a prototype was planned, developed and evaluated. The result of the evaluation showed that the audit trail can successfully detect and verify the addition, modification and deletion of documents. An audit trail should log all changes in a database so that a chronological sequence of all changes can be retrieved at a later date and manipulations can be detected.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Probleme 2
1.2 Ziel der Arbeit 2
1.3 Forschungsfrage 2
1.4 Struktur der Arbeit 3

2 Grundlagen **5**
2.1 Digitale Forensik 5
2.1.1 Datenbank-Forensik 7
2.2 RDBMS | Relational Database Management System 9
2.2.1 SQL 9
2.3 NoSQL | Not only SQL 11
2.3.1 NoSQL Typen 11
2.4 ACID & BASE 17

3 Stand der Technik **19**
3.1 Bereich SQL 19
3.1.1 B-Bäume & B+-Bäume 20
3.1.2 InnoDB Database Forensics: Enhaunced Reconstruction of Data Manipulation Queries from Redo Logs 23
3.1.3 Efficient model for detection data and data scheme tempering with purpose of valid forensic analysis 24
3.1.4 Towards a forensic-aware database solution: Using a secured database replication protocol and transaction management for digital investigations 25
3.2 Bereich NoSQL 27
3.2.1 Top 5 Considerations When Evaluating NoSQL Databases 27

3.2.2 Forensic investigation framework for the document store NoSQL DBMS: MongoDB as a case study . . . 29
3.2.3 Forensic attribution in NoSQL databases . . . 30
3.2.4 A method and tool to recover data deleted from a MongoDB . . . 32
3.2.5 A data recovery technique for Redis using internal dictionary structure . . . 34
3.3 Resultat . . . 34

4 Herangehensweise . . . 35
4.1 Methodisches Vorgehen . . . 35
4.2 MongoDB . . . 36
4.2.1 Allgemeines über MongoDB . . . 36
4.2.2 Unterschiedliche MongoDB Bereitstellungsarten . . . 37
4.2.3 OpLog . . . 38
4.3 Resultat . . . 46

5 Implementierung & Evaluierung . . . 47
5.1 Planung des Prototyps . . . 47
5.1.1 Allgemeiner Programmablauf . . . 48
5.1.2 Anbindung diverser Dokument NoSQL Datenbank-Systeme . . . 50
5.2 Entwicklung des Prototyps . . . 51
5.2.1 Basis Daemon . . . 51
5.2.2 Daemon für MongoDB Instanzen . . . 54
5.2.3 API-Endpoint . . . 57
5.3 Evaluierung . . . 75
5.3.1 On-Premise & SaaS Umgebung . . . 75
5.3.2 Beispiel Audit-Trail | Interaktion mit der MongoDB . . . 76
5.3.3 Beispiel Audit-Trail | Verifizierung . . . 96
5.4 Resümee der Evaluierung . . . 103

6 Conclusio und weiterführende Arbeiten . . . 105
6.1 Conclusio . . . 105
6.2 Weiterführende Arbeiten . . . 106

Abbildungsverzeichnis . . . 107

Tabellenverzeichnis 108

Acronyms 113

Literatur 115

1 Einleitung

Datenbanken haben in der heutigen Zeit einen großen Stellenwert, da sie in vielen Applikationen verwendet werden. Die Bereiche, in denen solche Applikationen verwendet werden, erstrecken sich von einfachen Webshops bis hin zu ERP-Systemen (Enterprise Resource Planning), die in weltweiten Konzernen eingesetzt werden.
Umso wichtiger ist es, dass sich die digitale Forensik mit diesem Thema auseinander setzt. Die Datenbank-Forensik ist jener Teilbereich der digitalen Forensik, der sich mit den in Datenbanken gefundenen Informationen befasst. [1] Hierbei wird der Inhalt, wie die Protokolldateien, die Metadaten und die Daten an sich selbst, je nach Art der verwendeten Datenbank, analysiert und untersucht.
Bei der Datenbank-Forensik geht es darum, die Fragen zu beantworten, wann, warum, wo, wie und von wem eine Manipulation von Daten in der Datenbank stattgefunden hat. [2]

Das lange Bestehen von relationalen Datenbank-Verwaltungssystemen (Relational Database Management System, RDBMS) hat dazu geführt, dass es in diesem Bereich viele Ansätze und Vorgehensweisen für Datenbank-Forensik gibt.
Die heute weit verbreiteten NoSQL (Not only SQL) Datenbank-Systeme sollen die Nachteile von RDBMS adressieren. Diese Datenbank-Systeme wurden bisher noch nicht stark im Bezug auf Datenbank-Forensik unter die Lupe genommen.

1.1 Probleme

Eines der Probleme ist, dass sich Forschungsarbeiten im Bereich NoSQL auf die Leistungsvergleiche, Unterschiede zwischen RDBMS und NoSQL, besondere Merkmale der NoSQL und ihre Anwendungen in unterschiedlichen Bereichen beziehen. [3] Jedoch beleuchtet nur ein kleiner Teil der Arbeiten den Bereich der Datenbank-Forensik auf NoSQL Datenbank-Systemen.
Der Großteil der Forschungsarbeiten, der die Datenbank-Forensik im Bereich RDBMS beleuchtet, kann nicht direkt auf den Bereich NoSQL angewendet werden, da diese sehr spezifisch für den Bereich RDBMS sind. Dies liegt darin, dass sich die beiden Datenbank-Systeme stark von Grund auf unterscheiden.
Ein weiteres Problem in Bezug auf Datenbank-Forensik im NoSQL Bereich ist, dass die Datenmodelle nicht immer einem fixen Schema folgen müssen, sondern diese können dynamisch der entsprechenden Anwendung angepasst werden.

1.2 Ziel der Arbeit

Mit dieser Arbeit soll eine Übersicht über den aktuellen Stand der Technik in der Datenbank-Forensik im Bereich RDBMS und NoSQL geschaffen werden - eine aufgezeigte Methode aus dem Bereich RDBMS soll adaptiert werden und im Bereich NoSQL seine Anwendung finden.

1.3 Forschungsfrage

In dieser Diplomarbeit werden folgende Fragestellungen bezüglich Datenbank-Forensik auf NoSQL behandelt:

1. Welche Methoden im Bereich RDBMS werden als Stand der Technik bezeichnet?
2. Welche Methoden im Bereich NoSQL werden als Stand der Technik bezeichnet?
3. Welche Techniken aus dem Bereich RDBMS eignen sich dazu, um für den Bereich NoSQL adaptiert zu werden, unter der Berücksichtigung, dass diese bisher noch keine Anwendung in diesem Bereich gefunden haben?
4. Mit welchen Mitteln, kann diese Technik für den NoSQL Bereich adaptiert und in welchem Umfang kann diese bereit gestellt werden?

1.4 Struktur der Arbeit

Diese Diplomarbeit ist in mehrere Teile aufgegliedert. Kapitel 1 enthält eine Einführung in das Thema Datenbank-Forensik, die Herausforderungen und das Ziel der Arbeit ein. Kapitel 2 beschreibt die grundlegenden Kenntnisse von Datenbank-Forensik, RDBMS und NoSQL. Kapitel 3 gibt einen Überblick des aktuellen Standes der Technik. Kapitel 4 beschreibt die Herangehensweise, die geplante Durchführung der Arbeit und Allgemeines über MongoDB, worauf der Prototyp aufbaut. Kapitel 5 zeigt die technische Entwicklung und Implementierung des Prototyps der Arbeit auf. Kapitel 6 schließt mit einer Conclusio ab.

2 Grundlagen

In der heutigen Zeit laufen viele Prozesse digital ab oder werden durch digitale Systeme in ihrem Ablauf unterstützt. Durch diese Digitalisierung wurde es notwendig, digitale Informationen in Datenbanken abzuspeichern, sodass speziell benötigte Informationen zum Zeitpunkt der Anforderung schnell aus diesen Datenbanken geladen werden können und zum jeweiligen Prozessschritt zur Verfügung stehen. Diese Informationen sind nur so vertrauenswürdig, so sicher diese Informationen in das System gebracht werden und so vertrauenswürdig das Datenbank-System behandelt wird.

In den folgenden Abschnitten wird erklärt, was man unter Datenbank-Forensik versteht, es wird genauer auf die Bereiche RDBMS und NoSQL eingegangen und darüber hinaus werden die Grundlagen von ACID und BASE aufgezeigt.

2.1 Digitale Forensik

"Digitale Forensik ist die Anwendung von Methoden zur Identifizierung, Sammlung, Erhaltung, Validierung, Analyse, Interpretation, Dokumentation und Präsentation digitaler Beweise, die aus digitalen Quellen stammen, um die Rekonstruktion von Ereignissen zu erleichtern, die sich als unerwünscht herausgestellt haben und dazu beitragen können, zukünftige Vorfälle zu antizipieren oder zu verhindern."[4]

Die Digitale Forensik kann in mehrere Bereiche aufgeteilt werden.

Hierzu zählen folgende Bereiche:

- **Datenbank-Forensik**

  Zu diesem Bereich wird in Abschnitt 2.1.1 näher darauf eingegangen.
- **Computer-Forensik**

  Computer-Forensik beschäftigt sich mit Festplatten, Speichersticks, RAM (Random Access Memory), CPU Cache (Central Processing Unit) und Dateien auf Computern.
- **Netzwerk-Forensik**

  Die Netzwerk-Forensik beschäftigt sich mit dem Netzwerk-Verkehr, hierzu zählen LAN und WAN
- **Smartphone-Forensik**

  Der Bereich der Smartphone-Forensik beschäftigt sich mit Daten, die auf mobilen Endgeräten aufzufinden sind. Hierbei handelt es sich um Nachrichten/SMS, Lokalisierungs-Informationen, verwendete Apps, Kontakte und persönliche Informationen.
- **Cloud-Forensik**

  Cloud-Forensik beschäftigt sich mit der digitalen Forensik mit in der Cloud bezogenen Rechenressourcen, Speicherressourcen, ...
- **Code-Forensik**

  Die Code-Forensik beschäftigt sich mit Reverse Engineering von Anwendungs-Software, Malware, Viren, ...

2.1.1 Datenbank-Forensik

Die Datenbank-Forensik ist ein Unterbereich der digitalen Forensik, der sich mit der detaillierten Analyse einer Datenbank befasst, einschließlich ihres Inhalts, der Protokolldateien, der Metadaten und der Datendateien, je nachdem, welche Art von Datenbank verwendet wird. Die Datenbank-Forensik legt aktuell ihr Hauptaugenmerk auf die Prüfung der Datendateien und Protokolldateien auf Konsistenz. [5]
Die Datenbank-Forensik involviert die in der Abb. 2.1 dargestellten digitale forensische Prozess-Schritte.[2]

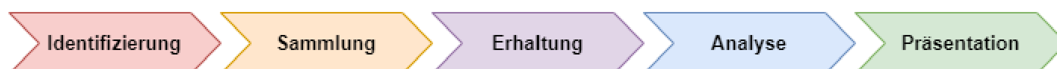

Abbildung 2.1: Datenbank-Forensik | Prozess

- **Identifizierung**
  Bei dem Schritt "Identifizierung" geht es darum, zu evaluieren was als Quellen für einen digitalen Beweis im Datenbank-Umfeld herangezogen werden kann. Hierzu können zum Beispiel Zugriff-Logs, Transaktions-Logs, Replikation-Logs, Daten-Dateien und je nach System weitere Dateien der Datenbank-Systeme als Quellen gelistet werden.
- **Sammlung**
  Im Schritt "Sammlung" geht es darum, wie auf diese Quellen zugegriffen werden kann.
- **Erhaltung**
  Bei "Erhaltung" dreht sich alles darum, in welchem Format die Informationen aus den davor angesprochenen Quellen erhalten werden können. Je nach System können manche Quellen in Klartext verwertet werden und für andere Quellen müssen eigene Parser verwendet werden, um die benötigten Informationen abgreifen zu können.
- **Analyse**
  Im Schritt "Analyse" geht es darum die gewonnen Informationen zu analysieren, um den Fall aufzuklären. Meistens wird eine Zeitleiste erstellt, sodass der Fall Schritt für Schritt nachvollzogen werden kann.
- **Präsentation**
  Bei dem Schritt "Präsentation" geht es darum, die gewonnen Informationen mithilfe der erstellten Zeitleiste so aufzubereiten, dass ersichtlich ist, was bei dem entsprechenden Fall vorgefallen ist. Als Endresultat wird ein Report erstellt, der alle Informationen aufgearbeitet zusammenfasst und dokumentiert, wie der Fall abgelaufen ist.

Bei der Datenbank-Forensik muss unterschieden werden, ob es sich um eine Datenbank im Bereich RDBMS oder NoSQL handelt. Des Weiterens fließt mit ein, ob es sich um eine Datenbank-Installation als ”SaaS” (Software as a Service) oder ”On-Premise” handelt. Je nachdem stehen nach der zu analysierenden Datenbank-Installation Protokolldateien und Systemdateien aufgrund des Abstraktionslevels nicht zur Verfügung. Durch das Fehlen dieser Dateien, die aufgrund des Abstraktionslevels nicht für forensische Zwecke herangezogen werden können, wird die forensische Untersuchung erschwert.

CountryCode	Name	Währung
AT	Austria	EUR
CZ	Czech Republic	CZK
CH	Switzerland	CHF

Tabelle 2.1: RDBMS-Tabelle [tblCountries], Länder inklusive Währung

2.2 RDBMS | Relational Database Management System

Relationale Datenbanken sind Datenbanken, die Daten mit definierten Beziehungen für einen schnellen Zugriff speichern und intern organisieren. Die Daten in einer relationalen Datenbank werden in Tabellen gespeichert. Der Aufbau der Entität, welche in der jeweiligen Tabelle gespeichert wird, muss im Vorfeld als Spalteninformation definiert werden.
Die Daten werden in den Zeilen der jeweiligen Tabelle gespeichert. Daraus resultiert eine Datenstruktur, die es erlaubt, dass die Informationen in der relationalen Datenbank effizient und flexibel aufgefunden werden können. Das Hauptaugenmerk liegt in der Beziehung zwischen den gespeicherten Informationen über die Tabellen hinweg.
Relationale Datenbanken sind so aufgebaut, dass sie mithilfe von SQL (Structured Query Language) angesprochen werden können. SQL wird im folgenden erklärt.

2.2.1 SQL

Durch den Einsatz von SQL, einer lesbaren Sprache für Interaktionen mit Datenbanken, können Informationen in die Datenbank hinzugefügt, abgerufen, bearbeitet und entfernt werden. Mit dieser Datenbanksprache kann ebenso die Struktur der Datenbank definiert und angepasst werden.
In Tabelle 2.1 ist eine Tabelle von Ländern mit ISO-Kürzel, Name und deren Währung angeführt. Mit dem SQL-Statement Listing 1 können die Informationen für das Land mit dem CountryCode "AT" abgerufen werden. Dieses SQL-Statement liefert nach dem Absetzen den Eintrag des Landes mit dem CountryCode "AT" als Ergebnis, welches in der Tabelle 2.2 dargestellt ist zurück.

```sql
1 SELECT * FROM tblCountries WHERE CountryCode='AT'
```

Listing 1: SQL-Statement um Informationen zu dem CountryCode "AT" abzurufen

CountryCode	Name	Währung
AT	Austria	EUR

Tabelle 2.2: SQL-Statement Ergebnis von Listing 1

DDL | Data Definition Language

Um Tabellen in einer relationalen Datenbank zu erstellen, wird die sogenannte DDL (Data Definition Language) verwendet. Im Listing 2 ist ein DDL-Statement zu sehen, welches die Tabelle ”tblCountries” mit den Spalteninformationen ”CountryCode”, ”Name” und ”Währung” erstellt.

```
1 CREATE TABLE tblCountries
2     ([CountryCode] varchar(2), [Name] varchar(250), [Währung]
  ↪  varchar(3))
3 ;
```

Listing 2: DDL-Statement um Tabelle ”tblCountries” zu erstellen

DML | Data Manipulation Language

Die DML (Data Manipulation Language) wird verwendet, um Daten in der Datenbank zu bearbeiten. Im Listing 3 ist ein DML-Statement zu sehen, wodurch ein Eintrag für ”Deutschland” mit den Werten ”DE” als ”CountryCode”, ”Germany” als ”Name” und ”EUR” als ”Währung” hinzugefügt wird.

```
1 INSERT INTO tblCountries
2     ([CountryCode], [Name], [Währung])
3 VALUES
4     ('DE', 'Germany', 'EUR')
5 ;
```

Listing 3: DML-Statement um Eintrag ”DE” in die Tabelle ”tblCountries” hinzuzufügen

2.3 NoSQL | Not only SQL

Bei ”NoSQL” wird von Datenbanken gesprochen, die flexibel und skalierbar sind sowie kein fixes Datenbankschema vorweisen. NoSQL bedeuted ”Not only SQL”. Sie bieten Speicher- und Abrufmechanismen mit weniger eingeschränkten Datenbankschemen als traditionelle relationale Datenbanken. [6]
Ihr Datenmodell ist dafür ausgelegt, dass große Mengen an sich schnell ändernden Daten verarbeitet werden können. Bei den Daten kann es sich um strukturierte, halbstrukturierte und unstrukturierte Daten handeln.

Gründe für das Aufkommen von NoSQL sind: [7]

- Um Multistrukturierte Datentypen zu verarbeiten, die nicht in die Datenstruktur von relationalen Datenbank passen.
- Sie sind praktikable Alternativen zu teuren proprietären relationalen Datenbanksystemen.
- Damit wird eine schnellere Anpassung am Markt ermöglicht, um sich der Agilität und Geschwindigkeit des Entwicklungszyklus anzupassen, sodass kurze agile Entwicklungszyklen angewendet werden können.

Im folgenden Kapitel wird auf die verfügbaren NoSQL Datenbank-Typen eingegangen.

2.3.1 NoSQL Typen

Bei NoSQL kann unter vier Datenbank-Typen unterschieden werden. Jeder Typ hat seine eigenen Vorteile bzw. Nachteile und seinen geeigneten Einsatzbereich.
Hierbei wird grundsätzlich in die folgenden vier Typen unterschieden:

- Dokumenten Datenbank
- Key-Value Datenbank
- Graph Datenbank
- Wide-Column Datenbank

In den folgenden Unterpunkten wird auf die unterschiedlichen Typen näher eingegangen.

Dokumenten Datenbank

Bei Dokumenten Datenbanken handelt es sich um Datenbanken, die als Datenmodell Dokumente verwalten. Die Struktur dieser Objekte ähnelt JSON (JavaScript Object Notation). Mithilfe dieser Struktur wird Entwicklern die Möglichkeit geboten, Daten einfach in der Datenbank zu speichern und abzurufen.
Bei den Feldern der Dokumente handelt es sich um typisierte Werte, wie zum Beispiel, Integer, Dezimal, String, Datum, Array und Binärdaten. Durch die Verwendung von JSON als Struktur der Dokumente, wird ermöglicht, dass die Dokumente jederzeit um weitere Felder erweitert werden können.
Ein großer Vorteil der Dokumenten Datenbanken ist, dass jedes Feld innerhalb eines Dokumentes abgefragt und aktualisiert werden kann. Die Entwickler-Bibliotheken der Dokumenten Datenbanken stellen die Funktion zur Verfügung, dass die entsprechenden Dokumente direkt als Objekte in der jeweiligen verwendeten Programmiersprache angesprochen werden können. Die Möglichkeit von eingebetteten Dokumenten und Arrays in den Dokumenten selbst erspart den Entwicklern die Verwendung von rechenintensiven Join-Operationen.

Die Hierarchie dieses Datenbank-Typs stellt sich wie folgt dar: Datenbank -> Sammlung (Collection) -> Dokument (Document)
In der Abb. 2.2 wird der Aufbau der Dokumenten Datenbank Schematisch dargestellt.

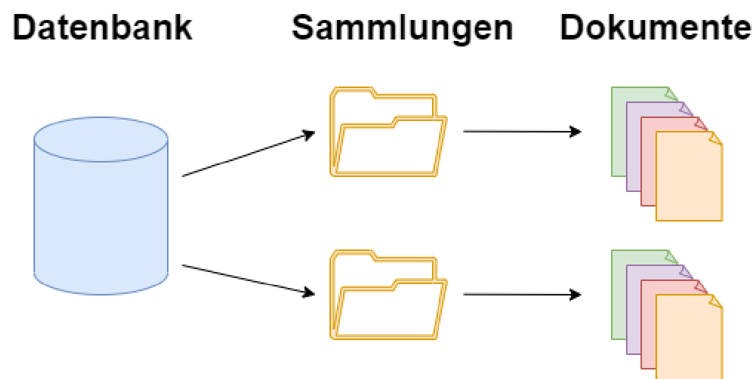

Abbildung 2.2: NoSQL | Schematische Darstellung der Dokumenten Datenbank

In Sammlungen werden nach "Best Practices" Dokumente desselben Typs verwaltet. In den meisten Dokumenten Datenbanken werden die Dokumente über eine Objekt-ID (`_id`) referenziert.
Die Datenbank CosmosDB bietet eine Schnittstelle zu SQL an, um die Einstiegshürde zu NoSQL-Datenbanken niedrig zu halten. MongoDB verwendet in diesem Fall eine eigens entwickelte Syntax. Auf diese Syntax wird in Kapitel 4 näher eingegangen.

Im Listing 4 ist ein Beispieldokument ersichtlich. In Kapitel 4 wird näher auf die Dokumenten Datenbanken eingegangen, im speziellen auf die MongoDB [8], von der gleichnamigen Firma MongoDB Inc.

```
1  {
2      "_id":"d4bf766d-501d-43ea-8b2f-6f9e328f3e58",
3      "first-name":"Max",
4      "last-name":"Mustermann",
5      "age":27,
6      "hobbies":[
7          "Lesen",
8          "Radfahren"
9      ]
10 }
```

Listing 4: NoSQL | Dokumenten Datenbank | Beispiel Dokument

Produkte: CosmosDB [9], MongoDB [8], CouchDB [10], Amazon DocumentDB [11], RavenDB [12]

Anwendungsbereich: Web Anwendungen, Content Management Systeme, Log Management

Key-Value Datenbank

Bei Key-Value Datenbanken handelt es sich um Datenbanken, die auf einer Tabelle mit nur zwei Spalten basieren. Eine Spalte beinhaltet den eindeutigen Schlüssel und in der zweiten Spalte wird der zu speichernde Wert verwaltet. Im Detail wird jeder Eintrag in der Datenbank mit einem Schlüssel und dem entsprechendem Wert gespeichert. Das Datenbanksystem arbeitet lediglich mit den eindeutigen Schlüsseln, die Werte der Einträge werden vom System nicht zur Indizierung herangezogen. Abfragen werden über die Schlüssel der Einträge gebildet. Es ist nicht möglich, dass nach einem speziellen Datenwert eines Eintrages gesucht werden kann.
Key-Value Datenbanken können nicht nur, wie bei Dokument Datenbanken, mit typisierten Werten arbeiten, sondern diese können zusätzlich mit Hashes, Listen und sortierte Sets umgehen. Der große Vorteil von Key-Value Datenbanken ist, dass sie eine hohe Performance bieten und eine leichte horizontale Skalierung ermöglichen.
In Abb. 2.3 ist ein Beispiel aufgezeigt, wie Daten in der Key-Value Datenbank gespeichert werden.

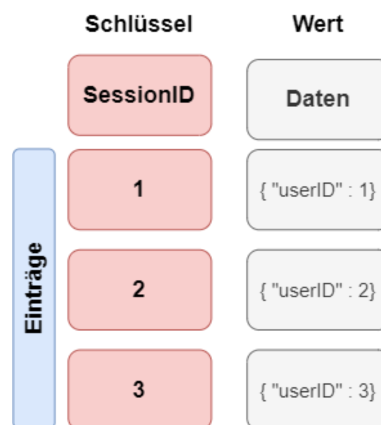

Abbildung 2.3: NoSQL | Key-Value DB | Sitzungsspeicher

Produkte: Redis [13], Amazon DynamoDB [14]

Anwendungsbereich: Sitzungs Daten, Nachrichten-Broker

Graph Datenbank

Graph Datenbanken werden auf Grundlagen von Graphen abgebildet. Graph Datenbanken sind Knoten zum Speichern von Datenentitäten und Kanten zum Speichern von Beziehungen zwischen Entitäten. Eine Kante besitzt immer einen Start- und einen Endknoten, einen Typen und eine Richtung kann die Beziehung zwischen Elementen, Aktionen und Besitzern beschreiben. [15]

Diese Datenbanken bieten umfangreiche Abfragemodelle, mit denen einfache und komplexe Beziehungen abgefragt werden können. Der Vorteil von Graphdatenbanken liegt darin, dass Joins und Beziehungen sehr schnell eruiert werden können, da Beziehungen zwischen den einzelnen Knoten nicht bei der Abfrage berechnet werden müssen, sondern in der Datenbank bestehen.

Graphdatenbanken werden nicht für herkömmliche Anwendungen verwendet. Deren Anwendung findet sich in jenem Bereich wieder, wo Beziehungen zwischen Entitäten schnell aufgefunden werden müssen. Zu diesen Bereichen zählen Soziale Netzwerke und benutzerdefinierte Onlinewerbung.

Produkte: Neo4j [16], Amazon Neptune [17]
Anwendungsbereich: Soziale Netzwerke, Onlinewerbung

Wide-Column Datenbank

Wide-Column Datenbanken zählen wie die Vorgänger zu den NoSQL-Datenbanken. Bei diesem Datenbank Typ ähnelt der Aufbau einer relationalen Datenbank, jedoch können sich in der selben Tabelle die Namen und Datentypen der jeweiligen Spalten von Zeile zu Zeile unterscheiden.
Diese Datenbanken bieten den Vorteil, dass sie horizontal sehr gut skalierbar sind, da die individuellen Spalten der einzelnen Zeilen über mehrere Server verteilt werden können.

Produkte: Cassandra [18], Google Bigtable [19], ScyllaDB [20]
Anwendungsbereich: Echtzeit-Anwendungen, IoT-Anwendungen

2.4 ACID & BASE

ACID & BASE sind Grundlagen an denen sich Datenbanksysteme orientieren, sodass diese reibungslos funktionieren. Der Hauptunterschied zwischen ACID & BASE liegt darin, dass ACID ein konsistentes System sicherstellt und BASE ein hoch verfügbares System bietet. In der Arbeit von Mohamed A. Mohamed [21] und Rasheed [6] wurde auf die Begriffe ACID und BASE im Detail eingegangen.
Zusammenfassend ACID stellt sicher, dass eine Datenbanktransaktion rechtzeitig abgeschlossen wird und sich das System zu jeder Zeit in einem sicheren Zustand befindet.
"**ACID**" ist eine Abkürzung für die folgenden Begriffe:

- **Atomicity**
  Wenn Teile der Transaktion nicht abgeschlossen werden, gilt die gesamte Transaktion als nicht erfolgreich.
- **Consistency**
  Die Consistency stellt sicher, dass sich die Datenbank vor und nach einer Transaktion in einem sicheren Zustand befindet.
- **Isolation**
  Sollten mehrere Transaktionen zeitgleich ablaufen, stellt die Isolierung sicher, dass sie sich nicht untereinander behindern und zu einem inkonsistenten Zustand führen. Dies setzt voraus, dass die Transaktionen seriell abgehandelt werden.
- **Durability**
  Die Durability stellt sicher, dass Änderungen, die durch autorisierte Transaktionen durchgeführt wurden permanent sind.

Im folgenden wird der Begriff "**BASE**" (Basically available, Soft state, Eventual consistency) genauer erklärt.
Das Ziel ist, dass die Konsistenz direkt nach einer Transaktion kein fester Zustand mehr ist. Die Konsistenz soll nicht sofort nach Abschluss der Transaktion erreicht werden, sondern erst nach einer gewissen Zeit. Bis ein fester Zustand erreicht wird, kann die Konsistenz in einem undefinierten Zustand sein. *The focus of BASE is the permanent availability. BASE is the opposite of ACID.* [21]

NoSQL Datenbanken sind Datenbank-Systeme, die sich zwischen ACID und BASE einfinden. [21]

3 Stand der Technik

In diesem Kapitel wird der Stand der Technik der Datenbank-Forensik im Bereich SQL und NoSQL aufgezeigt. Zuerst wird auf Verfahren des Bereichs SQL, die im Rahmen dieser Diplomarbeit evaluiert wurden, eingegangen. Anschließend werden bestehende Verfahren des Bereichs NoSQL, die im Rahmen der Diplomarbeit untersucht wurden, aufgezeigt. Abschließend wird ein Resultat über den Stand der Technik der analysierten Arbeiten verfasst.

3.1 Bereich SQL

Im Bereich SQL wurden die folgenden Verfahren und Arbeiten untersucht, ob diese Verfahren eine adaptierte Anwendung im Bereich NoSQL finden können.

- Verfahren, die sich mit ”B-Bäume” und ”B^+-Bäume” befassen von Kieseberg et al. [22][23][24] und Frühwirt et al. [25]
- Eine Methode, die sich mit ”Redo Logs” von InnoDB befasst von Frühwirt et al. [26]
- Ein Verfahren, das sich mit den ”Audit Logs” der Datenbanksysteme befasst, von Azemovic et al. [27]
- Die Arbeit ”Towards a Forensic-Aware Database Solution: Using a secured Database Replication Protocoll and Transaction Management for Digital Investigations” von Frühwirt et al. [28]

3.1.1 B-Bäume & B^+-Bäume

In diesem Kapitel wird auf B-Bäume und B^+-Bäume, welche Verwendung im Datenbanksystem InnoDB [29] finden, eingegangen. Als erstes werden allgemeine Informationen zu B-Bäume und B^+-Bäume aufgelistet. Anschließend wird näher auf die umgesetzten Ansätze eingegangen.

Allgemeine Informationen

Ein B^+-Baum ist ein ausgeglichener Baum, welcher folgende Eigenschaften besitzt: [23]

- Jeder Nicht-Wurzelknoten enthält zwischen $\frac{b}{2}$ und b Elemente
- Der Wurzelknoten enthält höchstens b Elemente
- Ein innerer Knoten mit y Elementen hat $y + 1$ Kindknoten
- Alle Blattknoten liegen auf derselben Ebene
- Alle Elemente innerhalb eines Blattes sind sortiert

Der Hauptunterschied zwischen B-Bäumen und B^+-Bäumen liegt darin, dass die Informationen der Daten in einem B^+-Baum nur in den Blattknoten gespeichert werden. In den inneren Knoten werden lediglich die Referenzen zu den Kindknoten und Blattknoten beherbergt. [22] Dadurch weisen B^+-Bäume eine höhere Performance als B-Bäume auf.
"A high branching level (i.e. a high number of child nodes) reduces the height of the tree and therefore the expensive read operations on nodes." [23]

Die drei nachfolgenden erwähnten Ansätze basieren auf folgendem Theorem: [23]
Ein B^+-Baum mit $n > b$ Elementen, die in aufsteigender Reihenfolge hinzugefügt werden. Dann gilt, dass die Partition der resultierenden k Blätter von dem B^+-Baum folgende Struktur aufweisen:

$$n = \sum_{i=1}^{k} a_i, \text{ mit } a_i = \frac{b}{2} + 1, \forall i \neq k \text{ und } a_k \geq \frac{b}{2}$$

Durch diese Struktur ist es für den Baum an sich selbst unmöglich, weitere Elemente als Blätter hinzuzufügen, außer ganz rechts in der Struktur. Der Baum wird mithilfe eines auto-increment Primary-Key gebildet. Auf den folgenden Seiten wird auf die drei Ansätze, die auf diesem Theorem [23] basieren, eingegangen.

Trees Cannot Lie: Using Data Structures for Forensics Purposes

In dem Artikel "Trees Cannot Lie: Using Data Structures for Forensics Purposes" von Kieseberg et al. [23] wurde gezeigt, wie detektiert werden kann, ob ein manipulierter Eintrag hinzugefügt oder ein Eintrag aus der Datenbank entfernt wurde.
Dadurch, dass der B^+-Baum aufgrund seiner sortiert hinzugefügten Datensätze gebildet wird, können nicht sortiert oder unsortiert hinzugefügte Einträge durch die Füllrate der einzelnen Blätter eruiert werden. Wenn Daten wie beschrieben sortiert hinzugefügt werden, dann weisen alle Blätter bis auf das rechts äußerste Blatt eine Füllrate von $\frac{b}{2} + 1$ auf. Das Hinzufügen eines manipulierten Datensatzes (gefälschter Zeitstempel) durch einen Angreifer, führt dazu, dass dieser Eintrag nicht dem rechts äußersten Blatt zugewiesen wird und somit der Füllgrad in einem anderen Blatt $> \frac{b}{2} + 1$ ist.
Damit kann festgestellt werden, in welchem Blatt sich ein manipulierter Datensatz befindet, jedoch nicht eruiert werden, welcher Eintrag genau.
In diesem Artikel wurden zwei Schwachstellen dieses Ansatzes aufgezeigt. Als erste Schwachstelle wurde angeführt, dass wenn genau $\frac{b}{2} + 1$ manipulierte Datensätze hinzugefügt werden, führt dies dazu, dass in allen, bis auf das rechts äußerste Blatt, die Füllrate $\frac{b}{2}+1$ beträgt. Dadurch weist der Baum keine Manipulation aufgrund des Füllgrades auf.
Als zweite Schwachstelle wurde genannt, dass wenn ein Datensatz aus einem Blatt gelöscht wird, fällt dessen Füllgrad auf $\frac{b}{2}$. Dies kann gegenüber den anderen Blättern wieder detektiert werden. Werden jedoch mindestens zwei Einträge gelöscht, so fällt der Füllgrad des Blattes unter $\frac{b}{2}$, was dazu führt, dass ein reorganisieren des Baumes stattfindet. Bei der Reorganisierung des Baumes werden Blätter zusammengeführt und weisen anschließend einen Füllgrad von $> \frac{b}{2} + 1$ auf.

Structural Limitations of B^+-Tree forensics

In der Arbeit "Structural Limitations of B^+-Tree forensics" von Kiesberg et al. [22] wurde dieser zuvor erwähnte Ansatz erneut untersucht und die Erkenntnis gezogen, dass der Ansatz zur Erkennung von Datenlöschung nicht verallgemeinert werden kann. Wo dieser Ansatz dennoch Verwendung finden kann ist, bei der Überprüfung von Audit-Tabellen, da bei diesen Tabellen eine strikte Insert-Only Policy besteht.

Using Internal MySQL/InnoDB B-Tree Index Navigation for Data Hiding Purposes

In dem Artikel ”Using Internal MySQL/InnoDB B-Tree Index Navigation for Data Hiding Purposes” von Frühwirt et al. [25] wird demonstriert, wie Datensätze in einer B-Baum Struktur versteckt werden können.

InnoDB erstellt zu jedem Primary-Key einer Tabelle einen Index und speichert die dazugehörigen Daten des Index direkt im B^{+}-Baum. Der Index wird in ”Index-Pages”, welche selbst in Containern mit einer Größe von 16KiB gespeichert werden, abgelegt.
Die Datensätze der Tabelle werden nach der Reihenfolge des Hinzufügens im User Records Bereich gespeichert. Die Einträge selbst weisen immer einen Zeiger auf den vorherigen und den nächsten Eintrag auf. [25]

Frühwirt et al. [25] haben in dem Artikel fünf Methoden aufgezeigt, wie Daten direkt in der Page versteckt werden können. Diese Methoden gliedern sich in die folgende Punkte auf:

- Manipulierung von Suchergebnissen
- Umstrukturierung des Indexes
- Daten innerhalb des ”Garbage Space” verstecken
- Daten innerhalb des ”Free Space” einer Page verstecken
- Page aus dem Index entfernen

3.1.2 InnoDB Database Forensics: Enhaunced Reconstruction of Data Manipulation Queries from Redo Logs

Frühwirt et al. [26] haben in ihrer Arbeit "InnoDB Database Forensics: Enhaunced Reconstruction of Data Manipulation Queries from Redo Logs" gezeigt, dass mithilfe der Redo Logs, die InnoDB während dem Betrieb schreibt, forensische Untersuchungen durchgeführt werden können. Im folgenden wird näher darauf eingegangen.

In der Arbeit wurde gezeigt, dass InnoDB zwei Log-Dateien zum Schreiben der Insert, Update und Delete-Statements verwendet. Bei diesen Log-Dateien handelt es sich um die Dateien `ib_logfile0` und `ib_logfile1`, die in der Standard-Konfiguration bis zu 5MB groß werden können. Das Datenbanksystem schreibt in diese beiden Log-Dateien rotierend die entsprechenden Log-Einträge. Die Dateien bestehen aus:

- **Einem Header-Block**, dieser beherbergt generelle Informationen über das entsprechende Log-File.
- Um einen Verlust von Daten vorzubeugen, werden **zwei Checkpoints** für eine Crash-Wiederherstellung verwaltet.
- **Mehreren Log Blöcken** mit einer größe von 512 Bytes.

Im weiteren Verlauf der Arbeit, haben Frühwirt et al. [26] gezeigt, dass anhand der Tabellen Definition Dateien (`tabellen_name.frm`) die eigentlichen Tabellen der Datenbank analysiert und wiederhergestellt werden können. Diese Dateien beinhalten die Spalten- und Schlüssel-Informationen der Tabellen, womit sie für forensische Zwecke rekonstruiert werden können.

Frühwirt et al. [26] definierten zum Schluss ein Verfahren, wie vorgegangen werden kann, damit Tabellen inklusive deren Inhalt für eine Untersuchung nachgestellt werden können.

3.1.3 Efficient model for detection data and data scheme tempering with purpose of valid forensic analysis

In dem Artikel ”Efficient model for detection data and data scheme tempering with purpose of valid forensic analysis” von Jasmin Azemović und Denis Mušić [27] wird ein Modell aufgezeigt, wie Audit Logs für ein Datenbanksystem generiert und sicher abgespeichert werden können.

Um festzustellen wer, wann und was eine Änderung von Daten veranlasst hat, wurde ein Modell entwickelt, dass auf SQL DML Trigger passiert. Einer dieser implementierten Trigger wird, nachdem ein neuer Eintrag in einer Audit-Tabelle hinzugefügt wurde, ausgeführt.
Um eine zusätzliche Sicherheit in diesem Audit Log zu bieten, werden kryptografische Hash-Funktionen angewendet. Hierzu werden der jeweiligen Audit-Tabelle zwei extra Spalten (`HReserved` und `VReserved`) hinzugefügt. Wird nun ein neuer Eintrag hinzugefügt, so wird, nachdem der Eintrag hinzugefügt wurde, der Trigger angestoßen. Dieser berechnet zwei Hash-Werte, die in den beiden zuvor genannten Spalten gespeichert werden.
In `HReserved` (Horizontal Reserved) wird ein Hash-Wert gespeichert, der über die Spalten des aktuellen Eintrages gebildet wird. In `VReserved` (Vertical Reserved) wird ein Hash-Wert über `HReserved` der letzten beiden Einträge und des aktuellen `HReserved` gebildet.
Mit dieser kryptografischen Hash-Funktion soll sichergestellt werden, dass das Löschen eines Eintrages aus der Audit-Tabelle, durch Überprüfung der `HReserved` und `VReserved` aller Eintrage, festgestellt werden kann.

Durch dieses Überprüfen wird die Audit-Tabelle vor Datenmanipulation geschützt.

3.1.4 Towards a forensic-aware database solution: Using a secured database replication protocol and transaction management for digital investigations

Frühwirt et al. [28] haben in der Arbeit ”Towards a forensic-aware database solution: Using a secured database replication protocol and transaction management for digital investigations” gezeigt, wie mithilfe interner Datenstrukturen für Replikationen und Transaktionen, die bei der Datenbank Crash Recovery Verwendung finden, ein Audit-Trail für einen Sicherheitsmechanismus vor einem Vorfall erstellt werden kann. Dieser Audit-Trail kann nach einem Vorfall verwendet werden, um unautorisierte Änderungen zu bestimmen.

Der Transaktionsmechanismus enthält Informationen, um frühere Versionen der Daten wiederherzustellen, jedoch keine Informationen über Änderungen an den Tabellenstrukturen oder allgemeinen DDL-Änderungen. Der Transaktionsmechanismus findet lediglich für Rollbacks und Undos Verwendung, um die Atomicity der Datenbank zu gewähren.
Der Daten-Replikations-Mechanismus beinhaltet alle Informationen der Datenbank, um diese von einer Master-Instanz auf eine Slave-Instanz zu duplizieren. Diese Datenstruktur beinhaltet alle Daten und auch die logische Struktur der Datenbank. Dieser Mechanismus wird in der Regel verwendet, um eine redundante Speicherung zu ermöglichen.

Bei MySQL funktioniert die Replikation der Master- zur Slave-Instanz mittels zwei Threads. Der Slave verbindet sich zur Master-Instanz und lädt mit dem Thread ”I/O Thread” das ”Binary log” herunter und speichert dieses lokal auf dem Slave als ”Relay log”. Der Thread ”SQL Thread” wendet im Anschluss nach dem Download alle Statements, die im ”Relay log” aufzufinden sind, an.

MySQL unterstützt zwei Methoden von Replikationen.

- Bei ”Statementbasiert” wird jedes SQL Statement gelogt, welches Daten in der Datenbank verändert.
- Bei ”Zeilenbasiert” wird jede Modifikation einer Zeile im Log gespeichert.

In dieser Arbeit werden die Zeiger der einzelnen Log-Einträge in den beiden Log-Dateien ”InnoDB Transaktion Log” und ”MySQL Binary Logs” verändert, sodass während des Schreibens des Log-Eintrages eine Signatur, über den zu schreibenden Log-Eintrages, am Ende des Eintrages hinzugefügt wird. Im Detail wird ein Slack-Space zwischen den einzelnen Log-Einträgen erzeugt, die vom Datenbanksystem ignoriert werden.

Bei den InnoDB Transaktion Logs wird als Trailer des Log-Blocks die Signatur mitgespeichert. InnoDB verwendet aus dem jeweiligen Log-Block Header die Länge des Datensatzes, um auf den nächstfolgenden Log-Block zu schließen. In diesem Fall wurde die Routine, welche die Länge des Log-Blocks in den Header schreibt, erweitert, sodass die Signatur mitberücksichtigt wird und somit ein Slack-Space zwischen den Log-Blöcken entsteht.
Im MySQL Binary Log wurde ähnlich vorgegangen. Um bei diesen Logs einen Slack-Space zwischen den einzelnen Log-Einträgen zu generieren, musste im Header der Offset zum nächsten Log-Eintrag verändert werden. Hierzu wurde die Routine, die den Offset zu dem nächstfolgenden Log-Eintrag in den Header schreibt, erweitert. Die Routine wurde so erweitert, dass sie zum bestehenden Offset zum nächsten Log-Eintrag die Länge der Signatur des aktuellen Log-Eintrages mitberücksichtigt.

Frühwirt et al. [28] haben in dieser Arbeit auch gezeigt, wie mit geschlossenen Datenbanksystemen umgegangen werden kann, sodass auch bei diesen Systemen ein Audit-Trail erstellt werden kann. Der Hauptunterschied liegt darin, dass die Signierung des Datensatzes nicht direkt auf der Master-Instanz stattfindet, sondern die Signierung der Datensätze findet dann statt, wenn diese über das Netzwerk an die Slaves transferiert werden. Hierzu werden die Daten, die für die Replikation übertragen werden, aufgezeichnet und anschließend signiert. Diese signierten Daten werden nachfolgend an die Slaves, die in einer vertrauenswürdigen Umgebung laufen, geleitet. Diese Slaves überprüfen im Vorfeld die Signatur und wenden anschließend die Replikations-Daten an. Danach können die Daten der Slaves in der vertrauenswürdigen Umgebung mit der Master- oder Slave-Instanz, in der nicht vertrauenswürdigen Umgebung verglichen werden. Werden beim Vergleich etwaige Unterschiede aufgedeckt, kann davon ausgegangen werden, dass Daten unautorisiert modifiziert wurden.

Bei beiden Ansätzen ist es wichtig, dass eine ”Chained Witness” angewendet wird. Mit ”Chained Witness” wird sichergestellt, dass die Signatur des einzelnen Log-Eintrages von der Signatur der vorherigen Log-Einträge abhängig ist und somit bei einer Überprüfung der Signatur alle Log-Einträge, Änderungen an früheren Log-Einträgen auffallen, da die Signatur-Kette nicht konsistent ist.

3.2 Bereich NoSQL

Im folgenden Abschnitt werden die Artikel und Arbeiten im Bereich NoSQL aufgezeigt.

- MongoDB Inc. [7] befasste sich mit den Erwägungen bei der Evaluierung von NoSQL Datenbanken.
- Jongseong Yoon et al. [3] widmeten sich einem Framework für forensische Untersuchung von Dokumenten-Datenbanken.
- Werner K. et al. [30] behandelten forensische Merkmale in NoSQL Datenbanken.
- Jongseong Yoon et al. [31] beschäftigten sich mit der Wiederherstellung von gelöschten Einträgen in der MongoDB.
- Rupali Chopade et al. [32] analysierten Wiederherstellungstechniken aus der Redis Datenbank.

3.2.1 Top 5 Considerations When Evaluating NoSQL Databases

MongoDB Inc. hat in einem White Paper [7] Gründe aufgezählt, die dazu führen, dass sich für ein NoSQL Datenbanksystem anstelle eines RDBMS entschieden wird. Darüber hinaus wurden darin fünf Aspekte aufgelistet, die bei der Auswahl eines NoSQL Datenbanksystems berücksichtigt werden sollten.

Gründe, warum sich für ein NoSQL Datenbanksystem entschieden wird, liegen im technischen Bereich (Verarbeitung von Datentypen, die nicht in die Datenstruktur von RDBMS passen), im finanziellen Bereich (Alternativen zu teuren proprietären RDBMS) und im Lifecycle des Unternehmens (agile Arbeitweise, schnelle Entwicklungszyklen).

MongoDB Inc. nannte in ihrem White Paper die folgenden fünf Aspekte, die bei der Auswahl eines NoSQL Datenbanksystems evaluiert werden sollen. [7]

- **Datenmodell**
  Bei "Datenmodell" nannte MongoDB Inc. die Datenmodelle Dokument Model, Graph Model, Key-Value Model und Wide Column Model, die je nach Datenbanksystem, das in Frage kommt, evaluiert werden sollen.
- **Abfragemodell**
  Dieser Punkt muss zusammen mit dem ersten Punkt evaluiert werden, da jede Applikation ihre eigenen Abfrageanforderungen hat und das benötigte Abfragemodell vom ausgewählten Datenmodell abhängig ist.

- **Konsistenz- und Transaktionsmodell**
  Im White Paper wird bei diesem Punkt zwischen stark konsistenten Systemen und eventuell konsistenten Systemen unterschieden. Der Hauptgedanke liegt darin, dass in diesem Fall evaluiert werden soll, ob die Replikation und Verteilung an Slave-Instanzen sofort oder erst zu einem späteren Zeitpunkt erfolgen soll. Je nach Anwendungsfall muss überlegt werden, ob nach einer Datenänderung die geänderten Daten sofort an alle Slave-Instanzen verteilt werden oder diese erst zu einem späteren Zeitpunkt an den Slave-Instanzen zur Verfügung stehen.
- **API´s**
  Bei NoSQL Datenbanksystemen gibt es keinen definierten Standard, wie mit diesen interagiert werden kann. Deshalb soll im Vorfeld evaluiert werden, welches Interface für den jeweiligen Anwendungsfall passend ist. MongoDB Inc. listet hier idiomatische Treiber, Restful API´s und SQL-Like API´s als mögliche Schnittstellen der NoSQL Datenbanksysteme auf.
- **Kommerzielle Unterstützung, Vendor Lock-In**
  Zu guter Letzt geht MongoDB Inc. auf Support und Community ein. In diesem Fall soll evaluiert werden, ob zu dem entsprechenden Datenbanksystem ein kommerzieller Support angeboten wird und ob eine starke Community besteht, damit auf Best Practices und auf weitere Vorteile einer starken Community zurückgegriffen werden kann.

3.2.2 Forensic investigation framework for the document store NoSQL DBMS: MongoDB as a case study

Jongseong Yoon et al. [3] präsentierten in ihrem Artikel ein Framework für forensische Investigation im Bereich der NoSQL-Datenbanken. Zu Beginn wird erwähnt, dass Khanuja Harmeet Kaur und Adane [33] ein Sechs-Phasen-Framework für die allgemeine forensische Untersuchung von RDBMS erarbeitet haben. Diese Phasen wurden wie folgt definiert: Identifizierung, Sammlung, Validierung, Interpretation, Generierung eines forensischen Berichts und Sicherung der Daten als Beweise.

Im Folgenden wird auf das Framework für NoSQL-Datenbanken von Jongseong Yoon et al. [3] eingegangen. Dieses Framework basiert auf fünf Phasen, diese nennen sich wie folgt: Vorbereitung, logische Beweiserfassung und -sicherung, Beschaffung und Sicherung von verteilten Beweismitteln, Untersuchung und Analyse sowie Berichterstattung und Präsentation.

- **Phase 1: Vorbereitung**
  Diese Phase beschäftigt sich mit der Identifizierung des Types des NoSQL Datenbanksystems und mit der Beschaffung von Zugriffsberechtigungen auf diese. Des Weiteren wird sich mit der Identifizierung der Replikations- und Verteilungsumgebung befasst und zusätzlich sollen in dieser Phase Informationen über das Zielsystem selbst eingeholt werden.
- **Phase 2: logische Beweiserfassung und -sicherung**
  In der zweiten Phase werden die gespeicherten Daten in der Datenbank aufgelistet und die Schemen der unterschiedlichen Dokumente analysiert.
- **Phase 3: Beschaffung und Sicherung von verteilten Beweismitteln**
  In der dritten Phase werden die physischen Server, welche zum verteilten System gehören, ausfindig gemacht. Anschließend werden die Daten auf den identifizierten Servern erfasst und für die weitere Analyse vorbereitet.
- **Phase 4: Untersuchung und Analyse**
  In der vierten Phase werden die gesammelten Beweise mit technischen Methoden und geeigneten Verfahren analysiert.
- **Phase 5: Berichterstattung und Präsentation**
  In der letzten Phase wird das Ergebnis der vierten Phase dokumentiert und für eventuelle rechtliche Schritte in einer einwandfreien Form niedergeschrieben.

Im Anschluss der Definition des Frameworkes, wurde dieses anhand eines Beispiels in einer MongoDB Implementierung angewendet.

3.2.3 Forensic attribution in NoSQL databases

In der Arbeit ”Forensic attribution in NoSQL databases’ von Werner K. Hauger und Martin S. Olivier [30] wurden NoSQL Datenbanksysteme untersucht, ob diese Sicherheitsmerkmale besitzen, die für eine forensische Zuordnung relevante Spuren hinterlassen. Darüber hinaus wurden die Logging Mechanismen der Datenbanksysteme unter die Lupe genommen. In der Arbeit wurden die NoSQL Datenbanksysteme MongoDB, Cassandra, Redis und Neo4j analysiert. Im Detail wurden die Features, die in der Free und Community-Version der genannten Datenbanksysteme zur Verfügung stehen, zur Analyse herangezogen.

In der Analyse wurden die Sicherheitsmerkmale Authentifizierung, Autorisierung und Logging untersucht. Alle vier Systeme bieten die Sicherheitsmerkmale der Authentifizierung und des Logging in der Free und Community-Version an. MongoDB und Cassandra bieten in der Free und Community-Version des Weiteren das Sicherheitsmerkmal der Authorisierung an. Redis und Neo4J haben die Möglichkeit der Autorisierung nicht implementiert. In der Tabelle 3.1 ist die Auflistung der Sicherheitsmerkmale grafisch dargestellt. MongoDB und Redis konnten bei der Untersuchung beweisen, dass sie die Möglichkeit bieten, dass jedes Query, das abgesetzt wird, in einem Log mitgeschrieben wird und somit für eine forensische Untersuchung herangezogen werden kann.

Datenbanksystem	**Authentifizierung**	**Authentisierung**	**Logging**
MongoDB	✓	✓	✓
Cassandar	✓	✓	✓
Redis	✓	✗	✓
Neo4j	✓	✗	✓

Tabelle 3.1: NoSQL Sicherheitsmerkmale [30]

Im zweiten Schritt untersuchten Werner K. Hauger und Martin S. Olivier, welche Features nach einer Default-Installation der Datenbanksysteme aktiv sind. Beim Thema Access-Control konnte lediglich Neo4J mit einer standardmässigen aktivierten Zugriffskontrolle punkten. Positiv vielen dabei die Systeme Cassandra und Redis auf, da das Logging bereits bei der Standard-Installation aktiviert ist. In der Tabelle 3.2 ist die Auflistung der Features, die bei der Standard-Installation aktiv sind, grafisch dargestellt.

Datenbanksystem	**Access-Controll**	**Logging**
MongoDB	✗	✗
Cassandar	✗	✓
Redis	✗	✓
Neo4j	✓	✗

Tabelle 3.2: Features, die bei der Standard-Installation aktiv sind [30]

Zum Ende der Arbeit wurde nochmal zusammengefasst, dass das Sicherheitsmerkmal ”Access-Controll” in MongoDB und Cassandra zur forensischen Zuordnung herangezogen werden kann, da eine Authentifizierung und Autorisierung implementiert wurde und verfügbar ist. Je nach verwendetem NoSQL Datenbanksystem können dessen Audit-Logs, System-Logs und Storage-Logs für die forensische Zuordnung herangezogen werden.

3.2.4 A method and tool to recover data deleted from a MongoDB

Jonseong Yoon und Sangjin Lee haben in ihrem Artikel "A method and tool to recover data deleted from a MongoDB" [31] Untersuchungen durchgeführt, um eine Wiederherstellungsmethode von gelöschten Daten in MongoDB zu finden. Sie sind dabei sehr genau auf die Storage-Engines "WiredTiger" und "MMAPv1" eingegangen. Im Zuge des Artikels wurde ein Tool entwickelt, dass deren Wiederherstellungsalgorithmus für WiredTiger und MMAPv1 implementiert.

MMAPv1 wurde bis zur MongoDB Version 3.2 als die Standard Storage-Engine verwendet. Seit MongoDB Version 3.2 ist als Standard Storage-Engine WiredTiger im Einsatz.

- **MMAPv1 Storage-Engine**
  Bei dieser Storage-Engine werden pro Datenbank zwei Dateien (Namespace-File & Datendatei) erzeugt. Das Namespace-File trägt den Dateinamen `<database-name>.ns` und in diesem werden die Metadaten (Datenmenge und Anzahl an Einträgen) von Collections abegelegt. Die Datendatei trägt den Namen `<database-name>.<nummer>` und in dieser werden die ursprünglichen Daten gespeichert. Diese Datendatei beginnt mit der Nummer 0 und mit steigender Volumegröße wird eine neue Datei mit der nächsten Nummer erzeugt. Pro 2GB an Daten wird eine Datendatei erstellt. Die Datendatei besteht aus einem sogenannten "Extent" und einer Datensatzstruktur. Der erwähnte Extent ist ein zusammenhängender Block, der viele Datensätze beherbergt. Im Datensatz selbst, werden die eigentlichen Daten als BSON-Dokumente (siehe Abschnitt 4.2) gespeichert. Extents, die derselben Collection angehören, sind mit einer doppelt verlinkten Liste verbunden. Die Daten im Extent sind ebenfalls in einer doppelt verlinkten Liste organisiert.
  Werden Dokumente aus der Datenbank mit einer MMAPv1 Storage-Engine gelöscht, so werden nur die ersten vier Bytes des Datensatzes zu `0xEEEEEEEE` geändert. Der restliche Datensatz bleibt unverändert und der Speicherort des gelöschten Datensatzes wird im Namespace-File vermerkt.
  Wird eine Collection gelöscht, so werden lediglich die Metadaten im Namespace-File gelöscht und die ursprünglichen Daten bleiben bestehen.
  Wird die gesamte Datenbank gelöscht, so wird das entsprechende Namespace-File und die dazugehörigen Dateien vom Dateisystem gelöscht.

- **WiredTiger Storage-Engine**
  Eine Eigenheit dieser Engine ist die Verwaltung der Daten in ”page units”. Die Seiten selbst sind mit einer B-Baum Struktur verbunden. In jeder Page werden die Daten mit der Snappy Komprimierung [34] in einer Zellen-Struktur gespeichert. Wie in der MMAPv1-Implementierung, werden die Daten ebenfalls mittels BSON-Dokumenten verwaltet.
  Wird bei der WiredTiger-Implementierung ein Dokument gelöscht, so wird die Seite, die diese Daten beinhaltet, aus dem B-Baum entfernt und eine neue Seite in dem B-Baum verknüpft. Die entfernte Seite mit den gelöschten Daten wird nicht sofort von dem Speichermedium gelöscht.
  Wird eine Collection oder Datenbank gelöscht, so wird die gesamte Datendatei vom Dateisystem gelöscht.

Nach der Erklärung der Funktionsweise der beiden Storage-Engine, stellten Jonseong Yoon und Sangjin Lee die Methoden wie folgt dar, um gelöschte Daten wiederherzustellen:

- **MMAPv1 | Metadata aus dem Namespace-File**
  Dadurch, dass beim Löschen einzelner Einträge die Datensätze nicht gelöscht oder überschrieben werden, sondern lediglich die ersten vier Bytes markiert werden und der Speicherort des gelöschten Datensatzes im Namespace-File vermerkt wird, müssen nur die vermerkten Datensätze eruiert werden und anschließen können diese im Filesystem aufgefunden werden. Mithilfe der Informationen der Größe des Datensatzes im Namespace-File, können sogar die ersten vier Bytes des gelöschten Datensatzes wiederhergestellt werden.
- **MMAPv1 | Signatur des gelöschten Datensatz**
  Sollte kein Namespace-File zur Verfügung stehen, so kann direkt mithilfe von File-Carving das Dateisystem auf die Bytesignatur `0xEEEEEEEE` überprüft werden. Dadurch, dass diese Bytesignature eine große False-Positive Rate liefert, haben Jonseong Yoon und Sangjin Lee, mit dem Wissen, dass der erste Key im BSON-Dokumente immer `_id` ist, die Methode so erweitert, dass zusätzlich auf das Vorkommen des ersten Keys überprüft wird.
- **WiredTiger**
  Bei WiredTiger haben Yoon und Lee eine Methode entwickelt, mit der alle Seiten auf das Vorkommen im B-Baum überprüft werden. Wird eine Seite vorgefunden, die nicht im B-Baum enthalten ist, kann darauf geschlossen werden, dass es sich bei dieser Seite, um eine Seite mit gelöschten Daten handelt.

3.2.5 A data recovery technique for Redis using internal dictionary structure

In der Arbeit ”A data recovery technique for Redis using internal dictionary structure” von Rupali Chopade und Vinod Pachghare [32] wurde eine Methode vorgestellt, um gelöschte Daten eines Redis Datenbanksystem wiederherzustellen.

Das Konzept von Redis passiert auf 16 Datenbanken, welche von 0 bis 15 durchnummeriert sind. Wird ein Eintrag aus einer Redis Datenbank gelöscht, so wird dieser Eintrag aus dem ”Hash Table Bucket” gelöscht und da es sich um ein In-Memory Datenbanksystem handelt, wird dessen allozierter Speicher eventuell freigegeben oder eventuell nicht. Wird der Speicher nicht freigegeben, so kann auf den Wert zugegriffen werden, bis dieser überschrieben wurde. Wurde der Speicher freigegeben, so kann auf den Wert nicht mehr zugegriffen werden, da der Speicher de-alloziert und zurück an den Heap gegeben wurde.

Die Methode von Rupali Chopade und Vinod Pachghare basiert darauf, dass das Datenbanksystem mit der Option ”nofree” betrieben wird und bei gelöschten Datensätzen diese anschließend aus dem `dictEntry` wiederhergestellt werden.

3.3 Resultat

Anhand der durchgeführten Analyse vom Stand der Technik der Datenbank-Forensik, im Bereich SQL und NoSQL, kann folgendes Resümee gezogen werden:
Datenbank-Forensik ist unumgänglich in der forensischen Aufarbeitung eines Cyber-Vorfalles. Im Bereich SQL decken die vorgestellten Methoden zur Wiederherstellung von Daten und den Tabellenstrukturen einen breiten forensischen Bereich ab. Dadurch, dass RDBMS schon länger in vielen Anwendungen verwendet werden, bietet die Literatur viele verschiedene Ansätze, wie bei einer forensischen Bearbeitung vorgegangen werden soll.
NoSQL Datenbanksysteme sind aktuell erst im Aufschwung und finden im Großteil Verwendung in Big-Data Applikationen. Dadurch bietet die Literatur nicht allzu viele technische Vorgehensweisen zu den betrachteten Datenbanktyp-Systemen an. Was die Literatur jedoch dennoch zu diesem Bereich bietet, sind Frameworks, die für forensische Untersuchungen von NoSQL Datenbanksysteme herangezogen werden können.
Im Zuge dieser Analyse wurde festgestellt, dass sich der Ansatz von Frühwirt et al. [28] mit der Replikation von geschlossenen Datenbanksystemen gut eignet, um in den Bereich NoSQL übergeführt zu werden.

4 Herangehensweise

Im folgenden Kapitel erfolgt die Darlegung der Herangehensweise der vorliegenden Arbeit. Hierfür wird zu Beginn das methodische Vorgehen geschildert, gefolgt von der Beschreibung der allgemeinen Funktionen von MongoDB und darüber hinaus wird in Abschnitt 4.3 auf die Planung und Entwicklung des Prototyps zur Erstellung des Audit-Trails eingegangen.

4.1 Methodisches Vorgehen

Um diese Arbeit durchzuführen wurde das folgende methodische Vorgehen angewendet:

1. **Literaturanalyse**

   Im ersten Schritt wurde der Stand der Technik im Bereich SQL und NoSQL analysiert und eruriert, ob eine Technik aus dem Bereich SQL in den Bereich NoSQL übergeführt werden kann.

2. **Prototyping-Entwicklung**

   Im zweiten Schritt wurde ein Prototyp entwickelt, der die Replikation-Logs der MongoDB verwendet, um einen sicheren Audit-Trail zu erstellen. Dieser Audit-Trail kann im Anschluss über eine API, zur Verifizierung der Daten angesprochen werden.

3. **Feldversuch**

   Im dritten Schritt wurde ein Feldversuch mit Test-Daten durchgeführt.

4. **Analyse des Feldversuchs**

   Im vierten Schritt wurde der Audit-Trail des Feldversuchs analysiert.

Diese Arbeit beschränkt sich auf das Datenbank-System MongoDB. Dies folgt daraus, dass MongoDB ein Interface/Collection zur Verfügung stellt, womit die Replikation-Logs abgegriffen werden können. Des Weiteren wird für die On-Premise Lösung die Version Community Server [35] der MongoDB verwendet und für die SaaS Lösung die Shared-Tier [36] Version herangezogen.

4.2 MongoDB

In diesem Kapitel werden auf allgemeine Funktionen von MongoDB eingegangen. Des Weiteren wird in Abschnitt 4.2.3 auf den OpLog, der die Daten für die Replikation, welche im Zuge der Arbeit verwendet werden, zur Verfügung stellt, eingegangen.

4.2.1 Allgemeines über MongoDB

MongoDB ist ein dokumentenbasierendes NoSQL Datenbank-System der Firma MongoDB Inc. [8]. Der Startschuss zur Entwicklung von MongoDB ist 2007 durch die Entwickler Dwight Merriam, Eliot Horowitz und Kevin Ryan gefallen. [37] Die Grundlagen zu dokumentenbasierenden NoSQL-Datenbanken werden in Abschnitt 2.3.1 aufgezeigt.

MongoDB wurde für diese Arbeit wegen den folgenden Gründen ausgewählt:

- stellt eine Collection für Replikation-Logs zur Verfügung
- kann in einer Cloud-Umgebung als SaaS ausgeführt werden
- kann On-Premise im eigenen Datacenter betrieben werden

Die Collection, welche MongoDB für die Replikation-Logs zur Verfügung stellt, ist der Oplog (Operations log). Auf diesen Oplog wird näher in Abschnitt 4.2.3 eingegangen.
MongoDB Inc. stellt MongoDB als Download zur Verfügung, sodass das Datenbank-System On-Premise ausgeführt werden kann. Darüber hinaus stellt MongoDB Inc. selbst eine Infrastruktur zur Verfügung, sodass MongoDB als SaaS genutzt werden kann. Je nach Anwendungsfall der Applikation und Daten, die in der Applikation verwaltet werden, ist zu entscheiden, ob eine On-Premise- oder SaaS-Lösung gewählt werden sollte.

In Dokumenten Datenbanken ähnelt die Struktur der Datensätze JSON-Objekte (Abschnitt 2.3.1). MongoDB verwaltet die Datensätze als BSON-Dokumente. MongoDB beschreibt BSON wie folgt:
"*A serialization format used to store documents and make remote procedure calls in MongoDB. "BSON" is a portmanteau of the words "binary" and "JSON". Think of BSON as a binary representation of JSON documents.*" [38]

Dokumente in MongoDB weisen als ersten Key immer das Feld `_id`, eine eindeutige ObjectId per Sammlung, auf. Dieser Key dient als Primärschlüssel.

4.2.2 Unterschiedliche MongoDB Bereitstellungsarten

MongoDB kann in drei verschiedenen Arten zur Verfügung gestellt werden. Diese drei verschiedenen Arten sind Standalone, Replica Set und Sharded Cluster.

Standalone

Dieser Bereitstellungstyp kann gewählt werden, wenn es sich um eine Verwendung für eine lokale Entwicklungsumgebung handelt. Hierbei wird lediglich ein `mongod` Prozess auf einem Server/PC ausgeführt. In diesem Fall befinden sich alle Daten auf dem Host-System, auf dem der `mongod` Prozess ausgeführt wird. Wird eine MongoDB-Instanz im "Standalone" Bereitstellungstyp betrieben, ist dies durch ihren Prompt ersichtlich. Bei diesem Bereitstellungstyp wird der Shell-Prompt wie folgt dargestellt: `>`

Replica Set

Bei diesem Bereitstellungstyp besteht MongoDB aus einer Gruppe von `mongod` Prozessen, die denselben Datensatz verwalten. Im Detail werden diese Prozesse auf unterschiedlichen Servern ausgeführt, sodass eine Redundanz und Hochverfügbarkeit geboten wird. Die Daten werden automatisch von MongoDB zwischen allen Servern repliziert. Dieser Bereitstellungstyp sollte für Produktionsumgebungen genutzt werden. [39]
Bei diesem Bereitstellungstyp wird der Shell-Prompt wie folgt dargestellt:
`<Replica set name: member typ>`

Sharded Cluster

Bei diesem Bereitstellungstyp werden die Daten, verteilt auf mehrere Server, aufgeteilt. Dieser Bereitstellungstyp soll gewählt werden, wenn der Anwendungsfall die Bereitstellung von sehr großen Datensätzen und großem Durchsatz erfordert. Dadurch kann die Kapazität mehrerer Server zusammen genutzt werden, um diese großen Datensätze zur Verfügung zu stellen.
Bei diesem Bereitstellungstyp wird der Shell-Prompt wie folgt dargestellt: `mongos>`

4.2.3 OpLog

Der OpLog ist eine spezielle Collection, die von MongoDB bei der Verwendung des Bereitstellungtypes ”Replica Set” verwaltet wird. Dieser OpLog beinhaltet eine Aufzeichnung über alle Operationen, die Daten in den Datenbanken verändert haben. Auf den primär Datenbank-Instanzen des ”Replica Set” wird der OpLog geschrieben und den sekundär Datenbank-Instanzen zur Verfügung gestellt. Die sekundär Datenbank-Instanzen kopieren diesen OpLog und wenden diese Operationen asynchron auf ihre Datenbank-Instanzen an.
Der OpLog wird in der Sammlung `db.oplog.rs` in der Datenbank `local` verwaltet. In Listing 5 wird dargestellt, wie Informationen aus dem OpLog abgerufen werden können.

```
1 Atlas atlas-44waib-shard-0 [primary] MyFirstDB> use local
2 switched to db local
3 Atlas atlas-44waib-shard-0 [primary] local> db.oplog.rs.find()
```

Listing 5: MongoDB | OpLog auf einer primär Datenbank-Instanz abrufen

Im folgenden Kapitel wird auf den allgemeinen Aufbau eines OpLog-Eintrages eingegangen.

Aufbau eines OpLog-Eintrages

Die OpLog-Einträge sind im Allgemeinen wie in Listing 6 dargestellt, aufgebaut.
Ein OpLog-Eintrag besteht hauptsächlich aus den folgenden Feldern, je nach Aktion können jedoch weitere Felder im OpLog-Eintrag vorhanden sein.

- Das Feld `op` spezifiziert, um welche Art von Operation es handelt.
  - **c**: Create
  - **i**: Insert
  - **u**: Update
  - **d**: Delete
- Das Feld `ns` gibt an, um welchen Namespace es sich handelt. Der Namespace ist wie folgt zusammengestellt: `<Datenbank>.<Collection>`
- Das Feld `ui` spezifiziert die eindeutige ID des betroffenen Objekts.

- Im Feld `o` wird das betroffene Objekt angeführt. Handelt es sich bei dem OpLog-Eintrag um einen Create- oder Update-Eintrag, so wird in diesem Feld das entsprechende Dokument oder die aktualisierten Felder angeführt.
- Das Feld `ts` stellt ein Unix-Timestamp zur Verfügung. Des Weiteren stellt dieses Feld einen Parameter "i" zur Verfügung, dieser gibt die Reihenfolge der Einfügungen in dieser Sekunde an.
- Das Feld `t` wurde für die Umsetzung des Prototyps nicht berücksichtigt.
- Das Feld `v` wurde ebenso nicht für die Umsetzung des Prototyps verwendet.
- Im Feld `wall` wird der entsprechenden Zeitstempel als ISO-Datum zur Verfügung gestellt.

```
1  {
2      op: 'c',
3      ns: 'masterDB.$cmd',
4      ui: UUID("5c5c0c01-0f1b-4dd1-a27f-8af04553f5ad"),
5      o: {
6        create: 'demo',
7        idIndex: { v: 2, key: { _id: 1 }, name: '_id_' }
8      },
9      ts: Timestamp({ t: 1679068818, i: 42 }),
10     t: Long("815"),
11     v: Long("2"),
12     wall: ISODate("2023-03-17T16:00:18.600Z")
13 }
```

Listing 6: MongoDB | Allgemeiner Aufbau eines OpLog-Eintrages

Analysierte OpLog-Einträge

In den folgenden Punkten, werden die OpLog-Einträge analysiert, die für die Entwicklung des Prototyps zur Erstellung eines Audit-Trails herangezogen wurden.

- **Erstellung einer Datenbank & Collection**

  In Listing 7 ist ein OpLog-Eintrag ersichtlich, der protokolliert, dass eine Collection mit dem Namen "demo" erstellt wurde. Zusätzlich zu den Feldern, die in Abschnitt 4.2.3 erklärt wurden, findet sich in diesem Log-Eintrag das Feld `o` mit einem Objekt vor. Dieses Objekt spezifiziert den Namen der Collection.

  Anhand des Namespaces, des Objekts und der Art der Operation "c" (Create) kann darauf geschlossen werden, dass in der Datenbank `masterDB` die Collection `demo` erstellt wurde. Besteht zum Zeitpunkt des Erstellens der Collection die Datenbank noch nicht, so wird diese im selben Zuge erstellt. Das Erstellen der Datenbank triggert keinen weiteren Eintrag im Log.

```
{
    op: 'c',
    ns: 'masterDB.$cmd',
    ui: UUID("5c5c0c01-0f1b-4dd1-a27f-8af04553f5ad"),
    o: {
      create: 'demo',
      idIndex: { v: 2, key: { _id: 1 }, name: '_id_' }
    },
    ts: Timestamp({ t: 1679068818, i: 42 }),
    t: Long("815"),
    v: Long("2"),
    wall: ISODate("2023-03-17T16:00:18.600Z")
}
```

Listing 7: MongoDB | Aufbau eines OpLog-Eintrages bei der Erstellung einer Collection

- **Erstellung eines Dokumentes**

  In Listing 8 zeigt ein OpLog-Eintrag, dass ein Dokument mit der Objekt-Id ”64148e922c11248c9a85bf1d” hinzugefügt wurde. Anhand dem Namespace und der Art der Operation ”i” (Insert) kann darauf geschlossen werden, dass dieses Dokument der Collection `demo`, die sich in der Datenbank `masterDB` befindet, hinzugefügt wurde. Im Feld `o` ist das Dokument, inklusive dessen eindeutigen Objekt-Id in dem Feld `_id`, ersichtlich. Das Objekt besteht aus einer eindeutigen Objekt-Id `_id`, einem Feld `name` mit dem Wert ”Florian Gruber” und einem weiteren Feld `age` mit dem Wert 27.

```
{
    lsid: {
        id: UUID("06ad640c-f0e8-4128-9888-26a0e4e8ee4c"),
        uid:
        ↪   Binary(Buffer.from("75daa8b7af2f2e90543e73aea858f671d0a5bff76ffef17
        ↪   "hex"), 0)
    },
    txnNumber: Long("1"),
    op: 'i',
    ns: 'masterDB.demo',
    ui: UUID("5c5c0c01-0f1b-4dd1-a27f-8af04553f5ad"),
    o: {
        _id: ObjectId("64148e922c11248c9a85bf1d"),
        name: 'Florian Gruber',
        age: 27
    },
    ts: Timestamp({ t: 1679068818, i: 43 }),
    t: Long("815"),
    v: Long("2"),
    wall: ISODate("2023-03-17T16:00:18.624Z"),
    stmtId: 0,
    prevOpTime: { ts: Timestamp({ t: 0, i: 0 }), t: Long("-1") }
}
```

Listing 8: MongoDB | Aufbau eines OpLog-Eintrages bei der Erstellung eines Dokumentes

- **Aktualisierung eines Dokumentes**

  In Listing 9 - 11 sind OpLog-Einträge ersichtlich, die protokollieren, dass ein Dokument mit der Objekt-Id "64148e922c11248c9a85bf1d" aktualisiert wurde. Dass ein Dokument aktualisiert wurde, ist anhand der Art der Operation "u" (Update) ersichtlich. Wie im Log-Eintrag davor, ist in diesem Fall erkenntlich, dass es sich um eine Änderung in der Collection `demo` in der Datenbank `masterDB` handelt.

  In Listing 9 ist im Feld `o` ein Objekt mit weiteren Feldern angeführt. In diesem Objekt befindet sich ein Feld `diff`. Anhand dem Wert in diesem Feld, kann darauf geschlossen werden, was im Detail in diesem referenzierten Dokument verändert wurde. In diesem Fall kann durch das "i" (Insert) darauf geschlossen werden, dass dem Dokument mit der Objekt-Id "64148e922c11248c9a85bf1d" ein Feld mit dem Namen `job` und dem Wert `Software Engineer` hinzugefügt wurde.

  In Listing 10 kann dem Objekt im Feld `diff` anhand des Indikators "u" (Update) entnommen werden, dass im referenzierten Dokument das Feld mit dem Namen `job` aktualisiert wurde. Im Detail wurde der Wert dieses Feldes zu `QA Tester` verändert.

  In Listing 11 ist im Feld `diff` ein "d" (Delete) und das Feld `job` mit einem Wert `false` ersichtlich. Anhand dieser Informationen kann darauf geschlossen werden, dass das Feld `job` im referenzierten Dokument entfernt wurde.

```
1  {
2      op: 'u',
3      ns: 'masterDB.demo',
4      ui: UUID("5c5c0c01-0f1b-4dd1-a27f-8af04553f5ad"),
5      o: { '$v': 2, diff: { i: { job: 'Software Engineer' } } },
6      o2: { _id: ObjectId("64148e922c11248c9a85bf1d") },
7      ts: Timestamp({ t: 1679069158, i: 7 }),
8      t: Long("815"),
9      v: Long("2"),
10     wall: ISODate("2023-03-17T16:05:58.241Z")
11 }
```

Listing 9: MongoDB | Aufbau eines OpLog-Eintrages bei der Aktualisierung eines Dokumentes (Hinzufügen eines Feldes)

```
{
    op: 'u',
    ns: 'masterDB.demo',
    ui: UUID("5c5c0c01-0f1b-4dd1-a27f-8af04553f5ad"),
    o: { '$v': 2, diff: { u: { job: 'QA Tester' } } },
    o2: { _id: ObjectId("64148e922c11248c9a85bf1d") },
    ts: Timestamp({ t: 1679139571, i: 99 }),
    t: Long("816"),
    v: Long("2"),
    wall: ISODate("2023-03-18T11:39:31.046Z")
}
```

Listing 10: MongoDB | Aufbau eines OpLog-Eintrages bei der Aktualisierung eines Dokumentes (Aktualisierung eines Feldes)

```
{
    op: 'u',
    ns: 'masterDB.demo',
    ui: UUID("5c5c0c01-0f1b-4dd1-a27f-8af04553f5ad"),
    o: { '$v': 2, diff: { d: { job: false } } },
    o2: { _id: ObjectId("64148e922c11248c9a85bf1d") },
    ts: Timestamp({ t: 1679139641, i: 14 }),
    t: Long("816"),
    v: Long("2"),
    wall: ISODate("2023-03-18T11:40:41.340Z")
}
```

Listing 11: MongoDB | Aufbau eines OpLog-Eintrages bei der Aktualisierung eines Dokumentes (Löschen eines Feldes)

- **Löschung eines Dokumentes**

  Listing 12 zeigt ein OpLog-Eintrag die Löschung eines Dokumentes mit der Objekt-Id ”64148e922c11248c9a85bf1 auf. Dass ein Dokument gelöscht wurde, kann anhand der Art der Operation ”d” (Delete) entnommen werden. Anhand des Namespace ist ersichtlich, dass es sich um ein Dokument in der Collection `demo` , welche sich in der Datenbank `masterDB` befindet, handelt.

```
{
    op: 'd',
    ns: 'masterDB.demo',
    ui: UUID("5c5c0c01-0f1b-4dd1-a27f-8af04553f5ad"),
    o: { _id: ObjectId("64148e922c11248c9a85bf1d") },
    ts: Timestamp({ t: 1679069199, i: 34 }),
    t: Long("815"),
    v: Long("2"),
    wall: ISODate("2023-03-17T16:06:39.620Z")
}
```

Listing 12: MongoDB | Aufbau eines OpLog-Eintrages bei der Löschung eines Dokumentes

- **Löschung einer Collection**
  In Listing 13 ist ein OpLog-Eintrag ersichtlich, der protokolliert, dass eine Collection mit dem Namen `demo` gelöscht wurde. Durch diese Information kann darauf geschlossen werden, dass im Namespace nur die Datenbank `masterDB` ersichtlich ist und im Feld `o` als Wert ein Feld mit dem Namen `drop` und als Wert mit dem Namen der Collection `demo` vorgefunden werden kann. Als weitere Information kann diesem Log-Eintrag entnommen werden, wieviele Dokumente mit dem Löschen der Collection gelöscht wurden. In diesem Fall kann dem Feld `numRecords` entnommen werden, dass in dieser Collection zum Zeitpunkt der Löschung keine Dokumente gespeichert waren.

```
1  {
2      op: 'c',
3      ns: 'masterDB.$cmd',
4      ui: UUID("5c5c0c01-0f1b-4dd1-a27f-8af04553f5ad"),
5      o: { drop: 'demo' },
6      o2: { numRecords: 0 },
7      ts: Timestamp({ t: 1679069239, i: 64 }),
8      t: Long("815"),
9      v: Long("2"),
10     wall: ISODate("2023-03-17T16:07:19.627Z")
11 }
```

Listing 13: MongoDB | Aufbau eines OpLog-Eintrages bei der Löschung einer Collection

4.3 Resultat

Anhand der durchgeführten Analyse des OpLog´s in Abschnitt 4.2.3 konnte festgestellt werden, dass anhand dieser OpLog-Einträge ein Audit-Trail abgeleitet werden kann. Mit den einzelnen OpLog-Einträgen wird jede Veränderung in den Datenbanken von MongoDB aufgezeichnet. Im folgenden Kapitel wird auf die Planung und Entwicklung des Prototyps zur Erstellung eines Audit-Trails eingegangen.

5 Implementierung & Evaluierung

In diesem Kapitel wird auf die Planung, Entwicklung und Evaluierung des Prototyps zur Erstellung eines Audit-Trails eingegangen. Zuerst wird in Abschnitt 5.1 auf das Big Picture der Prototypplanung eingegangen. Danach wird die Entwicklung und Umsetzung des Prototyps in Abschnitt 5.2 erläutert. Zum Schluss erfolgt in Abschnitt 5.3 die Evaluierung des Prototyps mit einem Set an Testdaten.

5.1 Planung des Prototyps

Im Zuge der Analyse der Dokumenten Datenbank-Systeme wurde festgestellt, dass diese Datenbank-Systeme als On-Premise Installation und als SaaS in der Cloud zur Verfügung gestellt werden. Durch diese Gegebenheit musste ein Ansatz gefunden werden, mit dem es möglich ist, vom ausgewählten Datenbank-System sowohl On-Premise als auch SaaS abgedeckt werden.

Somit wurde sich für den Ansatz einer Client-Server Architektur entschieden. In Abb. 5.1 ist eine Übersicht des Prototyps ersichtlich. In der Abbildung fungiert der "Daemon" als Client. Als Server fungiert der "Audit-Trail API-Enpoint", welcher in der Arbeit folgend als "API-Endpoint" bezeichnet wird. Zur leichteren Überprüfung der Audit-Trail kann eine Web-Oberfläche zur Verfügung gestellt werden, sodass der Audit-Trail grafisch ausgewertet werden kann. Die Web-Oberfläche wird in dieser Arbeit nicht umgesetzt, da alle Funktionen zur Überprüfung des Audit-Trails über den API-Endpoint zur Verfügung gestellt werden und dadurch eine manuelle Überprüfung ermöglicht wird.
Mit dieser Architektur wird gewährt, dass nicht nur ein Daemon, der den OpLog von MongoDB Instanzen verarbeitet, sondern auch weitere Dokumenten Datenbank-Systeme, wie zum Beispiel CosmosDB, mit einem abgeleiteten Daemon passend zum Datenbank-System angebunden werden können.

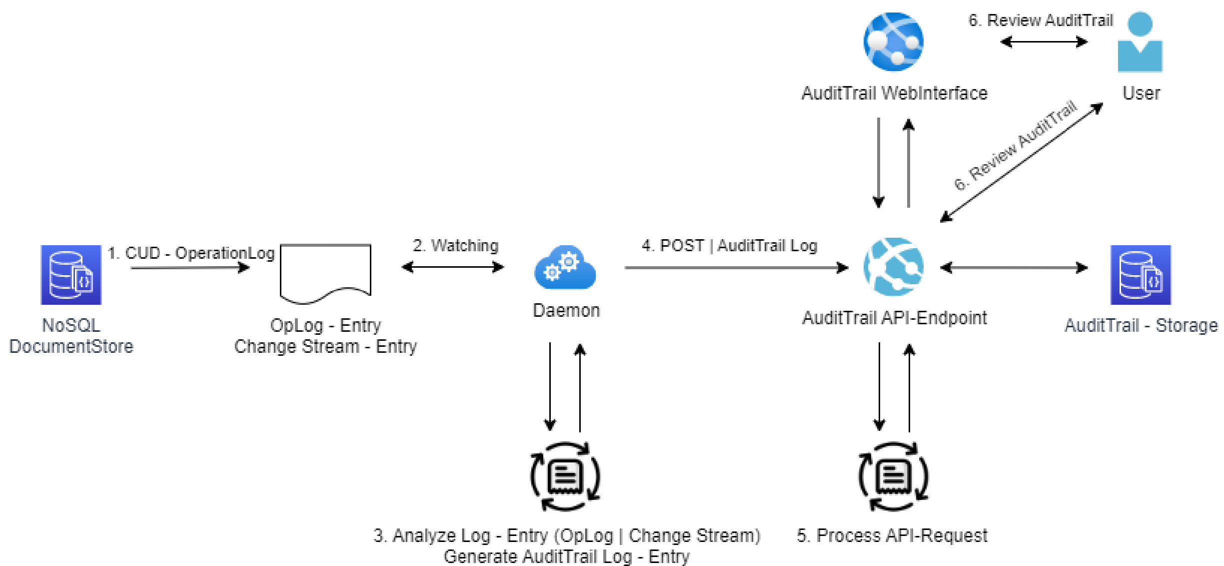

Abbildung 5.1: Audit-Trail | Übersicht

5.1.1 Allgemeiner Programmablauf

Im ersten Abschnitt wird der Daemon selbst beschrieben und darauffolgend wird näher auf den API-Endpoint eingegangen.

Als Daemon dient eine Konsolenanwendung, die Zugriff auf den OpLog der entsprechenden MongoDB Instanz hat. Diese Anwendung ist eventgesteuert, das heißt sobald ein neuer Eintrag im OpLog verfügbar ist, wird die Anwendung diesen verarbeiten. Dadurch wird sichergestellt, dass neue Einträge im OpLog zeitnahe vom System verarbeitet und somit für den Audit-Trail gesichert werden. Nachdem der Daemon die OpLog Einträge verarbeitet hat, wird er diese an den API-Endpoint zur weiteren Verarbeitung senden.

Der API-Endpoint nimmt die Daten der Daemons entgegen, wird diese weiter aufbereiten und anschließend in einem Audit-Trail abspeichern. Bei der Speicherung des Audit-Trails muss sichergestellt werden, dass die Einträge in einer ”Chained Whiteness” ähnlich wie in Abschnitt 3.1.4 gesichert abgelegt werden.
Zur Überprüfung des gesamten Audit-Trails und ausgewählter einzelner Dokumente wurden Endpunkte definiert, die diese Funktionen zur manuellen Überprüfung und für eine etwaige Web-Oberfläche zur Verfügung stellen.
In Abb. 5.2 ist der Ablauf der Audit-Trail im Detail grafisch dargestellt.

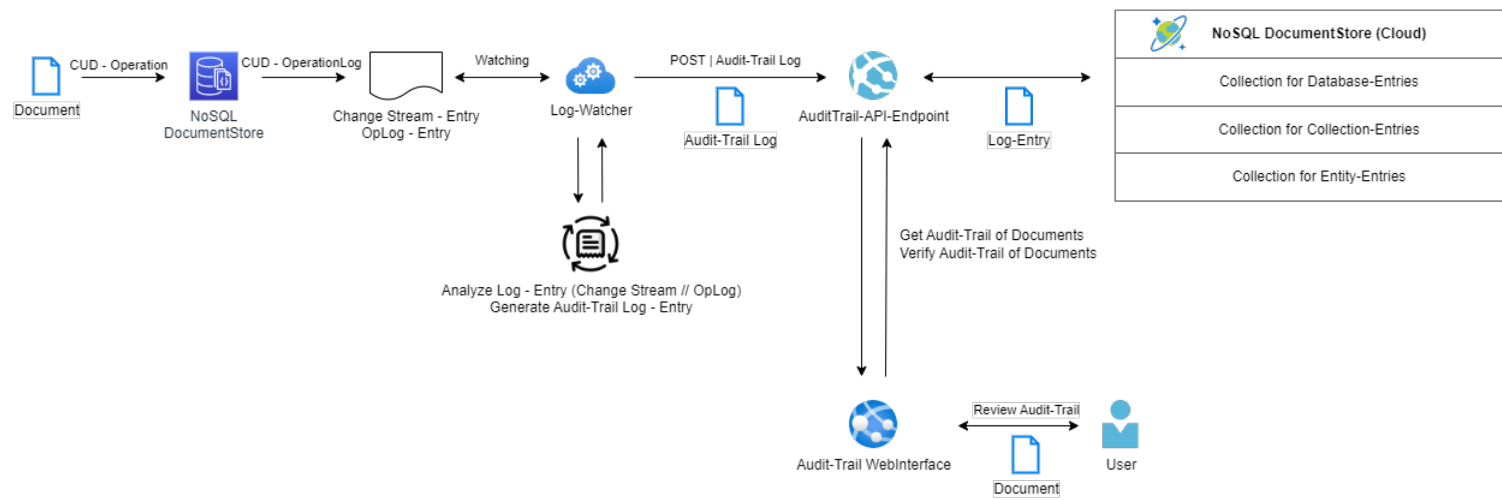

Abbildung 5.2: Audit-Trail | Detail Übersicht

5.1.2 Anbindung diverser Dokument NoSQL Datenbank-Systeme

In der Planung des Prototyps zur Erstellung eines sicheren Audit-Trails wurde berücksichtigt, dass weitere NoSQL Datenbank-Systeme an die Architektur angebunden werden können. In Abb. 5.3 ist ersichtlich, wie die Datenbank-Systeme MongoDB im Prototyp als SaaS und On-Premise angebunden sind. Darüber hinaus ist in dieser Abbildung ersichtlich, wie zum Beispiel die CosmosDB und weitere Dokumenten NoSQL Datenbank-Systeme angebunden werden können.

Diese Vielfalt an möglichen, angebundenen Dokumenten Datenbank-Systemen wird durch einen generischen Basis Daemon erreicht, der anschließend abgeleitet und je nach System angepasst werden kann.

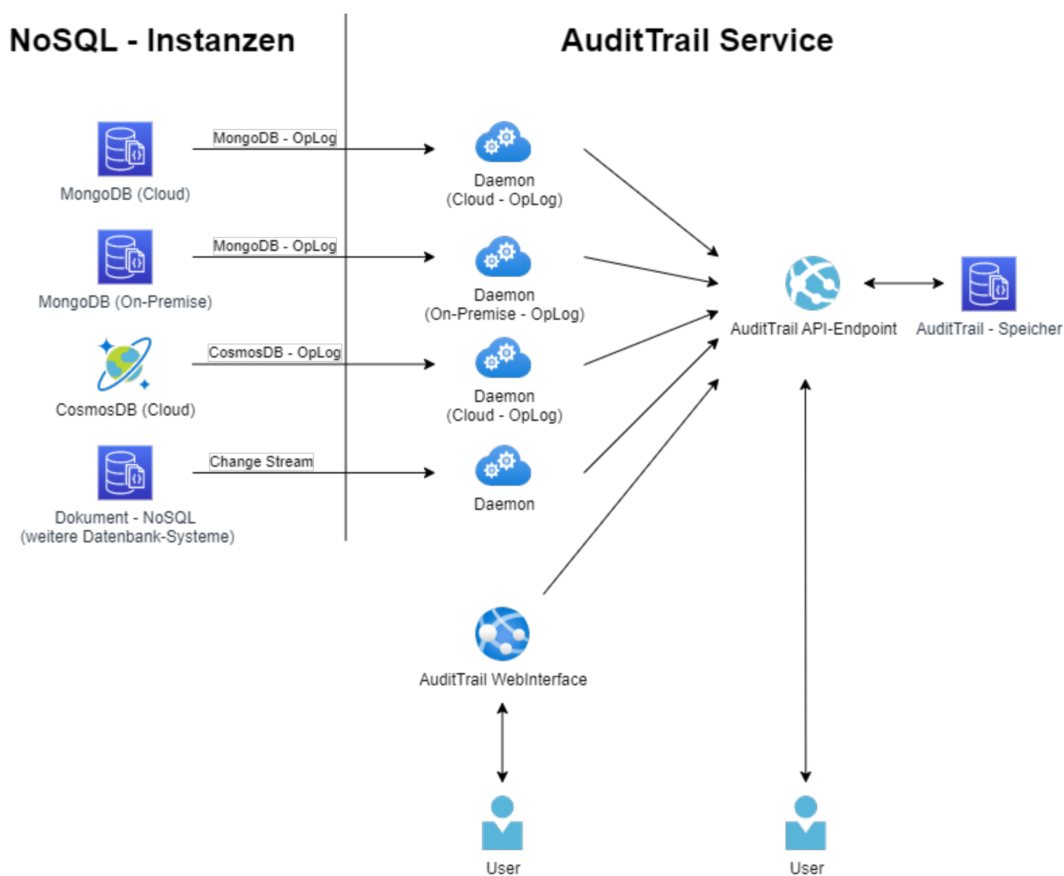

Abbildung 5.3: Audit-Trail | Anbindung weiterer Dokument NoSQL Datenbank-Systeme

5.2 Entwicklung des Prototyps

Die folgenden Abschnitte widmen sich der Entwicklung des Prototyps. Zuerst wird in Abschnitt 5.2.1 die Entwicklung des Daemons beschrieben. Anschließend wird näher auf die Entwicklung des API-Endpoints in Abschnitt 5.2.3 eingegangen.

Zur Umsetzung des Prototyps viel die Entscheidung auf das Framework .NET 6.0 [40] mit Long Term Support von Microsoft. Der Daemon basiert auf einer .NET Konsolenanwendung und der API-Endpoint basiert auf ASP.NET. Beide Anwendungen wurden in der Sprache C# [41] erstellt.
Als Grundlage wurde eine Klassenbibliothek erstellt, die anschließend im Daemon und im API-Endpoint Projekt referenziert wurden, sodass Modelle zwischen den beiden Projekten wiederverwendet werden konnten.

5.2.1 Basis Daemon

In der Planungsphase wurde entschieden, dass ein generischer Basis Daemon erstellt wird und somit weitere spezifische Daemons für weitere Implementierungen abgeleitet werden können. Im Detail wurde dies durch die Entwicklung eines Interfaces (ersichtlich in Listing 14) umgesetzt. Mithilfe dieses Interfaces wird sichergestellt, dass weitere spezialisierte Daemons diese Schnittstellen, die benötigt werden, implementieren.
Die Funktion `public bool Connect(string dbConnectionString, string database, string collectionName);` wird benötigt, damit eine Verbindung zur entsprechenden Datenbankinstanz aufgebaut werden kann. Die Funktion `public void Watch();` wird verwendet, um die Replikation-Logs auf neue Einträge zu überprüfen. Im Falle des Daemons für MongoDB wird die Collection `oplog` überwacht.
Der EventHandler `public event EventHandler<DevNoSQLDaemonEntityEventArgs> EntityEvent;` muss ausgelöst werden, wenn im Replikation-Log ein neuer Eintrag eines Dokumentes vorgefunden wird. Der EventHandler `public event EventHandler<DevNoSQLDaemonCollectionEventArgs> CollectionEvent;` muss ausgelöst werden, wenn im Replikation-Log ein neuer Eintrag einer Veränderung einer Collection vorgefunden wird. In Listing 15 und Listing 16 sind die EventArgumente der beiden EventHandler angeführt.
Im Zuge dieser Arbeit wurde auf Basis dieses definierten Interfaces ein Daemon für MongoDB implementiert. Auf die Implementierung des Daemon für MongoDB Instanzen wird in Abschnitt 5.2.2 näher eingegangen.

Als zweiter Schritt in der Entwicklung des Daemons wurde eine Klasse entwickelt, die die Kommunikation mit dem API-Endpoint implementiert. Diese Klasse konsumiert die beiden EventHandler, die zuvor genannt worden sind mit `public void SendEntityEventInformation(object? sender, DevNoSQLDaemonEntityEventArgs e)` und `public void SendCollectionEventInformation(object? sender, DevNoSQLDaemonCollectionEventArgs e)`. Um zu unterscheiden von welchem Daemon der Request abgesetzt wird, wird im Payload der Daten eine Agent-ID übermittelt.

```
1  namespace DevNoSQL.Interfaces
2  {
3      public interface IDevNoSQLDaemonClient
4      {
5          public event EventHandler<DevNoSQLDaemonEntityEventArgs>
   ↪   EntityEvent;
6          public event
   ↪   EventHandler<DevNoSQLDaemonCollectionEventArgs>
   ↪   CollectionEvent;
7          public bool Connect(string dbConnectionString, string
   ↪   database, string collectionName);
8          public void Watch();
9      }
10 }
```

Listing 14: DaemonClient | Interface

```
1 namespace DevNoSQL.Models.EventArguments
2 {
3     public class DevNoSQLDaemonCollectionEventArgs :
   ↪  DevNoSQLDaemonBaseEventArgs
4     {
5         public DevNoSQLEnums.Action Action { get; set; }
6         public string Collection { get; set; }
7         public string Database { get; set; }
8         public object Data { get; set; }
9     }
10 }
```

Listing 15: DaemonClient | DevNoSQLDaemonCollectionEventArgs

```
1 namespace DevNoSQL.Models.EventArguments
2 {
3     public class DevNoSQLDaemonEntityEventArgs :
   ↪  DevNoSQLDaemonBaseEventArgs
4     {
5         public DevNoSQLEnums.Action Action { get; set; }
6         public string EntityID { get; set; }
7         public string CollectionID { get; set; }
8         public object Data { get; set; }
9     }
10 }
```

Listing 16: DaemonClient | DevNoSQLDaemonEntityEventArgs

5.2.2 Daemon für MongoDB Instanzen

Für diese Arbeit wurde ein Daemon implementiert, der auf Events in der OpLog Collection der jeweiligen MongoDB Instanz schaut, neue Einträge auswertet und anschließend an den API-Endpoint sendet.

Der Daemon für die MongoDB Instanz führt folgenden Ablauf während seinem Betrieb aus:

1. **Verbindung zur MongoDB Instanz herstellen**
2. **Datenbank `local` auswählen**
3. **Collection `oplog.rs` als aktive Collection auswählen**
4. **Einträge ab aktuellen UnixTimeStamp abrufen**
    a) **Art der Operation aus dem Feld `op` bestimmen**
        - ”`u`” **Aktualisierung eines Dokumentes**
            i. Extrahieren des Feldes `o`
            ii. Objekt-Id aus dem Feld `o._id` extrahieren
            iii. Prüfen auf das Vorhandensein von
                - ”`o.diff.u`” ein Feld im Dokument wurde aktualisiert
                - ”`o.diff.d`” ein Feld im Dokument wurde gelöscht
                - ”`o.diff.i`” ein Feld im Dokument wurde hinzugefügt
        - ”`i`” **Erstellung eines Dokumentes**
            i. Extrahiere des Feldes `o`
            ii. Objekt-Id aus dem Feld `o._id` extrahieren
        - ”`d`” **Löschung eines Dokumentes**
            i. Extrahieren des Feldes `o`
            ii. Objekt-Id aus dem Feld `o._id` extrahieren
        - ”`c`” **Erstellung oder Löschung einer Collection**
            i. Extrahieren des Feldes `o`
            ii. Prüfen auf das Vorhandensein von
                - ”`o.create`”, Erstellen einer Collection
                - ”`o.drop`”, Löschen einer Collection

b) **Daten zur Übermittlung aufbereiten**

   i. Namespace aus dem Feld `ns` extrahieren

   ii. Datenbank aus dem Namespace eruieren

   iii. Collection aus dem Namespace extrahieren

   iv. Datum aus dem Feld `wall` eruieren

   v. TimeStamp aus dem Feld `ts` interpretieren

c) **Aufbereitete Daten an den API-Endpoint senden**

In Listing 17 ist ersichtlich, wie im Daemon auf neue Einträge in der OpLog Collection gewartet wird.

```
1  private void TailCollection()
2  {
3      BsonValue lastInsertData = BsonMinKey.Value;
4
5      var options = new FindOptions<BsonDocument>
6      {
7          CursorType = CursorType.TailableAwait
8      };
9
10     var ts = new
   ↪   BsonTimestamp((int)Helpers.UnixTimeStampHelper.GetCurrentUnixTimeStamp(),
   ↪   1);
11     BsonDocument filter = new BsonDocument("ts", new
   ↪   BsonDocument("$gte", ts));
12
13     while (true)
14     {
15         using (var cursor = _curCollection.FindSync(filter,
   ↪   options))
16         {
17             foreach (var document in cursor.ToEnumerable())
18             {
19                 lastInsertData = document["ts"];
20                 AnalyseOpLogEntry(document);
21             }
22         }
23
24         filter = new BsonDocument("ts", new BsonDocument("$gt",
   ↪   lastInsertData));
25     }
26 }
```

Listing 17: DaemonClient | TailCollection

5.2.3 API-Endpoint

In diesem Abschnitt wird auf den in dieser Arbeit geplanten und entwickelten API-Endpoint eingegangen.

Anforderungen des Audit-Trails

Folgende Anforderungen mussten bei der Planung und Entwicklung des Prototyps berücksichtigt werden:

- Sichere Verarbeitung und Speicherung der Datensätze
- Umsetzung einer "Chained Witness" zur Sicherung der Datensätze vor Manipulation
- History aller Änderungen in den Dokumenten zur Überprüfung des Audit-Trails

Allgemeiner Aufbau

Zur erwähnten Client-Server Architektur in Abschnitt 5.2 wurde in dieser Arbeit als Server eine API entwickelt, welche die Daten der Clients entgegennimmt, verarbeitet und anschließend in einer Datenbank abspeichert. Im ersten Schritt wurden Überlegungen angestrebt, wie diese Daten, welche von den Daemons übermittelt werden, sicher abgelegt werden können und somit zu einem späteren Zeitpunkt zur Verifizierung herangezogen werden können.

Als erstes wurde in der Planungsphase der Entschluss getroffen, dass die Datensätze in einer dokumentenbasierenden NoSQL Datenbank abgelegt werden. In diesem Fall fiel die Entscheidung auf CosmosDB, da sich dieses Datenbank-System einfach in .NET einbinden lässt und in einer sicheren Azure Umgebung betrieben wird. Durch die Entscheidung, dass die Datensätze selbst in einer Dokumenten NoSQL Datenbank gespeichert werden, ist es möglich, dass die zu speichernden Daten, die von den Clients gesendet werden, ohne weitere aufwendige Verarbeitung in der Datenbank abgespeichert werden können. Des Weiteren ist dadurch der Vorteil gegeben, dass man nicht an ein fixes Datenbank-Model wie bei RDBMS gebunden ist.

Zur Speicherung der Datensätze wurde als Basis ein ähnlicher Ansatz wie in der Arbeit von Frühwirt et. al [28] entwickelt. Auf die Implementierung dieses Ansatzes wird in Abschnitt 5.2.3 näher eingegangen. In Abschnitt 5.2.3 wird darüber hinaus das Datenbank-Design der API erläutert.

Die Daten der Clients werden über zwei Endpunkte mit definierten Daten-Modellen entgegengenommen. Ein Endpunkt nimmt die Daten bei Änderungen von einzelnen Dokumenten selbst entgegen. Der zweite Endpunkt nimmt Daten bei neuen Collections und bei Änderungen an den Collections entgegen.
Nach Entgegennahme der Daten an den Endpunkten, werden die Daten entsprechend analysiert und verarbeitet. Im Anschluss werden die aufbereiteten Daten sicher in einer CosmosDB abgespeichert. Auf die erwähnten Endpunkte wird in Abschnitt 5.2.3 näher eingegangen.
Die API stellt des Weiteren zwei weitere Endpunkte zur Verfügung, mit denen der Audit-Trail und einzelne Dokumente auf Richtigkeit überprüft werden können. Diese beiden Endpunkte werden anschließend in der Evaluierungsphase verwendet um den Prototypen zu evaluieren.

Damit der Aspekt von Datenschutz während der Evaluierung des Audit-Trails gewahrt bleibt, wurde eine Möglichkeit implementiert, welche die Übermittlung von Daten zwischen überprüfender Stelle und der API optimiert. Dazu wird lediglich von der überprüfenden Stelle die ID und der Hash des aktuellen Dokumentes sowie der Name der Datenbank und Collection an die API übermittelt. Der Endpunkt zur Überprüfung, ob ein Dokument valide ist, antwortet anschließend mit einem JSON-Objekt. In diesem JSON-Objekt ist der Hash-Wert des Dokumentes des Audit-Trails angeführt (Damit soll eine erweiterte Überprüfung an der überprüfenden Stelle ermöglicht werden) und darüber hinaus werden Felder übermittelt, die angeben, ob der Audit-Trail zu diesem Dokument unverändert ist, das Dokument als gelöscht eingelastet wurde und ob die Überprüfung des Hash-Wertes, in der Audit-Trail übereinstimmt. Im Abschnitt Abschnitt 5.2.3 API-Endpoint wird näher darauf eingegangen.

Datenbank-Design

Im ersten Schritt der Planung des Prototyps wurde ein Datenbank Modell ausgearbeitet, sodass damit ein sicherer Audit-Trail erstellt werden kann. Das Datenbank Modell besteht aus drei Objekten. Die Objekte werden in einer separaten Collection in einer NoSQL Datenbank gespeichert.
Die drei Objekte des Datenbank Modells bestehen aus einem Objekt für registrierte Datenbanken, ein Objekt wird verwendet, um die zu verarbeitenden Collections zu protokollieren und das dritte Objekt wird verwendet, um die entsprechenden Dokumente und deren Änderungen zu verfolgen.
In der Abb. 5.4 ist das Datenbank Modell grafisch abgebildet, inklusive der internen Referenzierungen untereinander.

Im folgenden Teil wird im Detail auf die einzelnen Objekte des Datenbank Modells eingegangen.
Als erstes wird auf das Modell für die Protokollierung der einzelnen Datenbanken eingegangen.

- **ID** | String
  Das Feld **ID** wird für die interne Referenzierung verwendet. Als ID wird eine GUID verwendet.
- **DatabaseID** | String
  In dem Feld **DatabaseID** wird der Name der Datenbank verspeichert.
- **NameSpace** | String
  Im Feld **NameSpace** wird der Namespace, der während der Verarbeitung im Daemon extrahiert wurde, geführt.
- **StorageInformation** | String
  Das Feld **StorageInformation** ist vorgesehen, um für spätere Zwecke das Quell System zu referenzieren.
- **Collections** | String[]
  Das Feld **Collections** ist ein Array von Strings. In diesem Array werden die internen IDs der Collection Objekte, die zu der jeweiligen Datenbank verfolgt werden, mitprotokolliert.
- **AgentID** | String
  Im Feld **AgentID** wird die ID des jeweiligen Agents, der die Datenbanken überwacht, hinterlegt.

Folgend wird das Modell, welches für die Protokollierung der Collections verwendet wird dargestellt.

- **ID** | String
  Das Feld **ID** wird für die interne Referenzierung der Collection verwendet. Als ID wird auch in diesem Fall eine GUID eingesetzt.
- **CollectionID** | String
  Im Feld **CollectionID** wird der Name der entsprechenden Collection abgespeichert.
- **DatabaseID** | String
  Im Feld **DatabaseID** wird die interne ID der zugehörigen Datenbank geführt.
- **NameSpace** | String
  Das Feld **NameSpace** wird dazu verwendet, um den Namespace der Collection zu protokollieren.
- **Entities** | Dictionary<string, []>
  Das Feld **Entities** ist ein Dictionary, welches als Schlüssel die ID der Entities führt und als Wert zu den Schlüsseln, werden die internen IDs der einzelnen Entity Objekten mitgeführt.

- **AgentID** | String
  Im Feld **AgentID** wird die ID des jeweiligen Agent, der die Datenbanken überwacht, hinterlegt.
- **Key** | String
  Das Feld **Key** wird dazu verwendet, um den initialen Wert für die Chained Witness pro Collection abzuspeichern.

Das Modell, welches zur Protokollierung der einzelnen Änderungen der Dokumente verwendet wird, setzt sich wie folgt zusammen:

- **ID** | String
  Das Feld **ID** wird für die interne Referenzierung der Entity verwendet. Für die ID wird wie bei den beiden vorherigen Modellen eine GUID eingesetzt.
- **EntityID** | String
  Im Feld **EntityID** wird die Objekt-ID des zu protokollierenden Dokumentes geführt.
- **CollectionID** | String
  Im Feld **CollectionID** wird die interne CollectionID der zugehörigen Collection referenziert.
- **NameSpace** | String
  Im Feld **NameSpace** wird der Namespace des Dokumentes geführt.
- **Content** | String
  Im Feld **Content** wird das tatsächliche Dokument oder die entsprechende Änderung protokolliert.
- **ActionType** | Int32
  Im **ActionType** wird die tatsächliche Art der Operation festgehalten.
- **Hash** | String
  Im Feld **Hash** wird der generierte Hash für die Chained Witness verwaltet.
- **AgentID** | String
  Im Feld **AgentID** wird die ID des jeweiligen Agents, der die Datenbanken überwacht, hinterlegt.
- **TimeStamp** | Long
  Im Feld **TimeStamp** wird der Unix Timestamp zum Zeitpunkt der Protokollierung als Long-Wert festgehalten.
- **Date** | TimeStamp
  Im Feld **Date** wird der Zeitpunkt der Protokollierung als DateTime festgehalten.

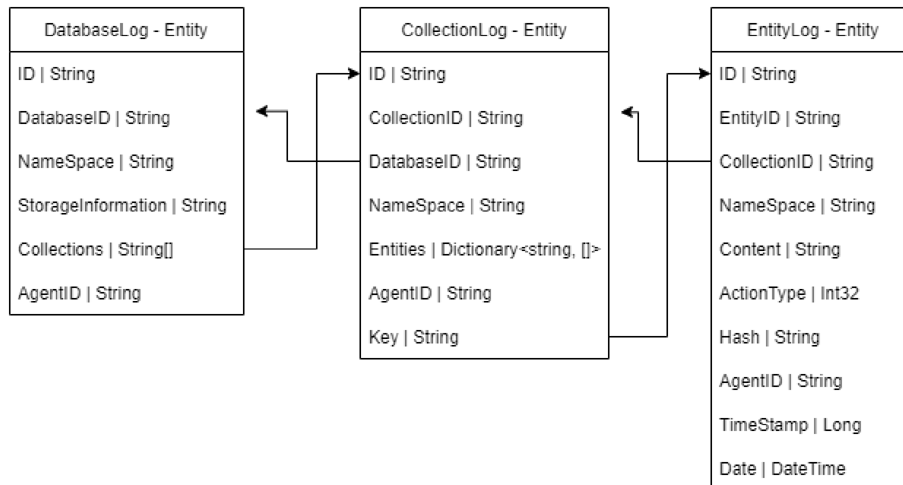

Abbildung 5.4: Audit-Trail | Datenbank Modell

In Abb. 5.5 ist ersichtlich, wie die entsprechenden Daten in den drei erwähnten Collections aufgeteilt abgespeichert werden. In der Abbildung wird dargestellt, dass zwei Datenbanken (DB1 und DB2) verfolgt werden. Die DB1 führt zwei Collections und DB2 führt eine Collection. In ”Entity-Collection” werden die Änderungen der verfolgten Dokumente protokolliert. Zur Vereinfachung der Darstellung wird in der Abbildung die Protokollierung von vier Dokumenten dargestellt. Entity-1 zeigt drei Änderungen, Entity-2 zeigt zwei Änderungen, Entity-4 zeigt ebenfalls zwei Änderungen und Entity-3 zeigt lediglich eine Änderung beziehungsweise die initiale Protokollierung des Dokumentes.

In der Collection ”Entity-Collection” ist ein Eintrag ”(DB1/Col1) Entity-1-1” angeführt. Dies symbolisiert die initiale Protokollierung des Dokumentes. Als dritter Eintrag in dieser Collection ist ein weiterer Eintrag mit dem Namen ”(DB1/Col1) Entity-1-2” ersichtlich, dieser Eintrag symbolisiert eine Änderung des Dokumentes. Zeile vier mit dem Namen ”(DB1/Col1) Entity-1-3” zeigt eine weitere Änderung dieses erwähnten Dokumentes.

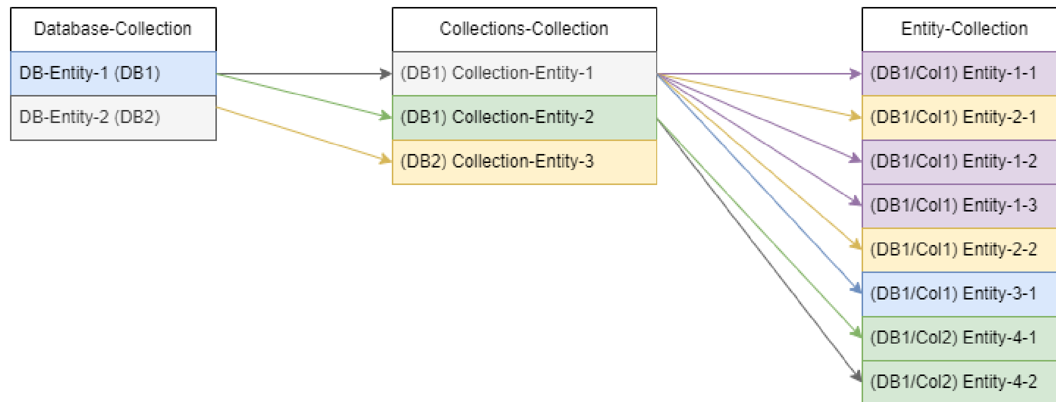

Abbildung 5.5: Audit-Trail | Beispiel Datenverteilung

Chained Witness

Wie in der Literaturanalyse in Abschnitt 3.1.4 festgestellt wurde, ist es bei diesem Ansatz wichtig, dass eine Chained Witness erstellt wird, sodass die Richtigkeit und Korrektheit der Daten gegeben ist. Nur durch die Unveränderbarkeit der Daten, die in der Audit-Trail Datenbank gespeichert werden, kann sichergestellt werden, dass der Audit-Trail einen korrekten Verlauf aller Änderungen im Quell-System protokolliert und darstellt.

Während der Planung des Prototyps wurde entschieden, dass eine Chained Witness per verfolgter Collection gepflegt wird. Damit werden alle Dokumente und deren Änderungen in einem chronologischen Verlauf als Chained Witness abgelegt.

Die Chained Witness, die im Prototyp im Rahmen dieser Arbeit in Einsatz kommt, wurde wie folgt entworfen.
Wie zuvor erwähnt, wird eine Chained Witness per verfolgter Collection des Quell-Systems verwaltet. Bei Chained Witness wird mithilfe eines Hash-Algorithmus ein Hash-Wert über die protokollierten Änderungen gebildet und der errechnete Hash-Wert wird anschließend zur Berechnung des folgenden Hash-Wertes herangezogen, sodass folgende Hash-Werte abhängig von den vorherigen Einträgen sind. In Abb. 5.6 ist eine grafische Darstellung der Chained Witness dargestellt.
Der Hash-Wert des ersten Änderungseintrages beinhaltet anstelle des Hash-Wertes des vorherigen Hash-Wertes einen Random-Seed Wert, der mithilfe der Funktion in Listing 18 generiert wird. Damit der Audit-Trail im Nachgang überprüft werden kann, ist der Random-Seed Wert auch später erforderlich, dadurch wird dieser Wert nach der Generierung im entsprechenden Collection-Objekt abgelegt.

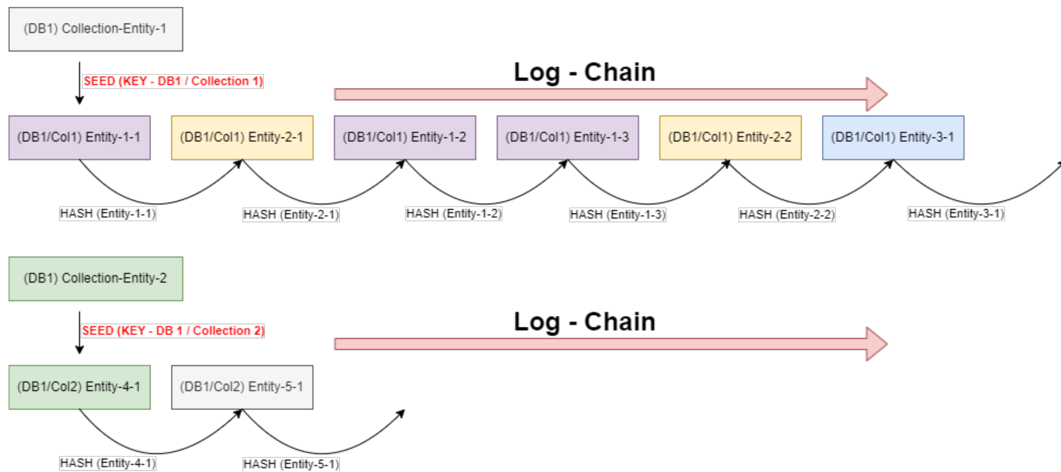

Abbildung 5.6: Audit-Trail | Beispiel Chained Witness

Als Hash-Algorithmus wird ein $SHA512$ eingesetzt, da dieser vom Bundesamt für Sicherheit in der Informationstechnik als kryptographisch stark eingestuft wird. [42] Als $TimeStamp$ wird der aktuelle UTC-Zeitstempel bei der Generierung des Eintrages für den Audit-Trail herangezogen. $Data_n$ sind die zu verarbeitenden Änderungsdaten, die am API-Endpoint entgegengenommen wurden.
Durch die Miteinbeziehung des Hash-Wertes des zuvor generierten Änderungseintrages und des aktuellen UTC-Zeitstempels wird sichergestellt, dass eine mögliche manuelle Veränderung der Chained Witness ausgeschlossen wird.
Der Hash-Wert des ersten Änderungseintrages wird wie folgt gebildet:

$$HASH_1 = SHA512(CollectionRandomSeed_n + TimeStamp + Data_1)$$

Der Hash-Wert der folgenden Änderungseinträge wird wie folgt berechnet:

$$HASH_n = SHA512(HASH_{n-1} + TimeStamp + Data_n)$$

Diese errechneten Hash-Werte werden zusätzlich zu den Objekten in der Audit-Trail abgespeichert, sodass diese Werte bei der Überprüfung der Audit-Trail herangezogen werden können.
Eine grafische Darstellung der Zusammensetzung der Hash-Werte der Audit-Trail ist in Abb. 5.7 ersichtlich.

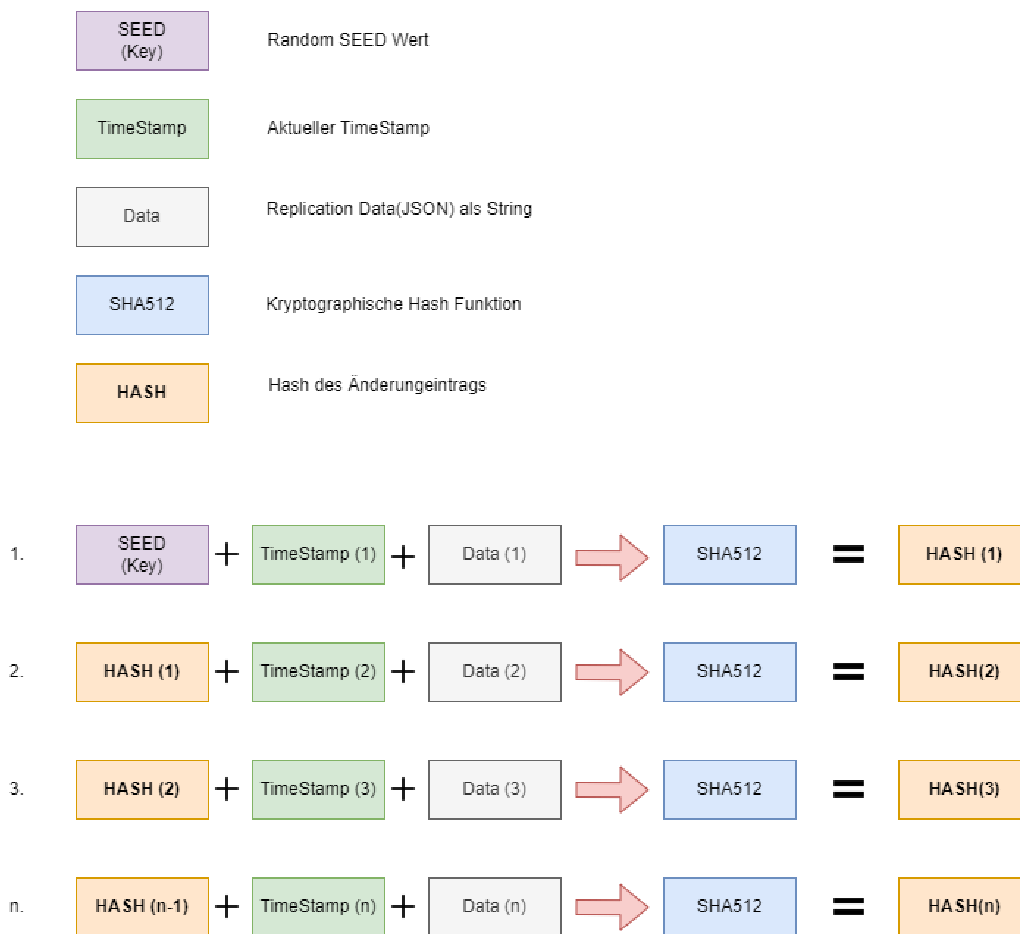

Abbildung 5.7: Audit-Trail | Hash-Funktion

```
namespace DevNoSQL.Helpers
{
    public class GuidHelper
    {
        public static Guid GetRandomNewGuid()
        {
            byte[] gBytes = new byte[16];
            RandomNumberGenerator rand =
    ↪   RandomNumberGenerator.Create();
            rand.GetBytes(gBytes);
            return new Guid(gBytes);
        }
    }
}
```

Listing 18: Audit-Trail | GetRandomNewGuid

API-Endpunkte

Der API Prototyp stellt vier Endpunkte zur Verfügung. Zwei Endpunkte werden dazu verwendet, um die Daten der Daemons zu verarbeiten und zwei Endpunkte werden zur Verfügung gestellt, damit der Audit-Trail überprüft werden kann. Die Endpunkte hinter der Route `api/Storage` stehen den Daemons zur Verfügung. Die Endpunkte hinter der Route `api/Verify`, werden zum Verifizieren des Audit-Trails herangezogen. Im folgenden Abschnitt werden im Detail auf die vier Endpunkte eingegangen.

- **api/Store/StoreCollectionInformation**

Dieser Endpunkt reagiert auf "HttpPost"-Anfragen und nimmt das JSON-Objekt, welches in Listing 19 dargestellt ist, entgegen. Mit diesem Modell werden der API die Collections, die überwacht werden, mitgeteilt. Die Daemons bereiten die Daten aus dem OpLog entsprechend diesem JSON-Objekt auf.
Das JSON-Objekt setzt sich aus folgenden Feldern zusammen:

- **nameSpace** | String
  In diesem Feld wird der entsprechende NameSpace der Collection übermittelt.
- **collectionID** | String
  In diesem Feld wir die entsprechende ID der Collection der API übermittelt.
- **databaseID** | String
  In diesem Feld übermittelt der Daemon der API die Datenbank, zu der diese Collection gehört.
- **date** | Datetime
  In diesem Feld wird das Datum und die Uhrzeit, an dem die Änderung im OpLog aufgefunden wurde protokolliert.
- **timeStamp** | Long
  In diesem Feld wird das entsprechende TimeStamp als UNIX-TimeStamp übermittelt.
- **version** | Int32
  Im Feld "Version" wird die Version des Schemas übermittelt. Im Prototypen wird die Schema-Version "0" verwendet.
- **actionType** | Int32
  In diesem Feld wird der API mitgeteilt, um welche Aktion es sich handelt. Im Falle der Collections, kann es sich um zwei Aktionen handeln.(Collection wurde hinzugefügt und Collection wurde gelöscht)
- **agendID** | String
  In diesem Feld übermittelt der Daemon seine AgentID.

```
{
  "nameSpace": "string",
  "collectionID": "string",
  "databaseID": "string",
  "date": "2023-04-01T09:02:02.477Z",
  "timeStamp": 0,
  "version": 0,
  "actionType": 0,
  "agentID": "string"
}
```

Listing 19: API-Endpoint | StoreCollectionInformation

Bei Einlangen einer Anfrage über das Hinzufügen oder Löschen einer Collection, werden die Daten wie folgt verarbeitet und in dem Audit-Trail verspeichert:

1. Entgegennahme der Daten
2. Extrahierung der AgentID
3. Extrahierung der DatenbankID
4. Zuordnung der Collection zu der entsprechenden Datenbank
5. Wird die Collection im Zuge der Erstellung einer Datenbank übermittelt, so wird in diesem Fall ein Eintrag im Datenbank-Log vorgenommen.
6. Daten im CollectionLog ablegen

- **api/Storage/StoreEntityInformation**

Dieser Endpunkt reagiert ebenfalls auf "HttpPost"-Anfragen und nimmt das JSON-Objekt, welches in Listing 20 dargestellt ist, entgegen. Mit diesem Modell werden der API Hinzufügungen und Änderungen von Dokumenten in den Collections und Dokumenten übermittelt. Wie im Fall zuvor, bereiten auch in diesem Fall die Daemons die Daten aus dem OpLog entsprechend auf und übermitteln dieses JSON-Objekt.
Das JSON-Object setzt sich aus folgenden Feldern zusammen:

- **nameSpace** | String
  In diesem Feld wird der entsprechende NameSpace der Collection übermittelt.
- **entityID** | String
  In diesem Feld übermittelt der Daemon der API die ID, des entsprechenden Dokumentes.
- **collectionID** | String
  In diesem Feld wir die entsprechende ID der Collection, in dem sich das Dokument befindet der API übermittelt.
- **date** | Datetime
  In diesem Feld wird das Datum und die Uhrzeit, an dem die Änderung im OpLog aufgefunden wurde, protokolliert.
- **timeStamp** | Long
  In diesem Feld wird das entsprechende TimeStamp als UNIX-TimeStamp übermittelt.
- **content** | String
  In diesem Feld übermittelt der Daemon der API den tatsächlichen Inhalt, der bei der Analyse des OpLogs protokolliert wurde.
- **version** | Int32
  Im Feld "Version" wird die Version des Schemas übermittelt. Im Prototyp wird die Schema-Version "0" verwendet.
- **actionType** | Int32
  In diesem Feld wird der API mitgeteilt, um welche Aktion es sich handelt. Im Falle der Entities, kann es sich um fünf Aktionen handeln. (Entity wird hinzugefügt, Entitiy wird gelöscht, Property wird hinzugefügt, Property wird aktualisiert, Property wird gelöscht)
- **agendID** | String
  In diesem Feld übermittelt der Daemon seine AgentID.

```
{
  "nameSpace": "string",
  "entityID": "string",
  "collectionID": "string",
  "date": "2022-11-21T13:07:37.039Z",
  "timeStamp": 0,
  "content": "string",
  "version": 0,
  "actionType": 0,
  "agentID": "string"
}
```

Listing 20: API-Endpoint | StoreEntityInformation

Bei Einlangen einer Anfrage über Hinzufügung eines neuen Dokumentes oder Änderung und Löschung eines Dokumentes, werden die Daten wie folgt verarbeitet und in dem Audit-Trail verspeichert:

1. Entgegennahme der Daten
2. Extrahierung der AgentID
3. Extrahierung der CollectionID
4. Extrahierung der EntityID
5. Generierung eines Hash-Wertes über die übermittelten Daten
6. Chained Witness über die entsprechende Collection erweitern
7. Daten im EntityLog ablegen

```
1 {
2   "entityID": "string",
3   "collectionID": "string",
4   "databaseID": "string",
5   "hash": "string"
6 }
```

Listing 21: API-Endpoint | Anfrage | EntityByHash

- **api/Verify/EntityByHash**

Der Endpunkt `api/Verify/EntityByHash` wird über eine ”HttpPost”-Anfrage angesprochen und liefert als Antwort ein JSON-Objekt, mit dem festgestellt werden kann, ob das zu überprüfende Dokument gültig und vertrauenswürdig ist. Um die Daten, welche zwischen der überprüfenden Stelle und der API übermittelt werden so minimal wie möglich zu halten, wird zur Verifizierung des Dokumentes lediglich ein Hash-Wert übermittelt. Am Endpunkt selbst, wird anschließend mit dem Audit-Trail die Überprüfung des Dokumentes vorgenommen.

API-Anfrage

Wie zuvor erwähnt, wird bei der ”HttpPost”-Anfrage dem Endpunkt ein JSON-Objekt, wie in Listing 21, übermittelt. Dieses Objekt setzt sich aus folgenden Feldern zusammen.

- **entityID** | String
  Im Feld ”entityID” wird die ID des zu überprüfenden Dokumentes übermittelt.
- **collectionID** | String
  Im Feld ”collectionID” wird der Name der entsprechenden Collection, in dem sich das Dokument befindet, übermittelt.
- **databaseID** | String
  Im Feld ”databaseID” wird der Name der entsprechenden Datenbank, zu der die Collection gehört, übermittelt.
- **hash** | String
  Im Feld ”hash” wird ein Hash-Wert (SHA512) des aktuellen Dokumentes aus der zu überprüfenden Datenbank übermittelt.

API-Antwort

Nach erfolgter Überprüfung antwortet die API mit einem JSON-Objekt, wie in Listing 22 dargestellt. Dieses Objekt setzt sich wie folgt zusammen:

- **hash**

  In diesem Feld wird der überprüfenden Stelle der errechnete Hash-Wert des Dokumentes in der Audit-Trail übermittelt. Mit diesem Hash-Wert soll ermöglicht werden, dass die überprüfende Stelle selbst weitere Überprüfungen durchführen kann.

- **flags.isDeleted**

  Das Feld "flags.isDeleted" spezifiziert, ob das Dokument von einem Daemon als gelöscht gemeldet wurde.

- **flags.isTrusted**

  Das Feld "flags.isTrusted" gibt an, ob der Audit-Trail zu diesem überprüften Dokument lückenlos und gültig ist.

- **flags.isValid**

  Das Feld "flags.isValid" spezifiziert, ob der Hash-Wert der übermittelt wurde, mit dem Hash-Wert der über den Audit-Trail errechnet wurde, übereinstimmt.

- **error.code**

  Das Feld "error.code" stellt bei einem Fehler einen Fehler-Code > 0 zur Verfügung.

- **error.message**

  Das Feld "error.message" stellt bei einem Fehler eine Nachricht zum aufgetretenen Fehler zur Verfügung.

```
{
  "hash": "string",
  "flags": {
    "isDeleted": true,
    "isTrusted": true,
    "isValid": true
  },
  "error": {
    "code": 0,
    "message": "string"
  }
}
```

Listing 22: API-Endpoint | Antwort | EntityByHash

- **api/Verify/GetEntityInformation**

Der Endpunkt `api/Verify/GetEntityInformation` wird ebenfalls über eine "HttpPost"-Anfrage angesprochen und liefert als Antwort ein ähnliches JSON-Objekt wie der Endpunkt `api/Verify/EntityByHash`. Der Unterschied bei diesem Endpunkt ist, dass dieses JSON-Objekt zusätzlich die Felder `data` und `history` besitzt, worin alle Änderungen, welche das entsprechende Dokument erfahren hat, aufgelistet werden und zusätzlich das aktuelle Dokument als String zur Verfügung gestellt wird.
Hierzu muss wie beim Endpunkt `api/Verify/EntityByHash` eine "HttpPost"-Anfrage mit dem JSON-Objekt wie in Listing 21 dargestellt gesendet werden. Als Antwort liefert der Endpunkt ein JSON-Objekt, welches dem JSON-Objekt wie in Listing 23 dargestellt wird, entspricht.

API-Anfrage

Auf das Modell, welches bei der Anfrage verwendet werden muss, wird in diesem Abschnitt nicht mehr eingegangen, da bereits im vorherigen Abschnitt darauf eingegangen wurde.

API-Antwort

Wie beim Endpunkt zur Verifizierung, wird bei diesem Endpunkt nach erfolgter Überprüfung ein JSON-Objekt mit dem Modell entsprechend dem Listing 23 übermittelt.

Das Modell ist wie folgt aufgebaut. Auf die Felder `hash`, `flags` und `error` wird in diesem Abschnitt nicht eingegangen, da diese mit dem Endpunkt zuvor übereinstimmen.

- **data**

  Das Feld "data" stellt das gesamte aktuelle Dokument, wie es in dem Audit-Trail aufgezeichnet wurde, der zur überprüfenden Stelle zur Verfügung. Das Dokument wird als String übermittelt.

- **history.initial**

  Im Feld "history.initial" wird das Ausgangsdokument des angeforderten Dokumentes, wie im Feld "data" zur Verfügung gestellt. Im Ausgangsdokument wird das Dokument dargestellt, so wie es das erste Mal in der Audit-Trail aufgezeichnet wurde.

- **history.historyList[].action**

  Das Feld "history.historyList[].action" gibt an, um welchen Typ an Aktion es sich handelt. Als Aktion werden folgende Status übermittelt:

  - **Insert_Entity** (5): Das Dokument wurde einer Collection hinzugefügt.
  - **Delete_Entity** (6): Das Dokument wurde aus einer Collection gelöscht.
  - **Insert_Property** (0): Dem Dokument wurde ein neues Property hinzugefügt.
  - **Update_Property** (1): In dem Dokument wurde ein Property aktualisiert.
  - **Delete_Property** (2): In dem Dokument wurde ein Property gelöscht.

- **history.historyList[].property**

  Das Feld "history.historyList[].property" spezifiziert, welches Property adressiert wurde.

- **history.historyList[].value**

  Das Feld "history.historyList[].value" liefert den aktuellen Wert zur Modifikation bei der Aufzeichnung.

- **history.historyList[].date**

  Das Feld "history.historyList[].date" stellt das Datum der angeführten Änderung zur Verfügung.

```json
{
  "hash": "string",
  "flags": {
    "isDeleted": true,
    "isTrusted": true,
    "isValid": true
  },
  "error": {
    "code": 0,
    "message": "string"
  },
  "data": "string",
  "history": {
    "initial": "string",
    "historyList": [
      {
        "action": 0,
        "property": "string",
        "value": "string",
        "date": "2023-04-08T09:17:11.308Z"
      }
    ]
  }
}
```

Listing 23: API-Endpoint | Antwort | GetEntityInformation

5.3 Evaluierung

In diesem Kapitel wird auf die Evaluierung des Prototyps eingegangen. Der entwickelte Prototyp wurde im SaaS und On-Premise Umfeld eingesetzt und untersucht.

5.3.1 On-Premise & SaaS Umgebung

Der API-Endpoint wurde in beiden Fällen in Azure als Web App Service [43] gehostet und als Backendspeicher für den Audit-Trail wurde eine CosmosDB-Instanz in derselben Subscription verwendet.

Beim Einsatz in der On-Premise Umgebung wurde eine lokale MongoDB-Instanz installiert und der Daemon am selben Rechner wie die MongoDB-Instanz ausgeführt. Mithilfe eines Skripts wurden Testdaten in der MongoDB-Instanz generiert, verändert und gelöscht, welche der Daemon über den OpLog abgegriffen und anschließend an den API-Endpoint eingemeldet hat. Der API-Endpoint hat über die eingemeldeten Daten entsprechend der jeweiligen Collections einen Audit-Trail gebildet.

Zur Evaluierung des Prototyps, im Einsatz in einer kompletten SaaS Umgebung, wurde ein MongoDB-Cluster auf der MongoDB Atlas Plattform von MongoDB Inc. [44] in Betrieb genommen. Dieser Cluster setzte sich aus drei Nodes zusammen, einen Primary-Node und zwei Secondary-Nodes. Der Daemon wurde in diesem Fall nicht auf einem lokalen Rechner betrieben, sondern ebenfalls in einer SaaS Umgebung betrieben. Der Daemon wurde in Azure als Container App [45] betrieben. Die MongoDB wurde in diesem Fall wieder mit Testdaten, welche Mithilfe einem Skript generiert wurden, befüllt, verändert und der Daemon griff die Replikationsdaten über den OpLog ab und meldete diese ebenfalls an den API-Endpoint ein.

In beiden Fällen konnte der Audit-Trail über alle Testdaten gebildet, zurückverfolgt und validiert werden. Zur einfacheren Darlegung der Evaluierung wird im folgenden Abschnitt die Erstellung des Audit-Trails, sowie die Validierung einzelner Dokumente dargestellt.

5.3.2 Beispiel Audit-Trail | Interaktion mit der MongoDB

In diesem Abschnitt wird die Erstellung, des Audit-Trails des Prototyps mit einem sehr kleinen Datensatz beispielhaft dargelegt. Zur Darstellung wurde ein sehr kleines Datenset gewählt, da damit die Nachverfolgung leichter erklärt werden kann.

Im ersten Schritt wurde eine Collection "employees" in der Datenbank "masterDB" angelegt. Diese Datenbank existiert in einer MongoDB-Instanz, die vom entwickelten Prototyp überwacht wird. Der Daemon hat die Erstellung der Collection in Listing 24 erkannt und an den API-Endpoint gemeldet. Der API-Endpoint hat anschließend einen Datenbank-Eintrag, sowie einen Collection-Eintrag für diese Collection erstellt. Die Erstellung des Datenbank-Eintrag ist in Listing 25 ersichtlich und der erstellte Collection-Eintrag ist in Listing 26 abgebildet. Im Datenbank-Eintrag ist der Name der Datenbank, der Verweis auf die Collection, die ID des Daemons (AgentID) und weitere Informationen ersichtlich. Im Collection-Eintrag ist der Name der Collection, der Name der Datenbank, der Verweis zur zugehörigen Datenbank sowie der Key für den Audit-Trail per Collection abgelegt.

```
1 Action: Create Collection
2 NS: masterDB.$cmd
3 Collection: employees
4 Date: 08.04.2023 14:37:06
5 Data: { "create" : "employees", "idIndex" : { "v" : 2, "key" : {
   ↪ "_id" : 1 }, "name" : "_id_" } }
```

Listing 24: Audit-Trail Daemon | Detektierung der Erstellung einer Collection

```
{
    "ID": "5d6d439d-13ba-4409-920b-07cfff5a955d",
    "AgentID": "c896c28a-4df1-4ef4-9984-0c4b667d0bda",
    "Collections": [
        "f96ac8d3-68d1-4aa3-b468-9ded154b8cf0"
    ],
    "DatabaseID": "masterDB",
    "Discriminator": "DevNoSQLDatabaseLog",
    "NameSpace": "masterDB.$cmd",
    "StorageInformation": "",
    "id":
    ↪  "DevNoSQLDatabaseLog|5d6d439d-13ba-4409-920b-07cfff5a955d",
    "_rid": "D1h+ANtNoMMLAAAAAAAAAA==",
    "_self":
    ↪  "dbs/D1h+AA==/colls/D1h+ANtNoMM=/docs/D1h+ANtNoMMLAAAAAAAAAA==/",
    "_etag": "\"000039e7-0000-1600-0000-64315ff70000\"",
    "_attachments": "attachments/",
    "_ts": 1680957431
}
```

Listing 25: Audit-Trail API-Endpoint | Erstellung eines Datenbank-Eintrages in der Audit-Trail

```
{
    "ID": "f96ac8d3-68d1-4aa3-b468-9ded154b8cf0",
    "AgentID": "c896c28a-4df1-4ef4-9984-0c4b667d0bda",
    "CollectionID": "employees",
    "DatabaseID": "masterDB",
    "Discriminator": "DevNoSQLCollectionLog",
    "Entities": {},
    "Key": "e6daa67d-8a76-406b-74a4-71dd50709b89",
    "NameSpace": "masterDB.$cmd",
    "id":
    ↪ "DevNoSQLCollectionLog|f96ac8d3-68d1-4aa3-b468-9ded154b8cf0",
    "_rid": "D1h+AP7jG-ELAAAAAAAAAA==",
    "_self":
    ↪ "dbs/D1h+AA==/colls/D1h+AP7jG-E=/docs/D1h+AP7jG-ELAAAAAAAAAA==/",
    "_etag": "\"000038e7-0000-1600-0000-64315ff70000\"",
    "_attachments": "attachments/",
    "_ts": 1680957431
}
```


Listing 26: Audit-Trail API-Endpoint | Speicherung der verfolgten Collection

In den folgenden Abschnitten werden die Aufzeichnungen dargestellt, die bei der Interaktion mit der MongoDB aufgezeichnet und in der Audit-Trail abgespeichert werden.

Hinzufügen eines neuen Dokumentes

Im Listing 28 ist ersichtlich, wie der Daemon das Hinzufügen eines neuen Dokumentes (Listing 27) detektiert hat. Diese Änderung wurde durch den Daemon an den API-Endpoint gemeldet. Der API-Endpoint hat die Änderung verarbeitet und in der Audit-Trail, wie in Listing 29 dargestellt, abgespeichert. Des Weiteren wurde die Liste der Entities im Collection-Eintrag um dieses Dokument erweitert. Die Erweiterung des Dokumentes ist in Listing 30 ersichtlich.

```
1 {
2    "_id":{
3       "$oid":"643160c028777846a68524b0"
4    },
5    "name":"Max Mustermann",
6    "age":"27",
7    "dept":"Software Development"
8 }
```

Listing 27: MongoDB | Initial Dokument

```
1 Action: Insert Entity
2 NS: masterDB.employees
3 ID: 643160c028777846a68524b0
4 Date: 08.04.2023 14:41:21
5 TS: 1680957681
6 Data: { "_id" : { "$oid" : "643160c028777846a68524b0" }, "name" :
   ↪ "Max Mustermann", "age" : "27", "dept" : "Software Development"
   ↪ }
```

Listing 28: Audit-Trail Daemon | Detektierung der Erstellung eines Dokumentes

```json
{
    "ID": "16406215-e671-46bb-830f-111c8f05a0a0",
    "ActionType": 5,
    "AgentID": "c896c28a-4df1-4ef4-9984-0c4b667d0bda",
    "CollectionID": "f96ac8d3-68d1-4aa3-b468-9ded154b8cf0",
    "Content": "{ \"_id\" : { \"$oid\" : \"643160c028777846a68524b0\" }, \"name\" : \"Max Mustermann\", \"age\" : \"27\", \"dept\" : \"Software Development\" }",
    "Date": "2023-04-08T14:41:21.015+02:00",
    "Discriminator": "DevNoSQLEntityLog",
    "EntityID": "643160c028777846a68524b0",
    "Hash": "e77db78c20151e41f7c929f5429aaa51cbab5dd9c406ccce0189127f9185218c",
    "NameSpace": "masterDB.employees",
    "TimeStamp": 1680957681,
    "Version": 0,
    "id": "DevNoSQLEntityLog|16406215-e671-46bb-830f-111c8f05a0a0",
    "_rid": "D1h+AKRJ+8ZWAAAAAAAAAA==",
    "_self": "dbs/D1h+AA==/colls/D1h+AKRJ+8Y=/docs/D1h+AKRJ+8ZWAAAAAAAAAA==/",
    "_etag": "\"000044e8-0000-1600-0000-643160f10000\"",
    "_attachments": "attachments/",
    "_ts": 1680957681
}
```


Listing 29: Audit-Trail API-Endpoint | Speicherung des verfolgten Dokumentes in der Audit-Trail

```
{
    ...
    "Entities": {
        "643160c028777846a68524b0": [
            "16406215-e671-46bb-830f-111c8f05a0a0"
        ]
    },
    ...
}
```

Listing 30: Audit-Trail API-Endpoint | Aufgezeichnete Logeinträge für Dokumente in der Collection ”employees”

Update eines Dokumentes

In Listing 32 ist ersichtlich, wie der Daemon das Aktualisieren eines Dokumentes festgestellt hat. Das Dokument in Listing 27 wurde um das Property in Listing 31 erweitert. Diese Änderung wurde ebenfalls durch den Daemon an den API-Endpoint gemeldet. Der API-Endpoint hat diese Information verarbeitet und der Audit-Trail hinzugefügt. Der abgelegte Audit-Trail Eintrag ist wie folgt in Listing 33 aufgebaut. Die Änderung dieses Dokumentes wurde darüber hinaus im entsprechenden Collection-Eintrag für den Audit-Trail vermerkt. Diese Vermerkung ist in Listing 34 abgebildet.

```
1 {
2    "address":{
3       "street":"Musterstraße 1",
4       "plz":"2340",
5       "city":"Musterstadt"
6    }
7 }
```

Listing 31: MongoDB | Hinzufügen des Property ”address”

```
1 Action: Insert Property
2 NS: masterDB.employees
3 ID: 643160c028777846a68524b0
4 Date: 08.04.2023 14:45:32
5 TS: 1680957932
6 Data: { "address" : { "street" : "Musterstraße 1", "plz" : "2340",
   ↪  "city" : "Musterstadt" } }
```

Listing 32: Audit-Trail Daemon | Detektierung des Hinzufügens eines weiteren Objekts eines Dokumentes

```json
{
    "ID": "03b70c6a-39b5-414b-b6d1-63f793f7e871",
    "ActionType": 0,
    "AgentID": "c896c28a-4df1-4ef4-9984-0c4b667d0bda",
    "CollectionID": "f96ac8d3-68d1-4aa3-b468-9ded154b8cf0",
    "Content": "{ \"address\" : { \"street\" : \"Musterstraße 1\", \"plz\" : \"2340\", \"city\" : \"Musterstadt\" } }",
    "Date": "2023-04-08T14:45:32.793+02:00",
    "Discriminator": "DevNoSQLEntityLog",
    "EntityID": "643160c028777846a68524b0",
    "Hash": "131fd52d75732cfe42552c9f5a636d7ff9c8155cfb765647c16edf81b4987886",
    "NameSpace": "masterDB.employees",
    "TimeStamp": 1680957932,
    "Version": 0,
    "id": "DevNoSQLEntityLog|03b70c6a-39b5-414b-b6d1-63f793f7e871",
    "_rid": "D1h+AKRJ+8ZXAAAAAAAAAA==",
    "_self": "dbs/D1h+AA==/colls/D1h+AKRJ+8Y=/docs/D1h+AKRJ+8ZXAAAAAAAAAA==/",
    "_etag": "\"00009de9-0000-1600-0000-643161ed0000\"",
    "_attachments": "attachments/",
    "_ts": 1680957933
}
```


Listing 33: Audit-Trail API-Endpoint | Speicherung der Änderung des verfolgten Dokumentes in der Audit-Trail

```
{
    ...
    "Entities": {
        "643160c028777846a68524b0": [
            "16406215-e671-46bb-830f-111c8f05a0a0",
            "03b70c6a-39b5-414b-b6d1-63f793f7e871"
        ]
    },
    ...
}
```

Listing 34: Audit-Trail API-Endpoint | Aufgezeichnete Logeinträge für Dokumente in der Collection ”employees”

Hinzufügen eines weiteren Dokumentes

In Listing 36 ist ersichtlich, wie der Daemon ein weiteres Hinzufügen eines neuen Dokumentes detektiert hat. In diesem Fall wurde das Dokument, dass in Listing 35 dargestellt ist, hinzufügt. Der API-Endpoint hat diese Änderung ebenfalls verarbeitet und im Audit-Trail hinzugefügt. Der Audit-Trail Eintrag für diese Änderung ist wie in Listing 37 dargestellt, aufgebaut. Dieses neue detektierte Dokument wurde im entsprechenden Collection-Eintrag als neue Entity eingetragen. Die Eintragung dieses neuen Dokumentes ist in Listing 38 dargestellt.

```
{
    "_id":{
        "$oid":"643162cb6bbc9b2ac9a95704"
    },
    "name":"Herbert Mustermann",
    "age":"30",
    "dept":"Finance Development",
    "address":{
        "street":"Musterstraße 2",
        "plz":"2340",
        "city":"Musterstadt"
    }
}
```

Listing 35: MongoDB | Hinzufügen eines weiteren Dokumentes

```
Action: Insert Entity
NS: masterDB.employees
ID: 643162cb6bbc9b2ac9a95704
Date: 08.04.2023 14:49:15
TS: 1680958155
Data: { "_id" : { "$oid" : "643162cb6bbc9b2ac9a95704" }, "name" :
↪ "Herbert Mustermann", "age" : "30", "dept" : "Finance
↪ Development", "address" : { "street" : "Musterstraße 2", "plz"
↪ : "2340", "city" : "Musterstadt" } }
```

Listing 36: Audit-Trail Daemon | Detektierung der Erstellung eines weiteren Dokumentes

```
1  {
2      "ID": "0e574025-3591-420b-ad66-49c529af1be0",
3      "ActionType": 5,
4      "AgentID": "c896c28a-4df1-4ef4-9984-0c4b667d0bda",
5      "CollectionID": "f96ac8d3-68d1-4aa3-b468-9ded154b8cf0",
6      "Content": "{ \"_id\" : { \"$oid\" :
   ↪   \"643162cb6bbc9b2ac9a95704\" }, \"name\" : \"Herbert
   ↪   Mustermann\", \"age\" : \"30\", \"dept\" : \"Finance
   ↪   Development\", \"address\" : { \"street\" : \"Musterstraße 2\",
   ↪   \"plz\" : \"2340\", \"city\" : \"Musterstadt\" } }",
7      "Date": "2023-04-08T14:49:15.172+02:00",
8      "Discriminator": "DevNoSQLEntityLog",
9      "EntityID": "643162cb6bbc9b2ac9a95704",
10     "Hash":
   ↪   "865d8568f91385adbf34df143437df78881d86e67d25152490052091527a7926",
11     "NameSpace": "masterDB.employees",
12     "TimeStamp": 1680958155,
13     "Version": 0,
14     "id": "DevNoSQLEntityLog|0e574025-3591-420b-ad66-49c529af1be0",
15     "_rid": "D1h+AKRJ+8ZYAAAAAAAAAA==",
16     "_self":
   ↪   "dbs/D1h+AA==/colls/D1h+AKRJ+8Y=/docs/D1h+AKRJ+8ZYAAAAAAAAAA==/",
17     "_etag": "\"0000bdea-0000-1600-0000-643162cb0000\"",
18     "_attachments": "attachments/",
19     "_ts": 1680958155
20 }
```


Listing 37: Audit-Trail API-Endpoint | Speicherung eines weiteren Dokumentes in der Audit-Trail

```
1  {
2      ...
3      "Entities": {
4          "643160c028777846a68524b0": [
5              "16406215-e671-46bb-830f-111c8f05a0a0",
6              "03b70c6a-39b5-414b-b6d1-63f793f7e871"
7          ],
8          "643162cb6bbc9b2ac9a95704": [
9              "0e574025-3591-420b-ad66-49c529af1be0"
10         ]
11     },
12     ...
13 }
```

Listing 38: Audit-Trail API-Endpoint | Aufgezeichnete Logeinträge für Dokumente in der Collection ”employees”

Löschen eines Property aus einem Dokument

In Listing 39 ist die Detektierung des Löschens eines Property aus einem Dokument ersichtlich. Der Daemon hat diese Änderung an den API-Endpoint gemeldet, welcher diese wieder in der Audit-Trail verspeichert hat. Der Audit-Trail Eintrag zu dieser Änderung ist in Listing 40 ersichtlich. Diese Änderung wurde des Weiteren zur entsprechenden Entity im Collection-Eintrag vermerkt, dies ist in Listing 41 ersichtlich.

```
1 Action: Delete Property
2 NS: masterDB.employees
3 ID: 643162cb6bbc9b2ac9a95704
4 Date: 08.04.2023 14:51:12
5 TS: 1680958272
6 Data: { "age" : false }
```

Listing 39: Audit-Trail Daemon | Detektierung des Löschens eines Property aus einem Dokument

```json
{
    "ID": "829b01ec-6156-4f44-a971-c437e97ebec7",
    "ActionType": 2,
    "AgentID": "c896c28a-4df1-4ef4-9984-0c4b667d0bda",
    "CollectionID": "f96ac8d3-68d1-4aa3-b468-9ded154b8cf0",
    "Content": "{ \"age\" : false }",
    "Date": "2023-04-08T14:51:12.472+02:00",
    "Discriminator": "DevNoSQLEntityLog",
    "EntityID": "643162cb6bbc9b2ac9a95704",
    "Hash": "de521e2ec0a466d58bdd0453f616b84dc2a6d4f8543e571746d6867e6f6b370f",
    "NameSpace": "masterDB.employees",
    "TimeStamp": 1680958272,
    "Version": 0,
    "id": "DevNoSQLEntityLog|829b01ec-6156-4f44-a971-c437e97ebec7",
    "_rid": "D1h+AKRJ+8ZZAAAAAAAAAA==",
    "_self": "dbs/D1h+AA==/colls/D1h+AKRJ+8Y=/docs/D1h+AKRJ+8ZZAAAAAAAAAA==/",
    "_etag": "\"000065eb-0000-1600-0000-643163400000\"",
    "_attachments": "attachments/",
    "_ts": 1680958272
}
```

Listing 40: Audit-Trail API-Endpoint | Speicherung der Löschung eines Property aus dem Dokument

```
{
    ...
    "Entities": {
        "643160c028777846a68524b0": [
            "16406215-e671-46bb-830f-111c8f05a0a0",
            "03b70c6a-39b5-414b-b6d1-63f793f7e871"
        ],
        "643162cb6bbc9b2ac9a95704": [
            "0e574025-3591-420b-ad66-49c529af1be0",
            "829b01ec-6156-4f44-a971-c437e97ebec7"
        ]
    },
    ...
}
```

Listing 41: Audit-Trail API-Endpoint | Aufgezeichnete Logeinträge für Dokumente in der Collection "employees"

Löschen eines Dokumentes

In Listing 42 ist die Detektierung der Löschung eines Dokumentes ersichtlich. Die Löschung des Dokumentes wird in dem Audit-Trail, wie in Listing 43 dargestellt, abgebildet. Des Weiteren wird diese Änderung im entsprechenden Collection-Eintrag aufgezeichnet und ist in Listing 44 abgebildet.

```
1 Action: Delete Entity
2 NS: masterDB.employees
3 ID: 643162cb6bbc9b2ac9a95704
4 Date: 08.04.2023 17:21:45
5 TS: 1680967305
6 Data: { "_id" : { "$oid" : "643162cb6bbc9b2ac9a95704" } }
```

Listing 42: Audit-Trail Daemon | Detektierung des Löschens eines Dokumentes

```json
{
    "ID": "6421634b-d7e8-405e-b8e7-5f05dc6ff734",
    "ActionType": 6,
    "AgentID": "c896c28a-4df1-4ef4-9984-0c4b667d0bda",
    "CollectionID": "f96ac8d3-68d1-4aa3-b468-9ded154b8cf0",
    "Content": "{ \"_id\" : { \"$oid\" :
    \"643162cb6bbc9b2ac9a95704\" } }",
    "Date": "2023-04-08T17:21:45.827+02:00",
    "Discriminator": "DevNoSQLEntityLog",
    "EntityID": "643162cb6bbc9b2ac9a95704",
    "Hash":
    "c3341d421e5d813d7651facea91e0527570788b6ee226a454743ee37e5ccf88a",
    "NameSpace": "masterDB.employees",
    "TimeStamp": 1680967305,
    "Version": 0,
    "id": "DevNoSQLEntityLog|6421634b-d7e8-405e-b8e7-5f05dc6ff734",
    "_rid": "D1h+AKRJ+8ZaAAAAAAAAAA==",
    "_self":
    "dbs/D1h+AA==/colls/D1h+AKRJ+8Y=/docs/D1h+AKRJ+8ZaAAAAAAAAAA==/",
    "_etag": "\"0100731f-0000-1600-0000-6431868a0000\"",
    "_attachments": "attachments/",
    "_ts": 1680967306
}
```


Listing 43: Audit-Trail API-Endpoint | Speicherung der Löschung eines Dokumentes

```
1  {
2      ...
3      "Entities": {
4          "643160c028777846a68524b0": [
5              "16406215-e671-46bb-830f-111c8f05a0a0",
6              "03b70c6a-39b5-414b-b6d1-63f793f7e871"
7          ],
8          "643162cb6bbc9b2ac9a95704": [
9              "0e574025-3591-420b-ad66-49c529af1be0",
10             "829b01ec-6156-4f44-a971-c437e97ebec7",
11             "6421634b-d7e8-405e-b8e7-5f05dc6ff734"
12         ]
13     },
14     ...
15 }
```

Listing 44: Audit-Trail API-Endpoint | Aufgezeichnete Logeinträge für Dokumente in der Collection ”employees”

5.3.3 Beispiel Audit-Trail | Verifizierung

In diesem Abschnitt wird auf die Validierung der zuvor gebildeten Audit-Trail eingegangen. Zur Validierung wird in diesem Fall ein Dokument mit wenigen Properties gewählt, da dieses in dieser Arbeit übersichtlicher dargestellt werden kann.

Als erstes Beispiel zur Validierung wird das Dokument mit der Objekt-ID ”643160c028777846a68524b0” herangezogen. In Listing 27 wurde dargestellt, wie das Ausgangsdokument in der Collection hinzugefügt wurde. In Listing 31 wurde diesem Dokument das Property ”address” hinzugefügt. Dadurch erfuhr dieses Dokument zwei Änderungen, welche in dem Audit-Trail detektiert wurden. Dieses Dokument ist aktuell, wie in Listing 45 dargestellt, in der Datenbank abgespeichert.

```
1  {
2     "_id":{
3        "$oid":"643160c028777846a68524b0"
4     },
5     "name":"Max Mustermann",
6     "age":"27",
7     "dept":"Software Development",
8     "address":{
9        "street":"Musterstraße 1",
10       "plz":"2340",
11       "city":"Musterstadt"
12    }
13 }
```

Listing 45: MongoDB | Dokument mit Objekt-ID ”643160c028777846a68524b0”

Verifizierung eines Dokumentes | API-Endpoint api/Verify/VerifyEntityByHash

Mit dem Endpoint `api/Verify/VerifyEntityByHash` kann, wie in Abschnitt 5.2.3 erläutert, das Dokument auf Validität überprüft werden. Hierzu muss dem API-Endpoint die Objekt-Id, Collection-Id, Datenbank-Id und der Hash des zu überprüfenden Dokumentes übermittelt werden.
Wird das Dokument in Listing 45 mit einem SHA512 gehasht, ergibt dies den folgenden Hash-Wert `2f01f6bc784bab13bb12dbd9b3c37b105213df04e381351f453c6725439eaadd`.
Die Anfrage zur Überprüfung wird wie in Listing 46 dargestellt, aufgebaut. In diesem JSON-Objekt sind die Properties entityID, collectionID, databaseID und hash ersichtlich.

```
1 {
2   "entityID": "643160c028777846a68524b0",
3   "collectionID": "employees",
4   "databaseID": "masterDB",
5   "hash":
  ↪  "2f01f6bc784bab13bb12dbd9b3c37b105213df04e381351f453c6725439eaadd"
6 }
```

Listing 46: Audit-Trail API-Endpoint | POST-Anfrage an ”api/Verify/VerifyEntityByHash”

Der API-Endpoint antwortet nach erfolgter Überprüfung mit einem JSON-Objekt wie in Listing 47 dargestellt. In diesem JSON-Objekt ist ersichtlich, dass das zu prüfende Dokument valide ist, nicht gelöscht wurde und der Audit-Trail, welcher alle Änderungen zu diesem Dokument verfolgt hat, gültig ist und somit keine nicht verfolgten Änderungen an diesem Dokument vorgenommen wurden. Das Dokument in der MongoDB entspricht somit genau dem Dokument, wie es in dem Audit-Trail mit den verfolgten Änderungen zusammengesetzt wurde.

```
{
  "hash":
↪  "2f01f6bc784bab13bb12dbd9b3c37b105213df04e381351f453c6725439eaadd",
  "flags": {
    "isDeleted": false,
    "isTrusted": true,
    "isValid": true
  },
  "error": {
    "code": 0,
    "message": ""
  }
}
```

Listing 47: Audit-Trail API-Endpoint | POST-Antwort von ”api/Verify/VerifyEntityByHash”

Verifizierung eines Dokumentes | API-Endpoint /api/Verify/GetEntityInformation

Mit dem Endpoint `/api/Verify/GetEntityInformation` kann, wie in Abschnitt 5.2.3 erläutert, das Dokument auf Validität überprüft werden. Zusätzlich wird bei der Antwort des API-Endpoints eine History aller Änderungen des Dokumentes in chronologischer Reihenfolge übermittelt. Hierzu muss, wie beim API-Endpoint zuvor, die Objekt-Id, Collection-Id, Datenbank-Id und der Hash des zu überprüfenden Dokumentes übermittelt werden.
Die Anfrage zur Überprüfung wird wie in Listing 46 dargestellt, aufgebaut.

Der API-Endpoint antwortet nach erfolgter Überprüfung mit einem JSON-Objekt, wie in Listing 48 dargestellt. In diesem JSON-Objekt ist, wie zuvor beschrieben, dass das zu prüfende Dokument valide ist, nicht gelöscht wurde und der Audit-Trail, welcher alle Änderungen zu diesem Dokument verfolgt hat, gültig ist und somit keine nicht verfolgten Änderungen an diesem Dokument vorgenommen wurden.
In Listing 48 wird zusätzlich zu den Properties, welche bei dem Endpoint zuvor übermittelt werden, ein Property "history" übermittelt. Dieses Property beinhaltet zwei Felder "initial" und "historyList". Das Property "initial" beinhaltet das zu überprüfende Dokument, wie es initial in der Datenbank erstellt wurde. Das Property "historyList" zeigt alle Änderungen eines verfolgten Dokumentes auf. In diesem Fall ist anhand des Properties "date" ersichtlich, dass das Dokument am 08.04.2023 um 14:45 verändert wurde. Im Detail wurde dem Dokument das Property "address" mit dem Objekt, welches im "value"-Property angeführt ist, ergänzt.
Die entsprechende Interpretation des Wertes des Properties "action" wurde in Abschnitt 5.2.3 aufgezeigt.

```
{
  "data": {
    "_id": {
      "$oid": "643160c028777846a68524b0"
    },
    "name": "Max Mustermann",
    "age": "27",
    "dept": "Software Development",
    "address": {
      "street": "Musterstraße 1",
      "plz": "2340",
      "city": "Musterstadt"
    }
  },
  "history": {
    "initial": {
      "_id": {
        "$oid": "643160c028777846a68524b0"
      },
      "name": "Max Mustermann",
      "age": "27",
      "dept": "Software Development",
      "address": {
        "street": "Musterstraße 1",
        "plz": "2340",
        "city": "Musterstadt"
      }
    },
    ...
```


Listing 48: Audit-Trail API-Endpoint | POST-Antwort von "/api/Verify/GetEntityInformation" | Teil 1

```
    ...
    "historyList": [
      {
        "action": 0,
        "property": "address",
        "value": {
          "street": "Musterstraße 1",
          "plz": "2340",
          "city": "Musterstadt"
        },
        "date": "2023-04-08T14:45:32.793+02:00"
      }
    ]
  },
  "hash":
  ↪ "2f01f6bc784bab13bb12dbd9b3c37b105213df04e381351f453c6725439eaadd",
  "flags": {
    "isDeleted": false,
    "isTrusted": true,
    "isValid": true
  },
  "error": {
    "code": 0,
    "message": ""
  }
}
```

Listing 49: Audit-Trail API-Endpoint | POST-Antwort von "/api/Verify/GetEntityInformation" | Teil 2

Verifizierung einer Löschung eines Dokumentes

Mit dem Endpoint `api/Verify/VerifyEntityByHash` kann, wie in Abschnitt 5.2.3 erläutert, das Dokument auf Validität überprüft werden. Des Weiteren kann dieser Endpunkt auch dazu verwendet werden, um festzustellen, ob das entsprechende Dokument gelöscht wurde.
Hierzu muss dem API-Endpoint die Objekt-Id, Collection-Id, Datenbank-Id und der Hash des zu überprüfenden Dokumentes übermittelt werden.
Wird das Dokument in Listing 35 mit einem SHA512 gehasht, ergibt dies den folgenden Hash-Wert `4fe84b501fd283fab26a93d741fadb7312b8023a5f374acdfc370f0ca88026c1`.
Der Payload der Anfrage muss, wie in den Beispielen davor, gleich aufgebaut werden.
Als Antwort liefert der Endpunkt das in Listing 50 dargestellte JSON-Objekt. In diesem JSON-Objekt ist anhand der Properties "flags.isValid" und "flags.isTrusted" ersichtlich, dass der Audit-Trail zu diesem Dokument unverändert ist und der Hash-Wert des Dokumentes mit dem Hash-Wert des Audit-Trails übereinstimmt. Darüber hinaus ist anhand dem Property "flags.isDeleted" ersichtlich, dass das Dokument in des Audit-Trails als gelöscht markiert wurde.

```
1  {
2    "hash":
   ↪  "4fe84b501fd283fab26a93d741fadb7312b8023a5f374acdfc370f0ca88026c1",
3    "flags": {
4      "isDeleted": true,
5      "isTrusted": true,
6      "isValid": true
7    },
8    "error": {
9      "code": 0,
10     "message": ""
11   }
12 }
```

Listing 50: Audit-Trail API-Endpoint | POST-Antwort von "/api/Verify/VerifyEntityByHash" | Gelöschtes Dokument

5.4 Resümee der Evaluierung

Anhand der durchgeführten Evaluierung in Abschnitt 5.3 ist ersichtlich, dass die Änderungen, welche die Dokumente erfahren haben, detektiert wurden und in dem Audit-Trail vermerkt wurden. Mithilfe der beiden Endpunkte am API-Endpoint konnten diese Dokumente erfolgreich verifiziert werden und mit dem Endpunkt `/api/Verify/GetEntityInformation` ist es möglich, alle Änderungen eines Dokumentes in chronologischer Reihenfolge zu verfolgen.
Durch die Verwendung des Hash-Wertes des Dokumentes, muss zur Verifizierung nicht das gesamte Dokument übermittelt werden. Dadurch wird vermieden, dass bei der Verifizierung relevante zu schützende Informationen über das Netz übertragen werden müssen.
Während der Entwicklung und Evaluierung des Prototyps wurde eine Azure-Subscription, worin der API-Endpoint und eine CosmosDB-Instanz betrieben wurden, mit einem Benutzer-Account verwendet. In einer produktiven Umgebung wird empfohlen, dass die Benutzer-Accounts in der Azure-Subscription mit entsprechenden Berechtigungen ausgestattet werden. Microsoft empfiehlt in diesem Fall eine feine Granulierung der Berechtigungen. Um den Audit-Trail manuell in der CosmosDB-Instanz zu inspizieren, würde in diesem Fall die Berechtigungsstufe Cosmos-DB-Account-Reader-Role [46] ausreichen. Somit würde eine weitere Sicherheitsebene im Betrieb des Audit-Trails hinzugefügt werden.

6 Conclusio und weiterführende Arbeiten

6.1 Conclusio

Das Ziel dieser Diplomarbeit war, eine Übersicht über den aktuellen Stand der Technik in der Datenbank-Forensik im Bereich RDBMS und NoSQL zu schaffen. Des Weiteren sollte eine ausgewählte Technik aus dem Bereich RDBMS adaptiert und im Bereich NoSQL seine Anwendung finden.

In Kapitel 3 wurde dabei eine Übersicht über den Stand der Technik in der Datenbank-Forensik geschaffen. In Abschnitt 3.1 wurden bestehende Ansätze und Techniken des Bereichs RDBMS aufgezeigt. In Abschnitt 3.2 wurden bestehende Ansätze und Techniken des Bereichs NoSQL aufgelistet. Anhand der durchgeführten Analyse wurde ersichtlich, dass Datenbank-Forensik unumgänglich in der forensischen Aufarbeitung von Cyber-Vorfällen ist. Im Bereich RDBMS decken die vorgestellten Methoden zur Wiederherstellung von Daten und den Tabellenstrukturen einen breiten forensischen Bereich ab. Dadurch, dass RDBMS schon länger in vielen Anwendungen verwendet werden, bietet die Literatur viele verschiedene Ansätze, wie bei einer forensischen Bearbeitung vorgegangen werden soll.
In Kapitel 4 wurde gezeigt, dass der Ansatz von Frühwirt et al. "Towards a forensic-aware database solution: Using a secured database replication protocol and transaction management for digital investigations" [28] adaptiert und für den Bereich NoSQL angewendet werden kann.
In Kapitel 5 wurde die Implementierung eines Prototyps auf Grundlage dieses adaptierten Ansatzes geplant, entwickelt und anschließend auf On-Premise und SaaS Infrastruktur evaluiert. In beiden Umgebungen konnte der Audit-Trail, nach Hinzufügung, Änderung und Löschung von Dokumenten erfolgreich verifiziert werden.
Damit während der Evaluierung von einzelnen Dokumenten in dem Audit-Trail keine gesamten Dokumente mit deren Inhalt zum entsprechenden Endpunkt gesendet werden müssen, wurde die Verifizierung so geplant und entwickelt, dass lediglich der Hash-Wert des zu überprüfenden Dokumentes übermittelt werden muss und aufgrund dieser Information die Validierung durchgeführt wird.

NoSQL Datenbank-Systeme sind aktuell im Aufschwung und finden überwiegend Verwendung in Big-Data Applikationen. [47] Dadurch bietet die Literatur nicht allzu viele technische Vorgehensweisen zu den bestehenden vier Datenbanktyp-Systemen an. Was die Literatur jedoch zu diesem Bereich bietet, sind Frameworks, die für forensische Untersuchungen von NoSQL Datenbanksysteme herangezogen werden können.

6.2 Weiterführende Arbeiten

In zukünftigen Arbeiten kann mithilfe des zur Verfügung gestellten Interfaces des Basis Daemons weitere NoSQL Dokumenten Datenbanken analysiert und angebunden werden. Des Weiteren kann eine Weboberfläche, welche die Validierung von einzelnen Dokumenten und ganzen Collections grafisch in einem Browser darstellt, geplant und umgesetzt werden.

Im Bereich der Key-Value NoSQL Datenbanken können Untersuchungen angestrebt werden, ob ein ähnlicher Ansatz für einen sicheren Audit-Trail herangezogen werden kann. Hierzu muss jedoch zuvor analysiert werden, wie die Daten aus dem jeweiligen System entsprechend abgegriffen werden können, damit diese anschließend aufbereitet an einen entsprechenden API-Endpoint gesendet werden können.

Abbildungsverzeichnis

2.1 Datenbank-Forensik | Prozess 7
2.2 NoSQL | Schematische Darstellung der Dokumenten Datenbank 12
2.3 NoSQL | Key-Value DB | Sitzungsspeicher 14

5.1 Audit-Trail | Übersicht 48
5.2 Audit-Trail | Detail Übersicht 49
5.3 Audit-Trail | Anbindung weiterer Dokument NoSQL Datenbank-Systeme 50
5.4 Audit-Trail | Datenbank Modell 61
5.5 Audit-Trail | Beispiel Datenverteilung 62
5.6 Audit-Trail | Beispiel Chained Witness 63
5.7 Audit-Trail | Hash-Funktion 64

Tabellenverzeichnis

2.1 RDBMS-Tabelle [tblCountries], Länder inklusive Währung 9
2.2 SQL-Statement Ergebnis von Listing 1 10

3.1 NoSQL Sicherheitsmerkmale [30] 30
3.2 Features, die bei der Standard-Installation aktiv sind [30] 31

List of Listings

1 SQL-Statement um Informationen zu dem CountryCode ”AT” abzurufen 9
2 DDL-Statement um Tabelle ”tblCountries” zu erstellen 10
3 DML-Statement um Eintrag ”DE” in die Tabelle ”tblCountries” hinzuzufügen 10
4 NoSQL | Dokumenten Datenbank | Beispiel Dokument 13

5 MongoDB | OpLog auf einer primär Datenbank-Instanz abrufen 38
6 MongoDB | Allgemeiner Aufbau eines OpLog-Eintrages 39
7 MongoDB | Aufbau eines OpLog-Eintrages bei der Erstellung einer Collection 40
8 MongoDB | Aufbau eines OpLog-Eintrages bei der Erstellung eines Dokumentes 42
9 MongoDB | Aufbau eines OpLog-Eintrages bei der Aktualisierung eines Dokumentes (Hinzufügen eines Feldes) 43
10 MongoDB | Aufbau eines OpLog-Eintrages bei der Aktualisierung eines Dokumentes (Aktualisierung eines Feldes) 44
11 MongoDB | Aufbau eines OpLog-Eintrages bei der Aktualisierung eines Dokumentes (Löschen eines Feldes) 44
12 MongoDB | Aufbau eines OpLog-Eintrages bei der Löschung eines Dokumentes 45
13 MongoDB | Aufbau eines OpLog-Eintrages bei der Löschung einer Collection 46

14 DaemonClient | Interface 52
15 DaemonClient | DevNoSQLDaemonCollectionEventArgs 53
16 DaemonClient | DevNoSQLDaemonEntityEventArgs 53
17 DaemonClient | TailCollection 56
18 Audit-Trail | GetRandomNewGuid 65
19 API-Endpoint | StoreCollectionInformation 67
20 API-Endpoint | StoreEntityInformation 69
21 API-Endpoint | Anfrage | EntityByHash 70

22 API-Endpoint | Antwort | EntityByHash 72
23 API-Endpoint | Antwort | GetEntityInformation 74
24 Audit-Trail Daemon | Detektierung der Erstellung einer Collection 76
25 Audit-Trail API-Endpoint | Erstellung eines Datenbank-Eintrages in der Audit-Trail 77
26 Audit-Trail API-Endpoint | Speicherung der verfolgten Collection 78
27 MongoDB | Initial Dokument 79
28 Audit-Trail Daemon | Detektierung der Erstellung eines Dokumentes 80
29 Audit-Trail API-Endpoint | Speicherung des verfolgten Dokumentes in der Audit-Trail . . . 81
30 Audit-Trail API-Endpoint | Aufgezeichnete Logeinträge für Dokumente in der Collection ”employees” 82
31 MongoDB | Hinzufügen des Property ”address” 83
32 Audit-Trail Daemon | Detektierung des Hinzufügens eines weiteren Objekts eines Dokumentes 83
33 Audit-Trail API-Endpoint | Speicherung der Änderung des verfolgten Dokumentes in der Audit-Trail 84
34 Audit-Trail API-Endpoint | Aufgezeichnete Logeinträge für Dokumente in der Collection ”employees” 85
35 MongoDB | Hinzufügen eines weiteren Dokumentes 86
36 Audit-Trail Daemon | Detektierung der Erstellung eines weiteren Dokumentes 87
37 Audit-Trail API-Endpoint | Speicherung eines weiteren Dokumentes in der Audit-Trail . . . 88
38 Audit-Trail API-Endpoint | Aufgezeichnete Logeinträge für Dokumente in der Collection ”employees” 89
39 Audit-Trail Daemon | Detektierung des Löschens eines Property aus einem Dokument 90
40 Audit-Trail API-Endpoint | Speicherung der Löschung eines Property aus dem Dokument . . 91
41 Audit-Trail API-Endpoint | Aufgezeichnete Logeinträge für Dokumente in der Collection ”employees” 92
42 Audit-Trail Daemon | Detektierung des Löschens eines Dokumentes 93
43 Audit-Trail API-Endpoint | Speicherung der Löschung eines Dokumentes 94
44 Audit-Trail API-Endpoint | Aufgezeichnete Logeinträge für Dokumente in der Collection ”employees” 95
45 MongoDB | Dokument mit Objekt-ID ”643160c028777846a68524b0” 96
46 Audit-Trail API-Endpoint | POST-Anfrage an ”api/Verify/VerifyEntityByHash” 97
47 Audit-Trail API-Endpoint | POST-Antwort von ”api/Verify/VerifyEntityByHash” 98

48 Audit-Trail API-Endpoint | POST-Antwort von "/api/Verify/GetEntityInformation" | Teil 1 . 100
49 Audit-Trail API-Endpoint | POST-Antwort von "/api/Verify/GetEntityInformation" | Teil 2 . 101
50 Audit-Trail API-Endpoint | POST-Antwort von "/api/Verify/VerifyEntityByHash" | Gelöschtes Dokument 102

Akronyme

CPU	Central Processing Unit
DBMS	Database Management System
DDL	Data Definition Language
DML	Data Manipulation Language
ERP	Enterprise Resource Planning
JSON	JavaScript Object Notation
NoSQL	Not only SQL
RAM	Random Access Memory
RDBMS	Relational Database Management System
SaaS	Software as a Service
SQL	Structured Query Language

Literatur

[1] Mario AM Guimaraes, Richard Austin und Huwida Said, „Database forensics“, in *2010 Information Security Curriculum Development Conference*, 2010, S. 62–65.

[2] Rupali Chopade und Vinod Keshaorao Pachghare, „Ten years of critical review on database forensics research“, *Digital Investigation*, Jg. 29, S. 180–197, 2019.

[3] Jongseong Yoon, Doowon Jeong, Chul-hoon Kang und Sangjin Lee, „Forensic investigation framework for the document store NoSQL DBMS: MongoDB as a case study“, *Digital Investigation*, Jg. 17, S. 53–65, 2016.

[4] Robert Luh, „Digital Forensics, Lecture, Master Course“, FH St.Pölten, 2022.

[5] Kyriacos E Pavlou und Richard T Snodgrass, „Forensic analysis of database tampering“, *ACM Transactions on Database Systems (TODS)*, Jg. 33, Nr. 4, S. 1–47, 2008.

[6] Yasmin Rasheed, Mahmoud Qutqut und Fadi Almasalha, „Overview of the current status of NoSQL database“, *Int. J. Comput. Sci. Netw. Secur*, Jg. 19, Nr. 4, S. 47–53, 2019.

[7] DB Mongo, „Top 5 considerations when evaluating NoSQL Databases“, *White Paper*, 2018.

[8] MongoDB Inc., *MongoDB*. Adresse: `https://www.mongodb.com/` (besucht am 13. 02. 2023).

[9] Microsoft, *Azure CosmosDB*. Adresse: `https://azure.microsoft.com/de-de/products/cosmos-db` (besucht am 13. 02. 2023).

[10] The Apache Software Foundation, *Apache CouchDB*. Adresse: `https://couchdb.apache.org/` (besucht am 13. 02. 2023).

[11] Amazon Web Services, *Amazon DocumentDB*. Adresse: `https://aws.amazon.com/de/documentdb/` (besucht am 13. 02. 2023).

[12] Hibernating Rhinos, *RavenDB ACID NoSQL Database*. Adresse: `https://ravendb.net/` (besucht am 13. 02. 2023).

[13] Redis Ltd., *Redis*. Adresse: `https://redis.io/` (besucht am 13. 02. 2023).

[14] Amazon Web Services, *Amazon DynamoDB*. Adresse: `https://aws.amazon.com/de/dynamodb/` (besucht am 13. 02. 2023).

[15] Amazon web Services, *Was ist eine Graphdatenbank?* Adresse: `https://aws.amazon.com/de/nosql/graph/` (besucht am 13. 02. 2023).

[16] Inc. Neo4j, *Neo4j Graph Data Platform*. Adresse: `https://neo4j.com/` (besucht am 13. 02. 2023).

[17] Amazon Web Services, *Amazon Neptune*. Adresse: `https://aws.amazon.com/de/neptune/` (besucht am 13. 02. 2023).

[18] The Apache Software Foundation, *Apache Cassandra*. Adresse: `https://cassandra.apache.org/` (besucht am 13. 02. 2023).

[19] Google, *Cloud Bigtable*. Adresse: `https://cloud.google.com/bigtable` (besucht am 13. 02. 2023).

[20] Inc. ScyllaDB, *Scylla Cloud*. Adresse: `https://www.scylladb.com/` (besucht am 13. 02. 2023).

[21] Mohamed A Mohamed, Obay G Altrafi und Mohammed O Ismail, „Relational vs. nosql databases: A survey“, *International Journal of Computer and Information Technology*, Jg. 3, Nr. 03, S. 598–601, 2014.

[22] Peter Kieseberg, Sebastian Schrittwieser und Edgar Weippl, „Structural limitations of B+-tree forensics“, in *Proceedings of the Central European Cybersecurity Conference 2018*, 2018, S. 1–4.

[23] Peter Kieseberg, Sebastian Schrittwieser, Martin Mulazzani, Markus Huber und Edgar Weippl, „Trees cannot lie: Using data structures for forensics purposes“, in *2011 European Intelligence and Security Informatics Conference*, IEEE, 2011, S. 282–285.

[24] Peter Kieseberg, Sebastian Schrittwieser, Peter Frühwirt und Edgar Weippl, „Analysis of the Internals of MySQL/InnoDB B+ Tree Index Navigation from a Forensic Perspective“, in *2019 International Conference on Software Security and Assurance (ICSSA)*, IEEE, 2019, S. 46–51.

[25] Peter Fruhwirt, Peter Kieseberg und Edgar Weippl, „Using internal MySQL InnoDB B-tree index navigation for data hiding“, in *IFIP International Conference on Digital Forensics*, Springer, 2015, S. 179–194.

[26] Peter Frühwirt, Peter Kieseberg, Sebastian Schrittwieser, Markus Huber und Edgar Weippl, „InnoDB database forensics: Enhanced reconstruction of data manipulation queries from redo logs“, *Information Security Technical Report*, Jg. 17, Nr. 4, S. 227–238, 2013.

[27] Jasmin Azemović und Denis Mušić, „Efficient model for detection data and data scheme tempering with purpose of valid forensic analysis“, in *Proc. Int. Conf. Comput. Eng. Appl.(ICCEA)*, Citeseer, 2009, S. 83–89.

[28] Peter Frühwirt, Peter Kieseberg, Katharina Krombholz und Edgar Weippl, „Towards a forensic-aware database solution: Using a secured database replication protocol and transaction management for digital investigations“, *Digital Investigation*, Jg. 11, Nr. 4, S. 336–348, 2014.

[29] MariaDB, *InnoDB - MariaDB Knowledge Base*. Adresse: `https://mariadb.com/kb/en/innodb/` (besucht am 20. 02. 2023).

[30] WK Hauger und Martin S Olivier, „NoSQL databases: forensic attribution implications“, *SAIEE Africa Research Journal*, Jg. 109, Nr. 2, S. 119–132, 2018.

[31] Jongseong Yoon und Sangjin Lee, „A method and tool to recover data deleted from a MongoDB“, *Digital Investigation*, Jg. 24, S. 106–120, 2018.

[32] Rupali Chopade und Vinod Pachghare, „A data recovery technique for Redis using internal dictionary structure“, *Forensic Science International: Digital Investigation*, Jg. 38, S. 301 218, 2021.

[33] Harmeet Kaur Khanuja und DS Adane, „A framework for database forensic analysis“, *Computer Science & Engineering: An International Journal (CSEIJ)*, Jg. 2, Nr. 3, S. 27–41, 2012.

[34] Google, *snappy | a fast compresor/decompressor*. Adresse: `https://google.github.io/snappy/` (besucht am 08. 05. 2023).

[35] MongoDB Inc., *Community Server | MongoDB*. Adresse: `https://www.mongodb.com/try/download/community` (besucht am 26. 04. 2023).

[36] MongoDB Inc., *Pricing | MongoDB*. Adresse: `https://www.mongodb.com/pricing` (besucht am 26. 04. 2023).

[37] MongoDB Inc., *Über Uns - Unsere Geschichte | MongoDB*. Adresse: `https://www.mongodb.com/de-de/company` (besucht am 11. 03. 2023).

[38] MongoDB Inc., *Glossary - MongoDB Manual*. Adresse: `https://www.mongodb.com/docs/v6.0/reference/glossary/#std-term-BSON` (besucht am 11. 03. 2023).

[39] MongoDB Inc., *Replication - MongoDB Manual*. Adresse: `https://www.mongodb.com/docs/v6.0/replication/` (besucht am 19. 05. 2023).

[40] Microsoft, *.NET | Build. Test. Deploy.* Adresse: `https://dotnet.microsoft.com/en-us/` (besucht am 20. 03. 2023).

[41] Microsoft, *C# docs | Microsoft Learn*. Adresse: `https://learn.microsoft.com/en-us/dotnet/csharp/` (besucht am 20. 03. 2023).

[42] BSI, „Kryptographische Verfahren: Empfehlungen und Schlüssellängen“, Nr. BSI TR-02102-1, 2023-01.

[43] Microsoft, *App Service | Microsoft Azure*. Adresse: `https://azure.microsoft.com/de-de/products/app-service/` (besucht am 08. 04. 2023).

[44] MongoDB Inc., *MongoDB Atlas*. Adresse: `https://www.mongodb.com/atlas` (besucht am 08. 04. 2023).

[45] Microsoft, *Azure Container Apps | Microsoft Azure*. Adresse: `https://azure.microsoft.com/de-de/products/container-apps/` (besucht am 08. 04. 2023).

[46] Microsoft, *Azure RBAC | Microsoft Learn*. Adresse: `https://learn.microsoft.com/en-us/azure/role-based-access-control/built-in-roles#cosmos-db-account-reader-role` (besucht am 27. 03. 2023).

[47] Jagdev Bhogal und Imran Choksi, „Handling big data using NoSQL“, in *2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops*, IEEE, 2015, S. 393–398.