JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von
Michaela Wakolbinger, MSc

Angefertigt am
Institut für Wirtschaftsinformatik - Data & Knowledge Engineering

Beurteiler / Beurteilerin
o. Univ.-Prof. Dipl.-Ing. Dr. techn. Michael Schrefl

Mitbetreuung
Dr. Michael Karlinger

Februar 2016

VERSCHLÜSSELUNG MITTELS QUERY-REWRITING AUF DER CLOUD-DATENBANK AMAZON DYNAMODB

Masterarbeit

zur Erlangung des akademischen Grades

Master of Science

im Masterstudium

Wirtschaftsinformatik

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenberger Straße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, am 22. Februar 2016

Michaela Wakolbinger, MSc

Kurzfassung

Cloudbasierte Datenbanken gewannen in den letzten Jahren zunehmend an Popularität. Durch diese Auslagerung der Daten in Cloud-Dienste muss auch vermehrt die Thematik des Datenschutzes und der Privatsphäre der Nutzer beachtet werden. Um den Datenschutz in einer Cloud-Datenbank zu ermöglichen, präsentiert diese Arbeit einen Ansatz zur clientseitigen Datenbankverschlüsselung. Hierbei werden die Daten verschlüsselt, bevor sie die Cloud-Datenbank geschrieben werden, wodurch Cloud-Anbieter keine Informationen über Inhalte erlangen können. In weiterer Folge wird in dieser Arbeit untersucht, inwieweit die Verschlüsselung Einfluss auf die Abfragefunktionalität hat und inwieweit etwaige Einschränkungen umgangen werden können. Dies resultiert in einer Beispiel-implementierung eines Verschlüsselungsclients für die Amazon DynamoDB, welche den Großteil der Datenbankfunktionalität unterstützt.

Abstract

The popularity of cloud based databases has increased over the last years. Due to this popularity of external storage of data in cloud services, the issues of data security and user privacy are also receiving increased attention. To provide data confidentiality in cloud databases, this thesis investigates client side database encryption. This allows encrypting the data before sending it to the cloud database, thus preventing cloud providers from accessing any unencrypted data. Moreover this thesis investigates how this encryption scheme limits the database's query functionality and attempts to alleviate some of those limitations. This results in an example implementation of an encryption client for the Amazon DyanamoDB which supports the majority of the database's functionality.

Inhaltsverzeichnis

Eidesstattliche Erklärung i

Kurzfassung ii

Abstract iii

1 Einleitung 1
- 1.1 Motivation . . . 1
- 1.2 Idee . . . 1
- 1.3 Stand der Forschung . . . 2
- 1.4 Struktur der Arbeit . . . 2

2 NoSQL-Datenbanken 3
- 2.1 Begriffsabgrenzung . . . 3
- 2.2 Relationale- und NoSQL-Datenbanken . . . 5
  - 2.2.1 CAP-Theorem . . . 7
  - 2.2.2 Konsistenzmodelle . . . 8
- 2.3 NoSQL-Datenbankarten . . . 9
  - 2.3.1 Column-Family-Systeme . . . 10
  - 2.3.2 Document-Stores . . . 10
  - 2.3.3 Graphdatenbanken . . . 11
  - 2.3.4 Key-Value-Systeme . . . 12
- 2.4 Amazon und NoSQL . . . 12
- 2.5 Amazon Dynamo . . . 13
  - 2.5.1 Abfragefunktionalität . . . 14
  - 2.5.2 Skalierbarkeit . . . 14
  - 2.5.3 Konsistenzmodell . . . 16
- 2.6 Amazon DynamoDB . . . 16
  - 2.6.1 Datenmodell . . . 18
  - 2.6.2 Datenbankindizes . . . 22
  - 2.6.3 Datenbankbeschränkungen . . . 23
  - 2.6.4 Datenbankoperationen . . . 23
  - 2.6.5 Abfragefunktionalität . . . 27

2.6.6 Datenkonsistenz . . . 32
2.6.7 Zusammenfassung . . . 33

3 Verschlüsselung 34
3.1 Verschlüsselungstechniken . . . 34
3.1.1 Symmetrische Verschlüsselung . . . 34
3.1.2 Asymmetrische Verschlüsselung . . . 36
3.1.3 Homomorphe Verschlüsselung . . . 36
3.1.4 Kryptographische Hash-Funktionen . . . 37
3.2 Verschlüsselung von Datenbanken . . . 37
3.2.1 Vergleichskriterien . . . 37
3.2.2 Oracle Database . . . 41
3.2.3 Microsoft SQLServer . . . 43
3.2.4 CryptDB . . . 44
3.2.5 Bucketing . . . 45
3.2.6 Structrue Preserving Database Encryption Scheme . . . 47
3.2.7 Hierarchically Derived Symmetric Encryption . . . 50
3.2.8 Searchable Symmetric Encryption . . . 52
3.2.9 Zusammenfassung Verschlüsselung von Datenbanken . . . 53

4 Der Verschlüsselungsclient für die DynamoDB 56
4.1 Konzeption . . . 57
4.1.1 Kommunikation . . . 57
4.1.2 Verschlüsselung der Daten und Indizes . . . 58
4.1.3 Unterstützung von Datenbankoperationen . . . 61
4.1.4 Unterstützung von Datenbankabfragen . . . 63
4.2 Architektur . . . 71
4.3 Zusammenfassung . . . 73

5 Implementierung des Verschlüsselungsclients 76
5.1 Data-Dictionary . . . 76
5.2 Crypto-Config . . . 77
5.2.1 Konfiguration der Blockverschlüsselung . . . 78
5.2.2 Konfiguration des Initialisierungsvektors . . . 78
5.2.3 Konfiguration der Schlüsselerstellung . . . 78
5.2.4 Konfiguration von Scan-Operationen . . . 79
5.2.5 Konfiguration der Order-Preserving-Encryption . . . 79
5.3 Encrypted-Client-Core . . . 80
5.4 Crypto-Modul . . . 83
5.5 Index-Modul . . . 84
5.5.1 Bucketing-Index . . . 84
5.5.2 Verschlüsselter invertierter Index . . . 86

6 Laufzeittests **92**
6.1 Testablauf . . . 92
6.1.1 Allgemeiner Ablauf . . . 92
6.1.2 Messung der Ausführungszeiten . . . 94
6.2 Daten und Datenaufbau . . . 94
6.3 Testkonfiguration . . . 96
6.3.1 Konfiguration der Amazon DynamoDB-Datenbank . . . 96
6.3.2 Konfiguration des Verschlüsselungsclients . . . 96
6.3.3 Konfiguration der Testdurchführung . . . 97
6.4 Ergebnisse . . . 101
6.4.1 Testfall 1: Standardbetrieb . . . 101
6.4.2 Testfall 2: Verschlüsselungsclient unter Einsatz des Bucketing-Index 107
6.4.3 Testfall 3: Verschlüsselungsclient unter Einsatz des invertierten Index . . . 114
6.4.4 Fazit . . . 120

7 Zusammenfassung **122**

Quellenverzeichnis **124**
Literatur . . . 124

A Appendix **129**
A.1 Entwicklung . . . 129
A.2 Installation . . . 129
A.3 Codestruktur . . . 130

Kapitel 1

Einleitung

1.1 Motivation

Das Interesse an Cloud-Datenbanken ist in den letzten Jahren gestiegen. Auch Unternehmen wie Microsoft und Amazon investieren in cloud-basierte Dienste und ihre Rechenzentren [41]. Das steigende Angebot wird von zahlreichen Unternehmen in Anspruch genommen, da sich diese nicht mehr um laufende Kosten wie Anschaffung, Betrieb und Wartung kümmern müssen. Dies erlaubt Unternehmen sich stärker auf ihre Kernkompetenzen zu orientieren durch die Auslagerung von Rechen- und Dateneinheiten. Nichtsdestotrotz ist einer der großen Nachteile von Cloud-Diensten, dass Daten auf externen Servern ausgelagert werden. Hier stellen sich im Besonderen die folgenden Fragen: Wie sicher sind die Daten auf diesen Servern? Wo befinden sich diese Server und wer hat auf diese Daten Zugriff? Die Zugriffsthematik wurde durch die Aufdeckungen bzgl. Abhörung und geheimdienstlicher Überwachung von Unternehmen und deren Mitarbeitern durch Edward Snowden erneut ins Zentrum der Medien gerückt.

Eine mögliche Lösung, um diesen Problemen entgegenzuwirken, ist die Verschlüsselung von cloud-basierten Datenbanken. Daher wird in dieser Arbeit ein Verschlüsselungsclient implementiert, der Daten bereits vor dem Eintreffen in die Cloud verschlüsselt.

1.2 Idee

Die Verschlüsselung der Daten in einer Datenbank kann auf unterschiedliche Art und Weise erfolgen, wie z. B. spaltenbasiert, zellenbasiert, clientseitig oder serverseitig.

Die Art und Weise der Verschlüsselung hat großen Einfluss auf die Qualität bzw. das Niveau des Datenschutzes und der Datenbankfunktionalität wie beispielsweise die Abfragemächtigkeit.

In dieser Arbeit wird ein beispielhafter Verschlüsselungsclient für die NoSQL-Datenbank Amazon DynamoDB implementiert, der die Daten clientseitig und zellenbasiert verschlüsselt, um eine möglichst hohe Datensicherheit zu garantieren. Dieser zellenbasierte Ansatz wird als *Hierarchically Derived Symmetric Encryption* (HDSE) bezeichnet und wird im Detail in Abschnitt 3.2.7 erörtert. Er erstellt die Verschlüsselungsschlüssel der

Zellenwerte abhängig von der Verschachtelungstiefe des Datenmodells der Datenbank. Die Verschlüsselung resultiert somit in unterschiedlichen Chiffren pro Datenwert. Wodurch der Schutz der Daten verbessert wird, nachdem es erschwert wird, auf die Klartextdaten rückzuschließen.

Darüber hinaus wird in dieser Arbeit der Einfluss der Verschlüsselung auf die Datenbankfunktionalität, im Besonderen auf die Abfragefunktionalität, untersucht. Daher werden auch andere Ansätze zur Unterstützung von Abfragen auf verschlüsselte Daten untersucht.

1.3 Stand der Forschung

Aktuell existieren diverse proprietäre Ansätze zur Datenbankverschlüsselung [43, 46] wie auch zahlreiche Forschungsansätze, welche sich mit der Frage der Abfragefunktionalität [23, 36, 50] auf verschlüsselten Daten auseinander setzen. Diese Ansätze werden in dem Abschnitt 3.2 im Detail erörtert. Der in dieser Arbeit verwendete Verschlüsselungsansatz *Hierarchically Derived Symmetric Encryption* (HDSE) (siehe Abschnitt 3.2.7) wurde bereits am Institut für Wirtschaftsinformatik - Data & Knowledge Engineering der Johannes Kepler Universität Linz des öfteren für andere Datenbanken wie Cassandra [51] implementiert.

1.4 Struktur der Arbeit

Der Inhalt dieser Arbeit ist wie folgt gegliedert: Im nachfolgenden Kapitel werden die Eigenschaften von NoSQL erörtert sowie ein Einblick in das Datenbankmodel und die Datenbankfunktionalität der DynamoDB gegeben. Im darauffolgenden Kapitel werden diverse Verschlüsselungsansätze erörtert und auf Basis von definierten Kriterien wie Sicherheit und Abfragefunktionalität verglichen. Anschließend wird die Idee des HDSE-Verschlüsselungsansatzes konkret anhand der DynamoDB gezeigt und eine Beispielimplementation davon vorgestellt. Im abschließenden Kapitel werden Laufzeittests durchgeführt, um zu zeigen, in wieweit die Verschlüsslung der Daten Einfluss auf deren Laufzeit hat.

Kapitel 2

NoSQL-Datenbanken

Dieses Kapitel beschreibt die Grundlagen von NoSQL. Es werden die verschiedenen Datenbankmodelle von NoSQL erläutert und in weiterer Folge wird die Funktionalität der *Amazon DynamoDB* detailliert dargestellt.

2.1 Begriffsabgrenzung

Die Entstehung von NoSQL begann parallel zur Entwicklung von relationalen Datenbanksystemen in den 80er Jahren [32]. Zu dieser Zeit entstanden die ersten NoSQL-Systeme wie Lotus Notes und BerkleyDB [32]. Die Bezeichnung NoSQL wurde von Carlo Strozzi geprägt [32]. Er entwickelte eine Datenbank, die zwar auf einem relationalen Schema basierte, jedoch keine SQL-Schnittstelle anbot, deshalb *NoSQL*. Im Vergleich zur SQL-Schnittstelle ist die von NoSQL von Strozzi sehr einfach gehalten, da keine komplexen Abfragen auf die Daten unterstützt werden [32]. Des Weiteren distanziert sich Strozzi auf der Webseite seines Projekts vom mittlerweile verstandenen Begriff NoSQL, da dieser nichts mit seiner Implementierung zu tun hat [56]. Bis heute gibt es keine einheitliche Definition für den Begriff NoSQL. Dieser Begriff ist eher eine Marketingbezeichnung für Datenbanken, die andere Aufgabenbereiche als relationale Datenbanken abdecken. Teilweise ist mit dem Begriff NoSQL auch *Not only SQL* gemeint [32, 40]. Nichtsdestotrotz versuchen McCreary und Kelly [40] den Begriff NoSQL sehr breit gefächert zu definieren:

> „NoSQL is a set of concepts that allow the rapid and efficient processing of data sets with a focus on performance, reliability and agility [40].“

Diese Definition umfasst die Kernfaktoren, welche NoSQL beliebt gemacht haben. Große Datenmengen müssen schnell und zuverlässig verarbeitet werden und die Datenbank muss schnell auf häufige Änderungen von Datenmodell und Datensätzen (agil) reagieren können. Eine weitere Definition von NoSQL wird von Edlich et al. [31, 32] getroffen:

> „NoSQL ... being non-relational, distributed, open-source and horizontally scalable [31].“

Diese Definition spricht konkrete Charakteristiken von NoSQL an, die die Kernaussage von McCreary und Kelly unterstützen [32]:

Nicht-relational – NoSQL basiert, im Vergleich zu einer relationalen Datenbank, nicht auf einer relationalen Algebra.

Verteilt – Bei NoSQL wird der Fokus auf eine effiziente, schnelle und zuverlässige Verarbeitung von Daten gesetzt. Eine Möglichkeit, um diesen Anforderungen gerecht zu werden, ist der Einsatz von verteilten Systemen. Ein verteiltes System besteht aus mehreren Rechnerknoten, die gemeinsam einen Cluster bilden. Über diese Knoten wird die Arbeitslast (z. B. Datenbankabfragen oder Datenbestand) aufgeteilt. Die Verteilung der Daten erfolgt einerseits mittels Partitionierung[1] bzw. eine spezielle Form der Partitionierung das Sharding[2], was zu einer Steigerung der Performanz führt. Andererseits erfolgt die Verteilung auch über Replikation der Daten auf mehrere Knoten, was wiederum die Ausfallsicherheit steigert [57].

Open-Source – Open-Source wird in diversen Quellen [31, 32, 40] als Kriterium für NoSQL gesehen. Dennoch wird auch angeregt, dass man diese Aussage nicht als strikt erachten soll. Immerhin bieten diverse Firmen wie CouchDB oder Neo4j NoSQL-Datenbanken als kommerzielle Lösungen an, deren Code im Sinne von Open-Source frei zugänglich ist.

Skalierbar – Skalierbarkeit kann in vertikal und horizontal unterschieden werden. Laut der Definition von Edlich ist NoSQL für horizontale Skalierbarkeit ausgelegt. Bei dieser Skalierung, die auch als *Scale-out* bezeichnet wird, werden neue Knoten zu verteilten System hinzugefügt, um die Arbeitslast aufteilen (vgl. Partitionierung als Verteilungstechnik) zu können. Unter der vertikalen Skalierbarkeit (*Scale-up*) wird das Ersetzen oder Erweitern von Hardware-Komponenten einzelner Knoten, um deren Leistung zu steigern, verstanden [32]. Ein Ziel von NoSQL und dem Einsatz der Skalierbarkeit ist es, eine rasche und zuverlässige Verarbeitung der stark wachsende Datenbestände durch das Vergrößern des verteilten Systems sicherstellen zu können.

Edlich et al. beschreiben neben der bereits erörterten Definition auch weitere Eigenschaften von NoSQL [31, 32]:

Schemafreiheit – Eine NoSQL-Datenbank ist im Vergleich zu einer relationalen Datenbank nicht an ein vordefiniertes Schema (relationales Schema) gebunden. Daher sind NoSQL-Datenbanken schemafrei oder beinahe schemafrei.

Einfache API – NoSQL-Datenbanken bieten verglichen mit SQL üblicherweise nur stark eingeschränkte Abfragemöglichkeiten an (es sind z. B. keine Joins möglich [40]). Möchte der Entwickler komplexere Abfragen auf Daten innerhalb einer NoSQL-Datenbank durchführen, muss er diese selbst implementieren.

Bei relationalen Datenbanken kann beobachtet werden, dass Abfragen über die

[1] Partitionierung ist die logische Aufteilung von Daten auf mehrere Knoten. [59] Ein Beispiel hierzu wäre die Auslagerung von leseintensiven Daten auf einen anderen Datenbankserver.

[2] Sharding ist die Aufteilung der Daten basierend auf einen definierten Schlüssel (Shard-Key) auf mehrere Knoten [59]. Ein Beispiel hierzu wäre die Aufteilung des Inhalts einer Tabelle.

Lebensdauer einer Anwendung tendenziell immer komplexer werden. Dies ist dadurch bedingt, dass die Datenbankschemata ihrerseits immer komplexer werden bzw. Schemaänderungen in der Praxis nicht immer wohl überlegt erfolgen. Durch die Schemafreiheit und die reduzierte Abfragefunktionalität kann diese Problematik relationaler Systeme bei NoSQL Datenbanken vermieden werden.

Konsistenzmodell – Das Konsistenzmodell bei NoSQL ist anders als bei relationalen Systemen. NoSQL verwendet das *BASE*-Modell (Basically Available, Soft state, Eventual consistency).

Datenmengen – NoSQL wurde erschaffen, um große Datenmengen schnell und zuverlässig zu verarbeiten. Diese Aussage von Edlich et al. deckt sich mit der von McCreary und Kelly.

Die Agilität von NoSQL aus der Definition von McCreary und Kelly ist durch die Schemafreiheit gegeben. Die schnelle und zuverlässige Verarbeitung von großen Datenmengen wird durch den Einsatz von verteilten Systemen (Partionierung und Replikation) und BASE erreicht.

Diese hier beschriebenen Charakteristiken (nicht-relational, verteilt, Open-Source und skalierbar) und Eigenschaften (Schemafreiheit, einfache API, das Konsistenzmodell BASE und die Verarbeitung von großen Datenmengen) zeichnen eine NoSQL-Datenbank aus. Im nachfolgenden Abschnitt werden die hier aufgeführten Punkte mit relationalen Datenbanken verglichen.

2.2 Relationale- und NoSQL-Datenbanken

Die im vorherigen Abschnitt 2.1 beschriebenen Punkte, die NoSQL ausmachen, werden in diesem Abschnitt der relationalen Datenbank gegenübergestellt. Es werden die Charakteristik nicht-relational und die Eigenschaften Schemafreiheit, Abfragefunktionalität, Skalierungstechnik und Konsistenzmodell diskutiert.

Im Vergleich zu NoSQL basiert eine relationale Datenbank auf einer relationalen Algebra. In NoSQL wird auf diese verzichtet, um Lösungsansätze für Daten zu ermöglichen, deren Darstellung in einer relationalen Datenbank schwierig ist. Ein Beispiel hierzu ist die Speicherung eines Graphen [31]. Die Traversierung dieser Graphen mittels Tabellen und Joins wäre mühsam und ressourcenintensiv.

Eine relationale Datenbank erzwingt ein Schema, welches bei der Tabellenerstellung definiert werden muss. In einer relationalen Datenbank sind Zeilen, Spalten, deren Bezeichner, die Datentypen und deren Beziehung zueinander zu jeder Zeit bekannt. Dies erlaubt im Vergleich zu NoSQL die Erstellung eines umfangreicheren Datenmodells. In NoSQL wird von diesem Schemazwang abgegangen. NoSQL wurde für Anwendungen entwickelt, die oft die Datentypen, die Dateninhalte und die Beziehungen zwischen Daten ändern. Wodurch einfachere Datenmodelle resultieren. Ein Beispiel für eine Anwendung, die eine solche Flexibilität benötigt, wäre eine Webapplikation [31]. Wenn sie mit einer relationalen Datenbank zusammenarbeitet, aber dennoch ihr Schema öfter angepasst werden muss (z. B. durch ALTER TABLE...) [31], hat dies einen entscheidenden

Einfluss auf die Performanz der Webapplikation (Antwortzeiten der Datenbankabfragen oder im schlimmsten Fall steht die Datenbank in diesem Zeitraum nicht mehr zur Verfügung).

Die Abfragefunktionalität wird in einer relationalen Datenbank mithilfe von SQL realisiert [31]. SQL ist eine Sprache, die unter anderem das Einfügen, das Aktualisieren, das Löschen und das Abfragen von Daten ermöglicht. Der Zugriff auf Daten innerhalb von NoSQL wird mithilfe von APIs ermöglicht. Die Intention von NoSQL ist es, eine einfachere Abfragefunktionalität als SQL zu ermöglichen, die frei von Datenbank-Joins ist [31]. Mithilfe von SQL können sehr komplexe Abfragen formuliert werden, was bei NoSQL nicht der Fall ist. Entwickler, die komplexe Abfragen auf einer NoSQL-Datenbank durchführen möchten, müssen sich diese selbst in der Anwendung implementieren [31].

Die Verteilung des Datenbestands kann sowohl bei relationalen Datenbanken wie auch bei NoSQL erfolgen [31]. Dennoch ist dies bei relationalen Datenbanken aufwendiger durchzuführen als bei NoSQL-Datenbanken, welche von vornherein entwickelt wurden, um vergleichsweise flexibel und einfach zu skalieren. Ein Faktor, der die Skalierbarkeit bei relationalen Datenbanken erschwert, ist, dass diese immer überprüfen, ob ihr Datenbankschema (Integrität der Datenwerte und Konsistenz der Fremdschlüssel) bei meist umfangreichen Datenmodellen eingehalten wird. Dies hat ebenfalls einen Einfluss auf den Aufwand, der benötigt wird, um eine Verteilung der Daten durchzuführen und somit eine Skalierung zu erreichen.

Ein weiterer Faktor, der einen großen Einfluss hat, ist das Konsistenzmodell ACID (Atomic, Consitent, Isolated, Durable) (siehe Abschnitt 2.2.2), welches bei relationalen Datenbanken zum Einsatz kommt. Dies setzt einen konfliktfreien Datenbestand auf allen Knoten innerhalb eines verteilten Systems voraus. Diesen konfliktfreien Datenbestand zu erreichen verursacht einen erheblichen Aufwand, da die Datenbank sicherstellen muss, dass im verteilten System alle Knoten sich darüber einig sind, in welcher Reihenfolge die Schreiboperationen erfolgt sind bzw. noch zu erfolgen haben [1, 57].

Dieser konfliktfreie Datenbestand im verteilten System und die Skalierbarkeit des Systems stehen im Konflikt zueinander. Dieser Konflikt wird im CAP-Theorem (Consistency, Availablity, Partition Tolerance) formal festgehalten und näher analysiert [24, 32], worauf im nachfolgenden Abschnitt genauer eingegangen wird.

NoSQL löst diesen Konflikt mithilfe des Konsistenzmodells BASE [32]. Dieses garantiert, dass *irgendwann nach endlicher Zeit* alle Knoten im verteilten System einen aktuellen (und damit konfliktfreien) Datenbestand haben werden.

Zusammenfassend bedeutet dies, dass der Skalierungsaufwand bei relationalen Datenbanken, im Vergleich zu NoSQL wesentlich höher ist, da relationale Datenbanken wesentlich umfangreichere Datenmodelle verwalten sowie mächtigere Operationen bereitstellen. Diese Operationen umfassen komplexe SQL-Abfragen, die Abbildung von Transaktionen im Sinne von ACID auf unterschiedliche Datenbankknoten zur Erreichung eines konfliktfreien Datenbestandes und damit auch eine Verteilungstransparenz, wobei die Nutzer der Datenbank nicht die Verteilung der Daten wahrnehmen [35].

	Relationale Datenbanken	NoSQL
Relational	Ja	Nein
Schemafreiheit	Nein	Ja
Abfragefunktionalität	Abfragesprache SQL	API
Skalierbarkeit	Mühsam	Einfach
Konsistenzmodell	ACID	BASE

Tabelle 2.1: Tabelle gibt Überblick über die Unterschiede zwischen relationalen Datenbanken und NoSQL.

Ein zusammenfassender Überblick über die Unterscheide zwischen NoSQL und relationalen Datenbanken wird in Tabelle 2.1 gegeben. Im nachfolgenden Abschnitt wird zuerst das CAP-Theorem und auf dessen Basis die Konsistenzmodelle ACID und BASE in Abschnitt 2.2.2 genauer beschrieben.

2.2.1 CAP-Theorem

Das CAP-Theorem besagt, dass ein verteiltes System nie Datenkonsistenz (*Consistency*), Verfügbarkeit (*Availability*) und Ausfalltoleranz (*Partition Tolerance*) gleichzeitig erreichen kann. Nur zwei der drei Eigenschaften sind gleichzeitig möglich[25, 32]. Die drei Eigenschaften sind wie folgt zu verstehen:

Datenkonsistenz (Consistency (C)) – Innerhalb eines verteilten Datenbanksystems besitzen alle beteiligten Knoten einen konsistenten Datenbestand [32]. In anderen Worten sind sich alle Knoten, auch nach einer Änderung eines Datensatzes, darüber einig, welcher Wert für den Datensatz der derzeit gültige bzw. aktuelle ist [24]. Bei einer Leseoperation wird immer nur die aktuelle bzw. gültige Version des Datensatz zurückgeliefert [1, 32].

Verfügbarkeit (Availablity (A)) – Die Verfügbarkeit (*high availablity of data* [24]) bezeichnet die Bereitschaft der Datenbanken Benutzerabfragen (Lese- und Schreibeoperationen) zu jedem Zeitpunkt zu bearbeiten [1, 32].

Ausfalltoleranz (Partition Tolerance (P)) - Ein verteiltes System ist auch dann noch verfügbar (und verarbeitet Daten [1]), wenn es zu Netzwerkabspaltungen oder dem Ausfall einzelner Knoten kommt [24].

Die Wechselwirkung der drei Eigenschaften ist wie folgt erklärbar. Wird Datenkonsistenz in einem verteilten System verlangt und werden Anfragen erst nach dem erfolgreichen Abgleich der Daten mit allen anderen Knoten abgeschlossen, dann ist ein Ausfall (z. B. Netzwerk- oder Serverausfall) fatal. Die Anfrage würde solange warten müssen, bis der Ausfall behoben ist. Dementsprechend leidet die Verfügbarkeit der Datenbank und der Zugriff auf die Daten. Verfügbarkeit kann in so einem Fall nur erreicht werden, wenn Dateninkonsistenzen akzeptiert werden [32]. Schwächt man die Forderung nach Datenkonsistenz (C), dann kann wiederum eine bessere Verfügbarkeit und Ausfalltoleranz erreicht werden (AP). Als Lösung dieses Problems wurde für NoSQL-Datenbanken

das Konsistenzmodell BASE herangezogen [32], auf welches im nächsten Abschnitt 2.2.2 genauer eingegangen wird.

Darüber hinaus ist an dieser Stelle eine neue Publikation von Brewer zu erwähnen [24]. Darin diskutiert er die Aktualität des CAP-Theorems und erörtert, dass die Ausfalltoleranz nicht mehr ein akutes Problem innerhalb eines verteilten Systems darstellt, da Ausfälle (z. B. seitens des Netzwerkes oder der Server) nicht sehr häufig vorkommen. Die seltenen Ausfälle müssen daher rechtzeitig erkannt oder vorhergesagt werden, wodurch eine geeignete Strategie ausgeführt werden kann, um einen Ausfall des Systems zu verhindern. Brewer bezeichnet daher *2 of 3* [24] der Eigenschaften als irreführend in der Interpretation. Des Weiteren beschreibt er, dass in der klassischen Interpretation des CAP-Theorems die Netzwerklatenz nicht berücksichtigt wird. Hohe Latenz durch Replikationsprozesse reduziert die Verfügbarkeit, erhöht jedoch die Konsistenz.

Brewer beschreibt des Weiteren Strategien [24], wie bei Ausfällen die Datenkonsistenz erhalten werden kann. Eine dieser Strategien wird von Amazon implementiert. Sie findet Anwendung beim Einkaufswagen der E-Commerce-Plattform dieses Anbieters. Hierbei werden die unterschiedlichen Versionen des Einkaufswagens mithilfe einer *Union*-Operation zusammengefasst [24]. Der Nachteil dieser Strategie ist es, dass gelöschte Elemente wieder sichtbar werden können .

2.2.2 Konsistenzmodelle

ACID

In relationalen Datenbanken wird das Konsistenzmodell ACID verwendet. Dieses garantiert Datenkonsistenz durch Transaktionen, welche die ACID- Eigenschaften [1, 32, 57] erfüllen:

Atomarität (Atomic) – Die durchgeführte Transaktion bildet eine Einheit [57]. In anderen Worten wird eine Transaktion entweder vollständig durchgeführt oder nicht durchgeführt [1].

Konsistenz (Consistent) – Eine Transaktion überführt die Datenbank von einem konsistenten Zustand zum nächsten [1, 57, 61]. Zu beachten ist, dass Konsistenz im Kontext von ACID-Schemakonformität bedeutet. Dies ist eine grundsätzlich andere Konsistenz, als jene, welche im Kontext von CAP behandelt wird.

Isolation (Isolated) – Isolation bedeutet, dass sich Transaktionen nicht gegenseitig behindern [57]. Dadurch können sich gleichzeitige Schreibeoperationen (durch unterschiedliche Transaktionen) auf Daten nicht gegenseitig beeinflussen. In anderen Worten verhält sich eine Transaktion so, als ob sie die einzige ist, welche zum derzeitigen Zeitpunkt ausgeführt wird [1].

Dauerhaftigkeit (Durable) – Nach erfolgreichen Abschluss einer Transaktion sind alle durchgeführten Operationen permanent [1, 57].

Ein Nichterfüllen dieser Eigenschaften erzwingt ein Zurücksetzen aller in dieser Transaktion durchgeführten Aktionen. Edlich et al. [32] halten fest, dass die ACID-Transaktionen die Kernprinzipien von relationalen Datenbanken darstellen.

Jedoch kostet die Erfüllung von ACID Zeit. Dies ist auch einer der entscheidenden Gründe [32], warum NoSQL so beliebt geworden ist. Denn bei NoSQL steht die Verfügbarkeit und Ausfalltoleranz im Vordergrund, was in schnellere Reaktionszeiten resultiert. Dennoch ist der Einsatz von ACID und somit eine garantierte Datenkonsistenz in vielen Anwendungen (z. B. Bankanwendungen) essentiell. Im Kontext zum CAP-Theorem bedeutet dies, dass die Verfügbarkeit und Konsistenz (AC) bei ACID und relationalen Datenbanken im Vordergrund stehen. Inkonsistenzen durch Ausfälle (P) sind aufgrund der Transaktionen nicht möglich.

BASE

Um die Konflikte des CAP-Theorems bei NoSQL zu lösen, wird das Konsistenzmodell BASE (*Basically Available, Soft State, Eventually Consistent* [1, 32]) eingesetzt. Bei diesem Modell steht die Verfügbarkeit im Vordergrund [32]. Die Verfügbarkeit hat bei der klassischen Interpretation des CAP-Theorems oberste Priorität und die Konsistenz wird dieser untergeordnet. Die Ziele sind somit die Skalierbarkeit und die bessere Ausnutzung der Systemressourcen (*Performanz*) [27]. BASE führt ein neues Konsistenzverständnis ein, welches *eventual consistency* genannt wird. Datenkonsistenz bei *eventual consistency* wird irgendwann nach endlicher Zeit erreicht. Sie muss nicht unmittelbar nach dem Operationsende erfüllt sein [1, 32]. Konkret bedeutet dies: Wenn sich Daten ändern, dann garantiert die Datenbank nicht, dass mehrere gleichzeitige Lesevorgänge den aktualisierten Datensatz erhalten [61].

Eine der bekanntesten Implementierungen dieses Konsistenzverständnisses ist der *Domain Name Service* (DNS)[61]. Namensänderungen werden zu anderen DNS periodisch weiter propagiert. DNS-Clienten empfangen daher nicht unbedingt die aktuellste Adresse. Des Weiteren nennt Vogels [61] folgende mögliche Gründe für die Inkonsistenz der Daten:

- die Netzwerkausfälle,
- die hohe Netzlatenz,
- die hohe Systemlast und
- das verwendete Replikationsverfahren.

2.3 NoSQL-Datenbankarten

In [31, 32, 40] wird eine Einteilung von NoSQL in dessen bekanntesten Arten getroffen, die in den folgenden Abschnitten kurz präsentiert wird. Sie untergliedert sich in die Column-Family-Systeme, die Document-Stores, die Graph-Datenbanken sowie die Key-Value-Systeme. In dieser Arbeit liegt der Schwerpunkt auf der Amazon DynamoDB, die ein Key-Value-System ist.

2.3.1 Column-Family-Systeme

Ihre Datenstruktur ist spaltenorientiert und kann manchmal mit Excel-Tabellen verglichen werden [32]. Jedoch ist es möglich, dass innerhalb einer Spalte beliebige Attribut- und Attributwert-Paare abgelegt werden können[32]. Dies hat auch den Namen *Column-Family (CF)* geprägt [32]. Eine Weiterentwicklung dieser Art von Spalten sind die *Super-Columns*, die eine Form von Unter-Listen darstellen [32]. Ein Beispiel hierzu, basierend auf der spaltenorientierten Datenbank Apache Cassandra[3], ist in Abbildung 2.1 ersichtlich. Der *Keyspace* in diesem Beispiel ist das Schema und die *Column-Family* ist die Tabelle in Cassandra. Die Column selbst ist die Spalte (bzw. das Attribut pro Zeile) [32].

Keyspace: Music		
CF: Album		
	"Wasting Light"	"bought": "2"
CF: Year		
	"2011"	
CF: Song		
	"Wasting Light"	"played": "20"
	"Walk"	"played": "40"

Abbildung 2.1: Diese Darstellung zeigt einen beispielhaften Datensatz in Cassandra.

Weitere bekannte Vertreter dieser Art sind die Apache HBase[4] und die HyperTable[5] [32, 40].

2.3.2 Document-Stores

Document-Stores speichern hierarchisch strukturierte Daten (definierte strukturierte Dokumente [27]) direkt in die Datenbank [40]. Ein solches Dokument kann eine Datei (z. B. eine Word-Datei) sein, oder auch eine Datenstruktur wie JSON, YAML oder RDF-Dokumente [32]. Vertreter dieser Datenbanksysteme sind Apache CouchDB[6] und MongoDB[7] [32, 40]. In Abbildung 2.2 wird ein beispielhaftes Dokument dieser Datenbankart anhand der CouchDB gezeigt. Die CouchDB verwaltet ihre Daten mithilfe von JSON-Dokumenten [32]. Ein JSON-Dokument basiert auf Attributbezeichner- und Attributwert-Paaren sowie auf Listen mit beliebiger Verschachtelungstiefe [32]. Das Element `_id` wird vom Benutzer oder von der Datenbank definiert, das Element `_rev` wird

[3] Apache Cassandra - http://cassandra.apache.org/
[4] Apache HBase - http://hbase.apache.org/
[5] HyperTable - http://hypertable.org/
[6] Apache CouchDB - http://couchdb.apache.org/
[7] MongoDB - http://www.mongodb.org/

zur Konsistenzverwaltung verwendet [3].

```
{
        "_id": "Foo Fighters",
        "_rev": "1234567",

        "Song": "These Days",
        "Artist": "Foo Fighters",
         "Year": 2011,
        "Album": "Wasting Light",
        "Members": [ "Dave Grohl", "Nate Mendel", "Pat Smear",
                        "Taylor Hawkins", "Chris Shiflett" ]
}
```

Abbildung 2.2: Diese Darstellung zeigt einen Datensatz in der CouchDB.

Weitere bekannte Vertreter dieser Art sind die MongoDB[8] und Couchbase [9].

2.3.3 Graphdatenbanken

Graphdatenbanken basieren auf Graphstrukturen und sind besonders gut geeignet, um Beziehungen zwischen Elementen abzubilden [40]. Diese Datenbanken werden z. B. für die Verwaltung von Graphen im Semantic-Web eingesetzt [32]. Property-Graphen bilden die Basis für viele dieser Datenbanken [32]. Ein typischer Property-Graph ist in Abbildung 2.3 ersichtlich. Darin wird die Liederzugehörigkeit zu einem Album gezeigt. Dieses Beispiel basiert auf der Graphdatenbank Neo4j[10] [32, 40].

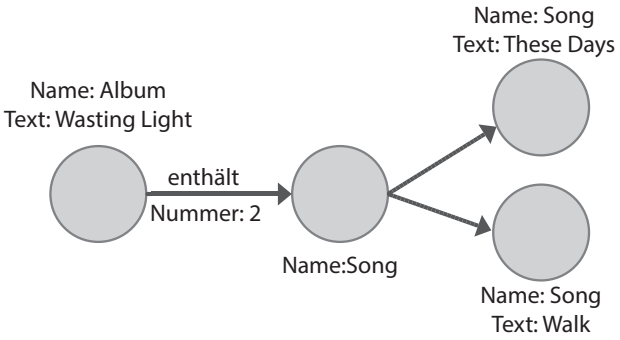

Abbildung 2.3: Diese Darstellung zeigt einen Graphen in Neo4j.

[8] MongoDB - http://www.mongodb.org/
[9] Couchbase - http://www.couchbase.com/ [32, 40]
[10] Neo4j - http://www.neo4j.org/

2.3.4 Key-Value-Systeme

Die häufigste Art von NoSQL-Datenbanken basiert auf Key-Value-Strukturen. Hier wird ein einfaches Schema angewendet, wobei der *Key* einen Wert identifiziert [40]. Dieser Key kann auch aus komplexeren Daten wie Hashes und Sets bestehen. Die Vorteile dieser Systeme sind die schnellen Zugriffszeiten und die einfache Datenverwaltung [32]. Jedoch bieten diese Datenbanken nur eine eingeschränkte Abfragefunktionalität, da nur Zugriffe über den Schlüssel (Key) möglich sind [32]. Bekannte Vertreter dieser Datenbanken sind die in dieser Arbeit eingesetzte Amazon DynamoDB[11], Redis[12] und Memcached[13] [32, 40].

Ein Beispiel für einen Datensatz innerhalb einer Key-Value Datenbank ist in 2.4 ersichtlich. Dieses Beispiel basiert auf der Amazon DynamoDB. Der Schlüssel, der diesen Datensatz identifiziert, ist das Attribut `Id`.

Key	Value
Foo Fighters_These Days	"Song": "These Days", "Artist": "Foo Fighters", "Year": 2011, "Album": "Wasting Light", "Members": ["Dave Grohl", "Nate Mendel", "Pat Smear", "Taylor Hawkins", "Chris Shiflett"]
Foo Fighters_Walk	"Song": "Walk", "Artist": "Foo Fighters", "Year": 2011, "Album": "Wasting Light", "Members": ["Dave Grohl", "Nate Mendel", "Pat Smear", "Taylor Hawkins", "Chris Shiflett"]

Abbildung 2.4: Darstellung zeigt einen Datensatz in DynamoDB.

2.4 Amazon und NoSQL

Amazon bietet eine der größten E-Commerce Plattformen weltweit an [30]. Diese Plattform muss fähig sein, Millionen von Kundenanfragen erfolgreich bearbeiten zu können. Dies verlangt, dass die Plattform auf zuverlässigen und schnellen Systemen basiert. Faktoren wie schnelle Reaktionszeiten und Verfügbarkeit sind hier von besonderer Bedeutung, da sie Auswirkungen auf das Kaufverhalten und das Kundenvertrauen haben. Um ein Wachstum der Plattform zu ermöglichen, muss auf die Skalierbarkeit geachtet

[11] Amazon DynamoDB - http://aws.amazon.com/de/dynamodb/
[12] Redis - http://redis.io/
[13] Memcached - http://memcached.org/

werden. Dazu musste Amazon neue Wege der Datenspeicherung untersuchen. Im Besonderen muss dieser Datenspeicher Dienste wie den Einkaufswagen, die Kundenvorlieben und den Produktkatalog bedienen können. Folgende Anforderungen werden von Amazon an einen Datenspeicher gestellt, um diese Dienste zu realisieren [30]:

- Sich häufig ändernde Daten müssen schnell und zuverlässig abgespeichert werden. Der Einsatz von relationalen Datenbanken wäre in diesem Szenario ineffizient, da die Verfügbarkeit und die Skalierung eingeschränkt werden würden.
- Des Weiteren benötigen zahlreiche Amazon Dienste nur anhand des Primärschlüssels Zugriff auf die Daten. Somit werden keine komplexen Abfragemechanismen wie bei relationalen Datenbanken benötigt. Eine einfache Schnittstelle ist ausreichend. Demnach wird von Amazon eine einfache API für Datenspeicherung, Verwaltung und eingeschränkte Abfragefunktionalität benötigt.
- Der von Amazon benötigte Datenspeicher muss zuverlässig und skalierbar sein.
- Eine hohe Datenkonsistenz an einen solchen Datenspeicher wird von Amazon zugunsten der bereits erwähnten Skalierbarkeit und der Verfügbarkeit aufgegeben. Primäres Ziel des Datenspeichers ist die Zufriedenheit der Plattformnutzer, welche durch einen zuverlässigen Zugriff auf die Plattform gegeben ist.

Zusammenfassend bedeutet dies, dass Amazon nach einer Lösung für diverse Dienste, wie den Einkaufswagen, strebt, welche die Eigenschaften von NoSQL benötigen. Der Datenspeicher soll flexible und sich häufig ändernde Daten verarbeiten können (nichtrelational). Zusätzlich stehen einfache Abfragen (z. B. über den Primärschlüssel) auf die Daten im Vordergrund. Darüber hinaus muss der Datenspeicher skalierbar sein und auf Ausfälle reagieren können, ohne die Funktionalität einzuschränken. Diese Aspekte wurden in Abschnitt 2.1 erörtert, in Abschnitt 2.2 diskutiert und sind Charakteristiken von NoSQL. Amazon hat auf der Basis dieser Anforderungen einen Datenspeicher namens *Dynamo* entwickelt. Auf diesem basiert eine weitere Lösung namens *DynamoDB*, welche die Komplexität von Dynamo kapselt. Im nachfolgenden Abschnitt wird zuerst der Datenspeicher Dynamo auf Basis der Eigenschaften von NoSQL erörtert. Im Abschnitt 2.6 wird die auf Dynamo aufbauende Lösung DynamoDB beschrieben.

2.5 Amazon Dynamo

Amazon Dynamo ist ein Datenspeicher, der von Amazon entwickelt wurde, um die bereits im Abschnitt 2.4 beschriebenen Anforderungen zu erfüllen. Die gespeicherten Datensätze benötigen weder komplexe Schemainformationen noch ein relationales Schema [30]. Das Datenmodell von Dynamo kann daher mit einer zero-hop *Distributed Hash Table* (DHT) [30] verglichen werden, wobei die Hashtabelle innerhalb eines verteilten Systems auf zahlreiche Knoten in einer Ringstruktur verteilt ist. Dies bedeutet, dass jeder Knoten innerhalb der DHT seine Routeninformationen selbst verwaltet, um Anfragen an den richtigen Knoten weiterzuleiten.

Die Inhalte dieses Abschnitts basieren auf der Publikation von DeCandia et al. [30]. Des Weiteren wird in den folgenden Abschnitten der Datenspeicher Dynamo anhand der

Charakteristiken Abfragefunktionalität, Skalierbarkeit und dem Konsistenzmodell von NoSQL erklärt.

2.5.1 Abfragefunktionalität

Der Zugriff auf eine Hashtabelle und auf die Daten erfolgt mithilfe einer Serviceschnittstelle, welche zwei Operationen `get` und `put` besitzt. Wird der `get`-Operation ein Schlüsselparameter übergeben, retourniert sie den gespeicherten Datensatz oder eine Liste von allen Versionen, die in Konflikt zueinander stehen. Der Schlüssel wird durch eine MD5-Hashfunktion in einen 128-bit langen Wert konvertiert. Dieser wird verwendet, um den Knoten, welcher für diesen Datensatz zuständig ist, im verteilten System zu finden. In Abbildung 2.5 wird ein Beispiel einer Hashtabelle dargestellt. Der Schlüssel wird mit der MD5-Hashfunktion auf einer Speicherstelle eines Knotens im System abgebildet. Jedes in Konflikt stehende Objekt wird mit seinem Kontext zurückgeliefert. Der Kontext beinhaltet Metainformationen über die Datenversion, welche Dynamo verwendet, um die Gültigkeit der Daten nach einer erneuten Schreibeoperation zu eruieren. Der Kontext wird zu jedem Objekt gespeichert. Bei der `put`-Operation werden der Schlüssel, der Kontext und der zu speichernde Datensatz übergeben. Die Größe der zu speichernden Datensätze liegt üblicherweise bei weniger als einem MB. Die übergebenen Daten, d.h. der Schlüssel und der Datenwert, der gespeichert werden soll, werden von Dynamo als byte Array verwaltet.

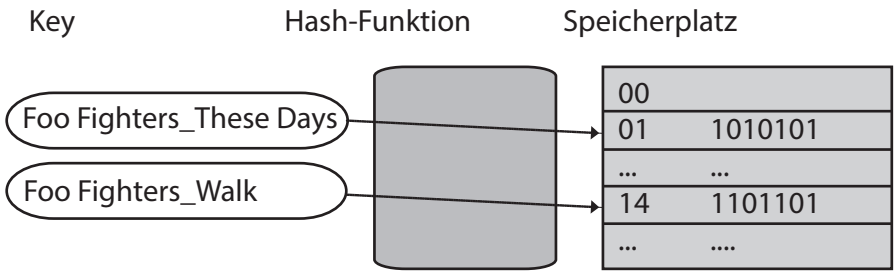

Abbildung 2.5: Darstellung zeigt eine Hashtabelle.

2.5.2 Skalierbarkeit

Dynamo verwendet diverse Strategien, um Skalierung zu ermöglichen. Bekannte Skalierungstechniken sind, wie bereits erwähnt, die Verteilung (Partionierung) und die Replikation. Positive Effekte dieser Techniken sind die Verfügbarkeit und Ausfallsicherheit des Systems. Im nachfolgenden Abschnitt wird beschrieben, wie Dynamo seine Daten innerhalb seines Systems speichert und die Replikation der Datensätze durchführt. Hier wird auch die Strategie beschrieben, wie sich Dynamo bei Ausfällen verhält.

Partitionierung und Replikation

Durch den Einsatz von *Consistent Hashing* [30] werden die Daten im Falle von Dynamo innerhalb des Systems partitioniert und somit eine Skalierbarkeit erreicht. Es kommt zur Aufteilung der Daten auf diverse Knoten innerhalb eines verteilten Systems. Die Knoten werden in einem Ringmuster angeordnet, da der Ergebnisbereich der eingesetzten Hashfunktion auf einem fixen zirkularen Bereich basiert. Um einen effizienten Zugriff auf die Daten zu ermöglichen, berechnet eine Hashfunktion einen Wert, der die Daten identifiziert (vgl. Abschnitt 2.5.1). Jedem Knoten innerhalb des Systems ist ein Schlüsselbereich zugewiesen, der seine Position im Ring definiert. Darüber hinaus dupliziert jeder der Knoten seine Schlüssel auf seinen nachfolgenden, dies erhöht die Ausfallsicherheit. Dadurch wird zusätzlich erkannt, ob ein neuer Knoten hinzugefügt oder ein vorhandener entfernt wird.

Zusätzlich verteilt Amazon die physischen Knoten des Systems innerhalb verschiedener Rechenzentren und Racks um ortsbedingte Ausfälle zu vermeiden.

Verarbeiten von Ausfällen

Um die Verfügbarkeit des Datenspeichers während Fehlersituationen wie Knotenausfällen zu garantieren, verwendet Dynamo eine adaptierte Version des Quorum-Protokolls. Mit diesem Protokoll können Lese- und Schreibeoperationen trotz Knotenausfällen propagiert werden. Das Quorum-Protokoll führt die Datenreplikation konsistent über mehrere verteilte Knoten durch, indem es von jedem für den Schlüsselbereich zuständigem Knoten seine Zustimmung zur Durchführung der Replikation einholt [57]. Dynamo setzt dazu das sogenannte *Sloppy Quorum* ein, d.h. es stimmt Operationen mit „gesunden“ Knoten ab.

Im Falle eines temporären Knotenfehlers oder Netzwerkausfalls wendet die Datenbank die sogenannte *Hinted Handoff*-Strategie an. Wenn der Knoten selbst nicht antwortet, antworten die Nachbarknoten auf die Anfrage. Der Nachbarknoten speichert daher ein sogenanntes *Hinted Replica* ab, welches jene Daten beinhaltet, die für den ausgefallenen Knoten bestimmt waren. Ist ein ausgefallener Knoten wieder verfügbar, versucht dieser sofort auf Anfragen zu antworten und die Hinted Replica Daten zu synchronisieren. Edlich et al. [32] bezeichnen dies als Selbstheilungsprozess. Sloppy Quorum und Hinted Handoff garantieren somit auch bei Ausfällen eine hohe Verfügbarkeit und eine dauerhafte Ausführung der Datenreplikation (Durability). Im Falle eines permanenten Ausfalles (und auch der gespeicherten Hinted Replica Data) ist ein Hinted Handoff jedoch nicht möglich. Die Datenbank verwendet daher Merkle-Bäume, um Inkonsistenzen innerhalb der replizierten Daten im Cluster schnell zu erkennen und die Anzahl der zu übertragenden Daten zu reduzieren. Ein Merkle-Baum ist eine Baumstruktur bestehend aus Hash-Werten, wobei die Blätter des Baumes Hash-Werte von Daten beinhalten. Höher geordnete Knoten im Baum beinhalten die Hash-Werte ihrer Kinderknoten. Der Vorteil dieser Eigenschaft ist es, dass man Baumbestandteile (Zweige) sehr effizient Vergleichen kann, ohne die gesamte Bauminformation (alle Kinder-Knoten) verarbeiten zu müssen.

Sie werden von Dynamo folgendermaßen eingesetzt: Jeder System-Knoten verwaltet einen eigenen Merkle-Baum, der Informationen über die individuellen Schlüssel seines Schlüsselbereiches speichert. Dies bedeutet, dass man durch den Vergleich des Root-Wertes der Merkle-Bäume sehr schnell Auskunft über etwaige Inkonsistenten der Schlüsselbereiche erhalten kann, ohne zahlreiche Werte vergleichen zu müssen.

2.5.3 Konsistenzmodell

Wie in vorherigen Abschnitt beschrieben, verwendet Dynamo diverse Strategien (Quorum, Hinted-Handoff und Merkle-Bäume), um auch im Falle eines Knotenausfalls die Lese- und Schreiboperationen zu garantieren. In diesem Abschnitt wird nun erörtert, welche Konsistenz Dynamo einsetzt und wie sich der Speicher verhält, wenn durch Fehlerfälle oder durch das Konsistenzmodell unterschiedliche Versionen der Daten im Speicher vorhanden sind.

Dynamo verwendet *eventual consistency* als Konsistenzmodell. Dies erlaubt die asynchrone Durchführung von Replikationen. Durch asynchrone Replikation und mögliche Netzwerkausfälle kann es zu unterschiedlichen Versionen der Daten kommen. Zusätzlich resultieren Datenbankänderungen in Dynamo in neuen Datensätzen. Meistens ersetzen neuere Datensätze (Vorgänger-) Datensätze, oder aber das System eruiert die richtige Version. Dies ist auch als *syntactic reconciliation* bekannt. Nichtsdestotrotz kann es zu zahlreichen unterschiedlichen Datensatzversionen kommen, wodurch das System keinen Abgleich der Daten durchführen kann. In diesem Fall müssen die verschiedenen Versionen der Daten vereint werden (*merging*). Dieses *merging* muss vom Nutzer des Dynamo-Speichers, z. B. der Business-Logik des Amazon-Einkaufswagens oder einer darauf basierenden Datenbankimplementierung (Amazon DynamoDB) durchgeführt werden. Hierbei ist es möglich, dass auch gelöschte Daten wieder im System erscheinen. Um unterschiedliche Versionen von Datensätzen zu erkennen, verwendet Dynamo *vector clocks*. Eine vector clock besteht aus einer Liste von Knoten und Zählerpaaren und zeichnet die Ausführungsreihenfolge von Schreibeoperationen auf.

Diese Strategie zur Verarbeitung von unterschiedlichen Versionen wurde auch in Abschnitt 2.2.1 im Kontext vom CAP-Theorem erwähnt. Dynamo implementiert hier eine Strategie, um für seinen Einsatzzweck bestmöglich alle drei erwünschten Aspekte (Consistency, Availablity und Partition Tolerance) innerhalb eines verteilten Systems zu erreichen.

2.6 Amazon DynamoDB

Die DynamoDB ist eine Datenbank, die nur in der Cloud verfügbar ist. Sie wird seit 2012 als *Database-as-a-Service* (DaaS) angeboten und basiert auf dem im vorherigen Kapitel beschriebenen Datenspeicher Dynamo (vgl. Abschnitt 2.5) [60]. Die DynamoDB wird als NoSQL-Datenbank bezeichnet, da sie die Eigenschaften von NoSQL besitzt. Die DynamoDB wird im Unterschied zur Dynamo für Entwickler zur Verfügung gestellt.

Des Weiteren kapselt DynamoDB die Komplexität von Dynamo durch eine einge-

schränkte API und eine Web-Schnittstelle. Entwickler müssen sich keine Gedanken zur Skalierung, zum Verhalten bei Ausfall, zur Replikation und zur Versionierung machen. DynamoDB verwaltet diese im Hintergrund.

NoSQL-Eigenschaft	**Dynamo**	**DynamoDB**
nicht-Relational	Ja	Ja
Schemafreiheit	Ja	Beinahe
Abfragefunktionalität	API	API
Konsistenzmodell		
BASE (eventual consistency)	Ja	eventual consistency, strong consistency
Versionsverwaltung	Vector clocks	gekappselt durch DynamoDB
Skalierbarkeit		
Partition und Replikation	Consistent Hashing	
Verarbeiten bei Ausfällen	Sloppy Quroum, Hinted Handoff, Merkle-Bäume	

Tabelle 2.2: Überblick über die NoSQL-Eigenschaften von Dynamo und DynamoDB. Die Skalierungstechniken und die Versionierung der Daten werden von DynamoDB gekapselt.

In Tabelle 2.2 wird ein Überblick über die gekapselte Funktionalität von Dynamo gegeben.

Im nachfolgenden Abschnitt wird zuerst auf das Datenmodell (nicht-relational und beinahe schemafrei), dann auf die Datenbankindizes, generelle Datenbankbeschränkungen und anschließend die Datenbankoperationen und die Abfragefunktionalität eingegangen. Im letzten Abschnitt wird noch das Konsistenzmodell der DynamoDB erörtert. Dieses kann auf zwei unterschiedliche Arten konfiguriert werden. Es wird einerseits *eventual* und anderseits *strong* consistency ermöglicht. Strong consistency bedeutet, dass nach einem erfolgreichen Update immer der aktualisierte Datensatz zurückgegeben wird [61] (Lese- und Schreibeoperationen sind *strong consistent*). Dies hat jedoch Auswirkungen darauf, wie viele Abfragen bzw. Operationen gleichzeitig auf der Datenbank verarbeitet werden können.

Die in den folgenden Abschnitten präsentierten Inhalte basieren Großteils auf der verfügbaren Entwicklerdokumentation [8, 14, 20]. Die Informationen in den folgenden Abschnitten basiert auf dem Stand der Dokumentation vom 14.07.2014.

2.6.1 Datenmodell

In diesem Abschnitt wird das Datenmodell von DynamoDB beschrieben. Zuerst wird auf den Datenaufbau eingegangen, der beschriebt, wie die Daten in der DynamoDB abgespeichert werden. Des Weiteren werden das Schlüsselschema und die Datentypen beschrieben. Am Ende dieses Abschnitts werden die Bestandteile des Datenmodells anhand von Beispielen erklärt.

Datenaufbau

In diesem Abschnitt wird das Datenmodell von DynamoDB gezeigt. DynamoDB ist eine beinahe schemafreie NoSQL-Datenbank und besteht aus Tabellen, Datensätzen und Attributen [8]:

Tabelle – Eine Tabelle der DynamoDB ist vergleichbar mit einer Tabelle aus einer relationalen Datenbank. Es können beliebig viele Datensätze gespeichert werden. DynamoDB ist fast schemafrei. Bei der Erstellung einer Tabelle müssen lediglich das Schlüsselschema ausgewählt und dessen Datentypen vordefiniert werden.

Datensatz – Ein Datensatz (Item) ist vergleichbar mit einer Zeile innerhalb einer relationalen Datenbank. Er besteht aus einer Menge von Attributnamen und Attributwerten (Key-Value-Paaren). Ein Datensatz darf die Gesamtgröße von 64 KB nicht überschreiten. Inkludiert in diesem Limit sind die Schlüsselparameter, alle Attributnamen und alle Attributwerte.

Attribut – Ein Attribut besteht aus einem Namen und einem Wert (Key-Value-Paar). Der Attributname muss eine Zeichenkette sein. Der Attributwert kann aus einem skalaren Wert oder einer Menge skalarer Werte bestehen. Innerhalb eines Datensatzes sind keine Attributduplikate möglich. Ebenso sind keine leeren bzw. Null-Werte erlaubt.

Eine der Anforderungen an Dynamo und somit auch an DynamoDB ist es, Daten sehr einfach über deren Schlüssel abfragen zu können. DynamoDB besitzt deshalb zwei unterschiedliche Schlüsselarten, welche im nachfolgenden Abschnitt beschrieben werden.

Schlüsselschema

Die Amazon DynamoDB unterstützt zwei unterschiedliche Schlüsselarten [8], welche bei der Tabellenerstellung definiert werden müssen. Die Tabellenerstellung setzt die exakte Definition des Bezeichners des Primärschlüssels und dessen Datentyp voraus. Über nachfolgende Schlüsselarten ist ein Datensatzzugriff möglich:

Der *Hash Type Primary Key*, auch als einfacher Schlüssel bezeichnet, nutzt ein definiertes Attribut (Attributname und Attributwert) eines Datensatzes, um diesen eindeutig zu identifizieren. Dieses Attribut wird auch vom Datenspeicher Dynamo verwendet, um den Speicherort des Datensatzes zu eruieren. Dieser Schlüsseltyp basiert auf einem unsortierten Hash Index.

Der *Hash und Range Type Primary Key*, auch als zusammengesetzter Schlüssel bezeichnet, benutzt zusätzlich zum einfachen Schlüssel einen sortieren Bereichsindex, der auf dem Attributnamen und dem Attributwert des Bereichsattributs basiert [8]. Somit ist es möglich, mehrere Datensätze mit demselben einfachen Schlüssel anzulegen.

Wie bereits erwähnt, ist es möglich, unterschiedliche Arten von Datentypen in der DynamoDB einzusetzen. Dynamo hingegen speichert die Datensätze als byte Arrays ab (vgl. Abschnitt 2.5.1). Im nachfolgenden Abschnitt werden nun die drei skalaren Datentypen und die Mengen (Sets) von DynamoDB beschrieben.

Datentypen

Diese Datenbank unterstützt zwei Arten von Datentypen: Skalare und Mengen (Sets). Skalare Datentypen sind Zahlen, Zeichenketten und Binärdaten. Mengen basieren nur auf den skalaren Datentypen. Leere Mengen werden nicht unterstützt. Des Weiteren sind keine Duplikate innerhalb einer Menge möglich [8]. Weiteres können die Werte der Schlüsselattribute nur aus skalaren Datentypen bestehen. Attributnamen und Tabellennamen können nur vom Typ Zeichenkette sein. Folgende Auflistung gibt einen Überblick über die unterstützten Datentypen [8]:

Zeichenketten – Eine Zeichenkette basiert auf der UTF-8-Kodierung und die Anzahl der möglichen Zeichen ist nicht eingeschränkt. Jedoch muss die maximale Datensatzgröße von 64 KB beachtet werden. Leere Zeichenketten sind nicht erlaubt.

Zahlen – Zahlen umfassen positive sowie negative Dezimal- und Integer-Werte. Eine Zahl kann eine Präzision von bis zu 38 Dezimalstellen besitzen und muss innerhalb des Wertebereichs von 10^{-128} bis 10^{+126} liegen [8].

Binärdaten – Binärdaten können beliebige binäre Werte darstellen. Hier ist ebenfalls die Einschränkung der maximalen Datensatzgröße von 64 KB zu beachten. DynamoDB kodiert Binärdaten mit Base64, da die Daten mittels JSON über die HTTP-Schnittstelle übertragen werden [13].

Nachdem in der DynamoDB kein Schema vorhanden ist, muss bei der Übertragung der Daten an die Datenbank der Datentyp zum jeweiligen Attribut definiert werden. Hierzu werden die in der Tabelle 2.3 definierten Bezeichner für Mengen bestimmter skalarer Werte und skalare Werte verwendet.

	Bezeichner
Zeichenketten	S
Zahlen	N
Binärdaten	B
Mengen aus Zeichenketten	SS
Mengen aus Zahlen	NS
Mengen aus Binärdaten	BS

Tabelle 2.3: Tabelle gibt Überblick Bezeichner von Datentypen.

Beispieldatensätze

Im vorherigen Abschnitt wurde das Datenmodell von DynamoDB präsentiert. DynamoDB besteht aus Tabellen, Datensätzen und Attributen. Es existieren zwei unterschiedliche Schlüsselarten, mit denen Datensätze aus der DynamoDB gelesen und in sie geschrieben werden können. In diesem Abschnitt werden nun zwei Beispiele gegeben, die einen Datensatz mit einem einfachen Schlüssel und einem zusammengesetzten Schlüssel zeigen. Des Weiteren wird beschrieben, wie dieser Datensatz in Dynamo verwaltet wird.

Datensatz mit einem einfachen Schlüssel

Der Datensatz in Codebeispiel 2.1 verwendet einen einfachen Schlüssel, welcher aus dem Attributnamen `Id` und dem Attributwert `Foo Fighters_These Days` besteht. Auf diesem Schlüssel wird, wie in Abschnitt 2.6.1 beschrieben, ein unsortierter Hash Index angewendet. Weitere Attribute mit skalaren Datentypen sind `Artist, Year` und `Album`. Das Attribut `Members` beinhaltet eine Menge aus Zeichenketten.

```
1 {
2   "Id": "Foo Fighters_These Days",
3   "Artist": "Foo Fighters",
4   "Year": 2011,
5   "Album": "Wasting Light",
6   "Members":  [  "Dave Grohl", "Nate Mendel",
7             "Pat Smear", "Taylor Hawkins",
8             "Chris Shiflett"  ]
9 }
```

Codebeispiel 2.1: Beispielhafter Datensatz in der Amazon DynamoDB.

In Abbildung 2.6 wird der Beispieldatensatz 2.1 auf die Hashtabellenstruktur von Dynamo abgebildet. In dieser Abbildung wird gezeigt, dass der Datensatzbestandteil `Id` verwendet wird, um den Speicherort mithilfe einer Hashfunktion zu eruieren.

```
{
  "Id": "Foo Fighters",
  {
    "Song": "These Days",
    "Year": 2011,
    "Album": "Wasting Light",
    "Members":  [   "Dave Grohl", "Nate Mendel",
            "Pat Smear", "Taylor Hawkins",
            "Chris Shiflett"  ]
  },
  {
    "Song": "Walk",
    "Year": 2011,
    "Album": "Wasting Light",
    "Members":  [  "Dave Grohl", "Nate Mendel",
            "Pat Smear", "Taylor Hawkins",
            "Chris Shiflett"  ]
  }
}
```

Codebeispiel 2.2: Beispielhafter Datensatz mit zusammengesetzten Schlüssel.

Abbildung 2.6: Darstellung zeigt den Datensatz aus Beispiel 2.1 basierend auf der Hashtabellenstruktur von Dynamo.

Datensatz mit einem zusammengesetzten Schlüssel

Der zweite Datensatz, welcher in diesem Kontext präsentiert wird, besitzt einen zusammengesetzten Schlüssel. Dieser verwendet das Attribut `Id` als einfachen Schlüssel und das Attribut `Song` als Bereichsschlüssel. In Codebeispiel 2.2 sind zwei Datensätze mit diesen Attributen sichtbar. Der Bereichsschlüssel `Song` mit den Werten `These Days` und `Walk` ermöglicht es, dass zu dem einfachen Schlüssel `Foo Fighters` zwei unabhängige Datensätze gespeichert werden können.

Diese Datensätze werden im Datenspeicher Dynamo, wie in Abbildung 2.7 ersichtlich, abgebildet. Der einfache Schlüssel wird mithilfe eines unsortierten Hash Index verwaltet und der Bereichsschlüssel mit einem unsortierten Bereichsindex. Die Schlüsselattribute bilden den Schlüssel ab, mit dem Dynamo den Speicherort eruiert.

Abbildung 2.7: Darstellung zeigt den Datensatz aus Beispiel 2.2 basierend auf der Hashtabellenstruktur von Dynamo.

2.6.2 Datenbankindizes

DynamoDB unterstützt neben den Indizes (Primärschlüssel) auf das Attribut des einfachen Schlüssels und auf das Bereichsattribut des zusammengesetzten Schlüssels auch lokale und globale sekundäre Indizes. Ein Sekundärindex ermöglicht es, Abfragen auf Attribute durchzuführen, die nicht Bestandteil eines Schlüssels sind. DynamoDB unterstützt zwei unterschiedliche Indizes [8]:

Local secondary index – Dieser Index hat den gleichen einfachen Schlüssel wie jener der Referenztabelle, jedoch einen anderen Bereichsschlüssel [12] als bei der Tabellenerstellung definiert. Ein Beispiel ist in Abbildung 2.8 ersichtlich. Hier wird ein weiterer Index auf das Attribute `Album` angewendet.

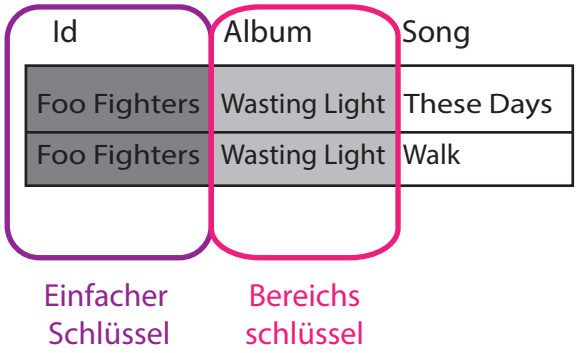

Abbildung 2.8: Darstellung zeigt einen Local Secondary Index in DynamoDB.

Global secondary index – Dieser Index kann aus einem anderen zusammengesetzten Schlüssel bestehen. Er wird eingesetzt, wenn Abfragen auf zahlreiche unterschiedliche Attribute benötigt werden [10]. Ein Beispiel, wie dieser Index strukturiert ist, ist in Abbildung 2.9 sichtbar. Hier wird ein Index mit einem zusammengesetzten Schlüssel auf die Attribute *Song* und *Year* gelegt. Zusätzlich wird der einfache Schlüssel `Id` referenziert, um den Datensatz in der Referenztabelle zu identifizieren.

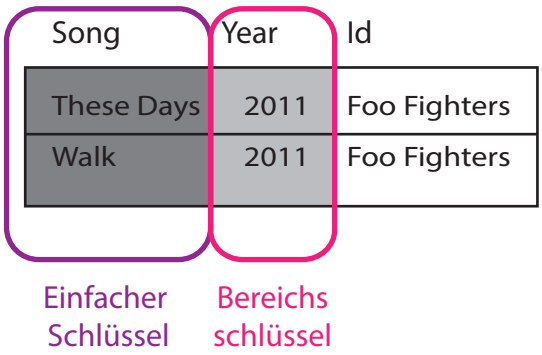

Abbildung 2.9: Darstellung zeigt einen Global Secondary Index in DynamoDB.

Es können bis zu fünf lokale und fünf globale Indizes pro Tabelle erstellt werden.

2.6.3 Datenbankbeschränkungen

In diesem Abschnitt werden die wichtigsten Beschränkungen der DynamoDB beschrieben. Es gelten folgende allgemeine Beschränkungen [11]:

Tabellennamen und Sekundärindex – Die DynamoDB erlaubt nur a-z, A-Z, 0-9, Unterstrich, Bindestrich und Punkt in den Tabellennamen und Sekundärindizes. Die Namen müssen zwischen drei und 255 Zeichen lang sein.

Datensatzgröße – Datensätze der DynamoDB dürfen nie eine Gesamtgröße (inkl. Schlüsselattribute, Attributnamen und Werte) von 64 KB überschreiten.

Länge der Attributnamen – Die Bezeichner der Schlüsselattribute müssen eine Zeichenlänge von 1 bis 255 einhalten.

Hash Primary Key – Der Schlüsselwert des Hash Primary Key darf maximal 2048 Bytes groß sein.

Range Primary Key – Der Schlüsselwert des Range Primary Keys darf maximal 1024 Bytes groß sein.

2.6.4 Datenbankoperationen

Die DynamoDB verfügt über eine einfache API, die es ermöglicht, Daten zu aktualisieren und zu schreiben. Es werden die Schreibe-Operationen `PutItem`, `BatchWriteItem`, `UpdateItem` sowie `DeleteItem` angeboten.

PutItem und BatchWriteItem

Die `PutItem`-Operation erlaubt es, Datensätze in die Datenbank zu schreiben oder vorhandene zu überschreiben [15]. Durch die Angabe eines `Expected`-Attributs ist es möglich, das Überschreiben der Datensätze an Bedingungen zu knüpfen. Treffen diese zu, wird der jeweilige Datensatz überschrieben [15]. Zusätzlich können diese Bedingungen

durch die Angabe eines logischen Operators (`ConditionalOperator`: `AND, OR`) verknüpft werden. Nach erfolgreicher Durchführung der `PutItem`-Operation ist es möglich, überschriebene Werte zu retournieren. Standardmäßig wird kein Wert retourniert.

```
{
  TableName:  Music,
  Item:
    {
      Year = { N: 2011, },
      Members = { SS:
            [
              Dave Grohl,  Nate Mendel,
              Pat Smear,  Taylor Hawkings,
              Chris Shiflett
            ],
      },
      Id = { S: Foo Fighters_These Days,},
      Artist = { S: Foo Fighters,},
      Album = { S: Wasting Light,}
    },
}
```

Codebeispiel 2.3: Beispiel einer `PutItem`-Operation eines Datensatzes mit einem einfachen Schlüssel.

In Beispiel 2.3 wird ein Datensatz in die Datenbank geschrieben. Die API benötigt die Angabe des `TableName`-Elements zur Definition der Zieltabelle. Das Element `Item` beinhaltet den vollständigen Datensatz (inkl. Attribute und deren Datentypen).

Die `BatchWriteItem`-Operation ermöglicht das Schreiben oder das Löschen von Daten aus unterschiedlichen Tabellen. Sie kann bis zu einem MB an Daten schreiben oder maximal 25 Schreibe- oder Löschoperationen durchführen [5]. In Beispiel 2.4 wird ein Datensatz aus der Tabelle `Music` gelöscht und ein neuer hinzugefügt. Innerhalb des Elements `RequestItems` können mehrere Tabellenbezeichner angegeben werden und auch die Operationen, welche auf diesen Tabellen durchgeführt werden sollen. Per Tabellenbezeichner (hier `Music`) ist es möglich, mit den Elementen `DeleteRequest` und `PutRequest` die Daten über den Schlüssel zu löschen oder neue hinzuzufügen.

```
{
  RequestItems:
  {
      Music:
      [
        {
          DeleteRequest: {
            Key:
            {
              S: Foo Fighters_These Days
            }
          }
      },
      {
        PutRequest: {
            Item:
          {
            Year = { N: 2011, },
            Members = { SS:
              [
                Dave Grohl,  Nate Mendel,
                Pat Smear,  Taylor Hawkings,
                Chris Shiflett
              ],
            },
            Id = { S: Foo Fighters_These Days,},
            Artist = { S: Foo Fighters,},
            Album = { S: Wasting Light,}
          }
          }
      }
      ]
  }
}
```

Codebeispiel 2.4: Beispiel einer `BatchWriteItem`-Operation mit einer Lösch- und Schreibeoperation.

Die Schreibe- und Löschoperationen werden atomar durchgeführt, die gesamte `BatchWriteItem`-Operation nicht. Dies bedeutet, kommt es zu einem Fehler bei der Durchführung, bricht die `BatchWriteItem`-Operation ab und retourniert eine Menge an nicht verarbeiteten Operationen (`UnprocessedItems`) [5].

UpdateItem

Die `UpdateItem`-Operation aktualisiert Datensätze oder fügt einen oder mehrere neue hinzu, wenn diese noch nicht vorhanden sind [19]. Des Weiteren werden verschiedene Aktionen auf Attribute ermöglicht. Diese Operation unterstützt die Aktionen `ADD, DELETE`

und `PUT`, wodurch die `UpdateItem`-Operation auf unterschiedliche Art und Weise ausgeführt werden kann. Die Resultate der Aktionen sind abhängig von der Existenz eines Datensatzes. Wird ein zu aktualisierender Datensatz in der Datenbank gefunden, liefern die Aktionen die folgenden Resultate [19]:

ADD - Wenn das Attribut nicht existiert, wird dieses zum Datensatz hinzugefügt. Existiert das Attribut, ist die durchgeführte Aktion abhängig vom Datentyp. Wird die `ADD`-Aktion auf numerische Werte angewendet, wird eine Addition durchgeführt. Bei Attributen mit Mengen wird ein neuer Wert der Menge hinzugefügt.

DELETE - Wenn der Aktion kein Wert übergeben wurde, wird das beschriebene Attribut aus dem Datensatz gelöscht.

PUT - Es wird ein neues Attribut hinzugefügt oder das existierende überschrieben.

Wird kein zu aktualisierender Datensatz in der Datenbank gefunden, verfahren diese Aktionen anders [19]:

ADD - Es wird ein neuer Datensatz mit angegebenen Schlüsselwerten und Attributen erzeugt. Es sind nur numerische Werte und Mengen aus numerischen Werten erlaubt.

DELETE - Es wird keine Aktion durchgeführt.

PUT - Es wird ein neuer Datensatz mit angegebenen Schlüsselwerten und Attributen erzeugt.

Zusätzlich ist es möglich, Aktualisierungen basierend auf einer Bedingung durchzuführen (ähnlich zur `PutItem`-Operation), indem auch hier ein `Expected`-Attribut angegeben werden kann [19]. Des Weiteren ist auch der logische Operator (`ConditionalOperator`) verfügbar [19]. Diese Operation kann auch Ergebnisse (z. B. alle Überschriebenen) retournieren. Standardmäßig wird kein Ergebnis retourniert.

In Beispiel 2.5 wird eine Aktualisierung des Attributes `Year` durchgeführt. Hier wird die `ADD`-Aktion auf einen existierenden Datensatz durchgeführt, wodurch es zu einer Addition auf einen nummerischen Wert kommt. Das Element `AttributeUpdates` bezeichnet die Menge an Attributen, die aktualisiert werden sollen. Das `Key`-Element gibt den Schlüssel des Datensatzes an.

DeleteItem

Diese Operation löscht durch die Angabe des Primärschlüssels einen Datensatz aus der Datenbank. Zusätzlich können Bedingungen definiert werden (`Expected`). Tritt eine Bedingung ein (z. B. ein erwarteter Attributwert), so wird der Datensatz dementsprechend angepasst [7] (Löschung des Datensatzes oder eines Attributes). Diese Bedingung kann auch mit der Angabe eines logischen Operators (`ConditionalOperator`), wie bei `PutItem` und `UpdateItem`, verknüpft werden. Diese Operation besäße auch die Funktionalität, gelöschte Werte zu retournieren, welche jedoch standardmäßig deaktiviert ist.

In Beispiel 2.6 wird ein Datensatz über seinen Schlüssel gelöscht. Der Schlüssel des Datensatzes wird im Element `Key` definiert. Die Tabelle wird ebenfalls über das Element `TableName` definiert.

```
{
  AttributeUpdates:
  {
    Year:
      {
      Action: ADD,
      Value: { N: 2},
      }
  },
  Key:
  {
    S: Foo Fighters_These Days
  }
  TableName: Music
}
```

Codebeispiel 2.5: `UpdateItem`-Operation, die eine Addition auf das Attribut *Year* durchführt.

Die in Beispiel 2.6 gezeigte Operation löscht den Datensatz anhand eines zusammengesetzten Schlüssels.

```
{
    TableName: MusicDetails,
    Key: {
        Id: {
            S: Foo Fighters
        },
        Song: {
            S: These Days
        }
    }
}
```

Codebeispiel 2.6: Beispiel einer `DeleteItem`-Operation.

Zusammenfassend manipulieren diese Operationen die Daten der DynamoDB. Im nachfolgenden Abschnitt wird auf die Abfragemöglichkeiten eingegangen.

2.6.5 Abfragefunktionalität

Die DynamoDB bietet mehrere einfache und komplexe Abfragemechanismen an. Die einfachen Abfrageoperationen (*GetItem* und *BatchGetItem*) greifen über den definierten Schlüssel (einfach oder zusammengesetzt) auf die Items zu. Die komplexen Abfragen suchen Datensätze über ihre Schlüsselattribute (*Query*) und ihre Nichtschlüsselattribute (*Scan*). Der nachfolgende Inhalt basiert auf den Informationen aus der offiziellen Entwicklerdokumentation [4, 9, 16–18] (Stand: Juli 2014). In dieser Arbeit werden nur

die wichtigsten Parameter der Abfragefunktionalitäten erörtert und in den Beispielen veranschaulicht, für weitere Informationen wird auf die Dokumentation [17] verwiesen.

In diesem Abschnitt werden zuerst wichtige Hinweise zur Ergebnismenge von DynamoDB gegeben und dann die möglichen Vergleichsoperatoren von DynamoDB beschrieben. Anschließend wird anhand von Beispielen detailliert auf die Abfrageoperationen eingegangen.

Ergebnismenge

DynamoDB beschränkt die zurückgelieferte Ergebnismenge mit der Größe von 1 MB oder 100 Datensätzen. Tritt diese Beschränkung ein, retourniert die Datenbank eine Referenz auf den nächsten Datensatz, welcher bei der durchgeführten Operation nicht mehr verarbeitet wurde. Der zuletzt verarbeitete Wert wird in einer Antwort der Datenbank im Element `UnprocessedKeys` gespeichert. Es obliegt dem Nutzer, weitere Datensätze abzufragen und dementsprechend eine Abfrage zu formulieren (über die Angabe des Elementes `ExclusiveStartKey`). Dies trifft auf die Abfragen `BatchGetItem`, `Query` und `Scan` zu. Zusätzlich können die retournierten Attribute eingeschränkt werden (`AttributesToGet`). Die selektive Auswahl der Attribute ist bei beiden Abfragen `GetItem`, `BatchGetItem`, `Query` und `Scan` möglich.

Vergleichsoperatoren

Es werden die in der Tabelle 2.4 angeführten Vergleichsoperationen von der DynamoDB-API [6] unterstützt:

Operator	Anwendbarkeit
Equals	A
Not Equals	A
Less Equals	S
Less Than	S
Greater Equals	S
Greater Than	S
Between	S
Not Null	E
Null	E
Contains	Z, B, M
Not Contains	Z, B, M
In	M
Begins with	Z, B

Tabelle 2.4: Die Tabelle zeigt eine Übersicht über die Vergleichsoperatoren und auf welche Parameter diese angewendet werden können.

Legende:

A	Alle
B	Binärdatentypen
S	Skalare Datentypen
M	Mengen
Z	Zeichenketten
E	Existenz des Attributes

Datenwerte können nur verglichen werden, wenn sie vom selben Datentyp sind. Somit würde beispielweise der Vergleich zwischen einer Zeichenkette { „S“: „1“ } und einem numerischen Wert{„N“: 1 } ein negatives Ergebnis retournieren. Vergleiche wie *größer als*, *ist gleich* und *weniger als* auf Zeichenketten basieren auf den numerischen Werten der ASCII-Zeichensatzkodierung [16].

GetItem und BatchGetItem

Die `GetItem`-Operation fragt einen Wert aus der Datenbank durch die Angabe des Schlüssels (einfacher Schlüssel oder zusammengesetzter Schlüssel) ab. Das Codebeispiel 2.7 stellt eine `GetItem`-Abfrage dar. Hier werden beide Schlüsselparameter des zusammengesetzten Schlüssels angegeben (mithilfe des Elements `Key`), um einen Datensatz abzufragen. Zusätzlich werden nur die Attribute (durch die Angabe des Elements `AttributesToGet`) `Song`, `Year` und `Album` retourniert. In dieser Beispielabfrage muss außerdem angegeben werden, welchen Datentyp das Schlüsselattribut hat. Hier wird eine Zeichenkette (S) abgefragt (vgl. Abschnitt 2.6.1 für die Kurzbezeichnungen der Datentypen und Mengen).

Die `BatchGetItem`-Operation besitzt primär dieselbe Funktionalität wie die `GetItem`-

```
{
    TableName: MusicDetails,
    Key: {
        Id: {
            S: Foo Fighters
        },
        Song: {
            S: These Days
        }
    },
    AttributesToGet: [Song, Year, Album]
}
```

Codebeispiel 2.7: Dieses Codebeispiel zeigt eine `GetItem`-Abfrage auf eine Tabelle mit zusammengesetztem Schlüssel. Zusätzlich sollen nur drei Attribute des Datensatzes geladen werden.

Operation, jedoch kann sie beliebig viele Punktabfragen auf unterschiedliche Tabellen in einer Abfrage durchführen. Das Codebeispiel 2.8 zeigt eine `BatchGetItem`-Abfrage auf mehrere Tabellen unter Einsatz unterschiedlicher Primärschlüssel. Es werden die Daten aus den Beispielen 2.1 und 2.2 abgefragt. Die Definition, welche Tabelle abgefragt werden soll, wird durch die Angabe des Elements `RequestItems` ermöglicht. Die Tabelle `Music` enthält Datensätze basierend auf einem einfachen Schlüsselschema und die Tabelle `MusicDetails` verwendet einen zusammengesetzten Schlüssel. Des Weiteren werden unterschiedliche Attribute pro Tabelle retourniert, dies wird durch die Angabe des Bezeichners `AttributesToGet` ermöglicht.

Query

Die `Query`-Operation ermöglicht Abfragen auf die Attribute des Primärschlüssels und auf die Sekundärindizes. Abfragen auf Sekundärindizes können über das Element `IndexName` definiert werden. Im Unterschied zur `GetItem`-Operation erlaubt die `Query`-Operation Suchabfragen (mit Vergleichsoperationen) auf den Bereichsschlüssel des zusammengesetzten Schlüssels. Um eine solche Abfrage zu definieren, müssen die Schlüsselbedingungen wie folgt formuliert werden: Das Attribut des einfachen Schlüssels muss mit dem `equals`-Operator und das Attribut des Bereichsschlüssels mit einer der Operationen `equals`, `less equals`, `less than`, `greater equals`, `greater than`, `begins with` und `between` eingeschränkt werden [16]. Über die Filterfunktion (`QueryFilter`) ist auch eine Selektion auf die anderen nicht-Schlüsselattribute möglich. Somit können weitere Attribute eines Datensatzes als Filterkriterium angegeben werden. Die zusätzliche Angabe eines `ConditionalOperator` (*AND* und *OR*) bei der QueryFilter verknüpft mehrere Attribute, um nur jene Datensätze zu retournieren, die diese Attribute beinhalten [16]. Zusätzlich ist es möglich, wie bei der `GetItem`-Operation, nur gewisse Attribute mithilfe des `AttributesToGet` zu laden.

In Beispiel 2.9 wird eine Query-Operation durchgeführt, die den zusammengesetz-

```
{
    RequestItems: {
        Music: {
            Keys: [
                {
                    Id:{S : Foo Fighters_These Days }
                }
            ],
            AttributesToGet:
                    [ Artist, Album, Members ]
        },
        MusicDetails: {
            Keys: [
                {
                   Id: {  S: Foo Fighters   },
                   Song: {  S: These Days   }
                }
            ],
            AttributesToGet: [Song, Year, Album]
        }
    }
}
```

Codebeispiel 2.8: Beispiel einer `BatchGetItem`-Abfrage auf zwei unterschiedliche Tabellen mit unterschiedlichen Schlüsselschemas (einfach und zusammengesetzter Schlüssel). Des Weiteren werden in diesem Beispiel unterschiedliche Attribute pro Tabelle geladen

ten Schlüssel nach Liedern durchsucht, die mit `These` beginnen. Im Elemente `Select` wird angegeben, dass alle Attribute der Datensätze retourniert werden sollen. Das Element `KeyConditions` beinhaltet die eigentliche Anweisung, nach welchen Kriterien die Attribute durchsucht werden. Der `ComparisonOperator` definiert die durchgeführte Vergleichsoperation der Attribute.

Scan

Die `Scan`-Operation durchsucht jeden Datensatz innerhalb einer Tabelle, wobei beliebige (Nichtschlüssel) Attribute gefiltert werden können. Ähnlich zur Query-Abfrage führt sie diverse Vergleichsoperationen auf die Attribute durch. Es werden folgende Operatoren unterstützt: `equals`, `not equals`, `less equals`, `less than`, `greater equals`, `greater than`, `begins with`, `between`, `not null`, `null`, `contains`, `not contains` und `in`. Somit ist es möglich, das Attribut `Year` innerhalb eines Zeitraums von Codebeispiel 2.2 abzufragen, obwohl es kein Bestandteil des Schlüssels ist. Des Weiteren erlaubt der `Scan` konjunktive (AND) und disjunktive (OR) Abfragen, indem dem `ScanFilter` mehrere Bedingungen durch die Angabe eines `ConditionalOperator` übergeben werden [18]. Standardmäßig werden die Bedingungen mit *AND* (Konjunktiv-Abfrage) verbunden.

```
{
    TableName: MusicDetails,
    Select: ALL_ATTRIBUTES,
    KeyConditions: {
        Song:
        {
            AttributeValueList: [
                {
                    S: These
                }
             ],
            ComparisonOperator: BEGINS_WITH
        },
        Id:
        {
            AttributeValueList: [
                {
                    S: Foo Fighters
                }
            ],
            ComparisonOperator: EQ
        }
    }
}
```

Codebeispiel 2.9: Beispiel enthält eine `Query`-Abfrage, welche einen Zeichenketten-Vergleich (`BEGINS_WITH`) auf das Bereichsattribut des zusammengesetzten Schlüssels durchführt.

Des Weiteren kann diese Operation parallelisiert werden. Die Durchführung ist allerdings limitiert auf eine durchsuchte Datensatzmenge von 1 MB. Wird dieses Limit erreicht, bricht die Abfrage ab und retourniert mit einer Referenz auf den zuletzt durchsuchten Datensatz (`LastEvaluatedKey`).

In Codebeispiel 2.10 wird eine Abfrage der Scan–Operation dargestellt. Mithilfe des Elementes `ScanFilter` wird das Attribut `Year` mit dem `Between`-Operator einschränkt. Ähnlich zur `Query`-Operation wird mithilfe des Elements `ComparisonOperator` der Vergleichsoperator definiert. Die Abfrage durchsucht die gesamte Tabelle `MusicDetails` und vergleicht alle `Year`-Attribute im Bereich von 2010 bis 2012.

2.6.6 Datenkonsistenz

DynamoDB unterstützt, wie bereits beschrieben, *eventual* und *strong* consistency. Um sie zu erreichen, werden *Read-* und *Write-Throughput*-Metriken bei DynamoDB definiert. Eine Read-Kapazität wird auf Basis der Datensatzgröße von 4 KB berechnet. Datensätze, welche größer als dieses Limit sind, benötigen mehr als eine Read-Kapazität. Die Berechnung basiert auf den gewünschten Leseoperationen pro Sekunden multipliziert

```
{
    TableName: MusicDetails,
       ScanFilter: {
         Year: {
            AttributeValueList: [
                {  N: 2010 } ,
                {  N: 2012 } ,
         ],
            ComparisonOperator: BETWEEN
        }
    }
}
```

Codebeispiel 2.10: Beispiel enthält eine Scan-Abfrage mit dem `Between`-Operator.

mit einer Datensatzgröße von 4 KB [14]. Eine Read-Kapazität repräsentiert eine strong consistent Leseoperation oder zwei eventual consistent Leseoperationen pro Sekunde.

Eine Write-Kapazität führt die gleiche Berechnung wie die Read-Kapazität durch, jedoch wird hier von einer Datensatzgröße von 1 KB ausgegangen [14]. DynamoDB verlangt, dass diese Throughput-Werte bei der Erstellung einer Datenbanktabelle definiert werden, ansonsten werden eingehende Lese- und Schreiboperationen gedrosselt.

2.6.7 Zusammenfassung

In diesem Kapitel wurden die Grundlagen von NoSQL-Datenbanken erörtert. Eine NoSQL-Datenbank zeichnet sich durch die Charakteristiken nicht-relational, verteilt, Open-Source und skalierbar aus. Des Weiteren besitzt sie die Eigenschaften der Schemafreiheit, einer einfachen API und eines schwachen Konsistenzmodells. Die Amazon DynamoDB ist eine solche NoSQL-Datenbank, welche die genannten Charakteristiken erfüllt und beinahe schemafrei ist. Die API der DynamoDB ist einfach gehalten. Jedoch sind auch komplexe Abfragen auf Nichtschlüsselattribute möglich. Im nachfolgenden Kapitel wird die Grundlage zu Verschlüsselung erörtert und ein Schwerpunkt auf die Datenbankverschlüsselung gesetzt.

Kapitel 3

Verschlüsselung

Dieses Kapitel erörtert die Grundlagen für den in dieser Arbeit implementierten Verschlüsselungsclients. Der erste Teil beschreibt konkrete Verschlüsselungsverfahren und -algorithmen. Der zweite Teil beschreibt diverse Ansätze zum Thema Verschlüsselung von Datenbanken, die auf Basis von definierten Kriterien verglichen werden.

3.1 Verschlüsselungstechniken

In diesem Abschnitt wird ein Überblick über Verschlüsselungsalgorithmen gegeben. Der Fokus hierbei liegt auf der symmetrischen Verschlüsslung von Daten. Daher wird für asymmetrischen Verfahren nur mehr ein Überblick gegeben. Diese Darstellungen basieren großteils auf den Quellen [52, 55]. Ein weiterer Punkt in diesem Abschnitt ist die homomorphe Verschlüsselung. Diese erlaubt es, mathematische Operationen auf die Klartexte verschlüsselter Daten durchzuführen. Im letzten Teil werden zusätzlich noch kryptographische Hash-Verfahren im Überblick dargestellt und beschrieben.

3.1.1 Symmetrische Verschlüsselung

Die Kernidee dieser Verfahrens ist es, mit *dem gleichen* Schlüssel sowohl die Verschlüsselung als auch die Entschlüsselung durchzuführen. Es existieren zwei unterschiedliche Arten von Algorithmen, die eine symmetrische Verschlüsselung realisieren: die *Strom*- und die *Block*-Chiffren [52]. Strom-Chiffren wandeln einen Klartext Bit-weise in einen verschlüsselten Text um. Die Bit-weise Verschlüsselung erfolgt durch die Verknüpfung mit einem Schlüsselstrom. Der Schlüsselstrom wird mithilfe des Schlüssels und in Kombination mit einem Schlüsselstromgenerator erstellt. Zur Entschlüsselung der Daten muss derselbe Schlüssel eingesetzt werden. Auf diese Algorithmen-Art wird in dieser Arbeit nicht näher eingegangen, da diese nicht in der Beispielimplementierung verwendet wurde.

Die zweite Art eine symmetrische Verschlüsselung zu realisieren erfolgt mithilfe von Block-Chiffren. Bekannte Vertreter dieser Kategorie sind der *DES* (Data Encryption Standard) und der *AES* (Advanced Encryption Standard). Bei der Anwendung dieser Algorithmen werden Klartexte in Blöcke fixer Größe unterteilt und diese Blöcke an-

schließend einzeln vom Verschlüsselungsalgorithmus verschlüsselt. Jeder dieser Blöcke wird immer mit jenem Schlüssel entschlüsselt, mit welchem er auch verschlüsselt wurde. Wenn jeder Block einzeln mit dem gleichen Schlüssel verschlüsselt wird, entstehen bei gleichen Datenreihenfolgen innerhalb vom Klartext gleiche Datenreihenfolgen innerhalb vom Schlüsseltext. Es sind somit Rückschlüsse auf die verschlüsselten Inhalte der Blöcke möglich.

Um die Sicherheit von Block-Chiffren zu erhöhen, werden sogenannte *Modes of Operation* und das Konzept des *Initialisierungsvektors* (Initial Vector) eingesetzt. Diese Modi erlauben es, durch einen definierten Ablauf bei der Verschlüsselung der Blöcke, die Schlüsseltextinhalte zufällig anzuordnen, ohne die Sicherheit der eingesetzten Verschlüsselungsalgorithmen zu beeinflussen. Dies ist dann von großer Bedeutung, wenn gleiche Fragmente eines Klartextes die Blockgröße der Block-Chiffren übersteigen und in gleichen Schlüsseltextfragmenten resultieren.

Ebenfalls werden in diesem Abschnitt die *Paddings* besprochen. Diese werden dann benötigt, wenn ein Klartext nicht genau ein vielfaches der Blockgröße der verwendeten Blockchiffre ist.

Verschlüsselungsalgorithmen

Zwei bekannte Vertreter der symmetrischen Blockverschlüsselungsalgorithmen sind der *DES* (Data Encryption Standard) und der *AES* (Advanced Encryption Standard).

DES wurde von IBM entwickelt und 1975 veröffentlicht. Er verschlüsselt 64-bit große Blöcke und verwendet eine Schlüssellänge von 56-bit.

Der zweite bekannte symmetrische Blockalgorithmus ist AES. Dieser wurde vom National Institute of Standards and Technology (NIST) 2001 veröffentlicht [54]. Entwickelt wurde dieser Algorithmus von zwei belgischen Forschern namens Daemen und Rijmen [55]. Er wird als Ablöse für den DES angesehen [54]. Der AES verwendet, im Vergleich zum DES, eine Blockgröße von 128-bit und erlaubt eine Schlüssellänge von 128-, 192- und 256-bit.

Um die Sicherheit dieser Blockalgorithmen zu erhöhen, werden Modes of Operation eingesetzt. Diese werden im nachfolgenden Abschnitt näher erörtert.

Modes of Operation

Die *Modes of Operation* sind für den DES entwickelt worden, um die Sicherheit zu erhöhen, ohne den zugrundeliegenden Algorithmus zu ändern [55]. Die Modi können auch für den AES eingesetzt werden. Es gibt zahlreiche Modi, jedoch werden in dieser Arbeit nur zwei beschrieben. Diese werden vom Verschlüsselungsclient, der im Zuge dieser Arbeit entwickelt wurde, verwendet.

Der erste Modus ist *ECB* (Electronic Codebook Mode). Jeder Block wird mit demselben Schlüssel verschlüsselt [55]. Die große Schwäche dieses Modus ist, dass zwei gleiche Blöcke nach der Verschlüsslung denselben Schlüsseltext produzieren [55]. Daher sind Rückschlüsse (durch statistische Auswertungen) auf die Inhalte möglich, ohne den Schlüssel zu kennen.

Der zweite Modus ist *CBC* (Cipher Feedback Mode). Bei diesem Modus wird jeder unverschlüsselte Block mit dem vorher verschlüsselten durch XOR verknüpft, bevor dieser mit dem Schlüssel erneut verschlüsselt wird [55]. Um den ersten unverschlüsselten Block zu kodieren, verknüpft ihn CBC durch eine XOR-Operation mit dem sogenannten *Initialisierungsvektor* (IV) [55]. Der Initialisierungsvektor ist, je nach Mode of Operations, eine beliebig wählbare Zusatzinformation. Er kann z. B. mithilfe von Zufallsgeneratoren generiert oder es kann einfach ein Zeitstempel verwendet werden [52]. Mit verschiedenen IV werden identische Klartexte mit gleichem Schlüssel zu unterschiedlichen Schlüsseltexten verschlüsselt. Der IV muss nicht geheim sein, er ist lediglich ein Zusatz zum Verschlüsselungsalgorithmus [52]. Der Schlüssel ist entscheidend für die Sicherheit der Chiffren.

Padding

In der Verschlüsselung ist *Padding* das Auffüllen von Blöcken, welche kleiner sind, als die Blockgröße der Blockchiffre (z. B. 128-bit bei AES, 64-bit bei DES [52]). Es gibt verschiedene Arten von Padding. Die Java-API unterstützt diverse Paddings, eines davon ist das *PKCS5-Padding* [44]. Es fügt die fehlenden Bytes dem Block hinzu, wobei in jedes dieser Bytes die Anzahl an fehlenden Bytes geschrieben wird [38]. Ein Beispiel hierzu ist: Wenn zwei Bytes innerhalb des letzten Blockes fehlen, dann füllt das Padding zwei Bytes mit jeweils dem Wert zwei auf. Es existieren zahlreiche andere Paddings, einige davon füllen die fehlenden Bits mit Füllwerten wie Nullen oder Einsen auf [52].

3.1.2 Asymmetrische Verschlüsselung

Die asymmetrische Verschlüsselung ist auch unter dem Namen *Public-key Cryptography* [52, 55] bekannt. Die Kernidee ist, dass zur Verschlüsselung und Entschlüsselung von Daten zwei unterschiedliche Schlüssel existieren, ein *öffentlicher* und ein *privater* [52, 55]. Die Verschlüsselung der Daten erfolgt mithilfe des öffentlichen Schlüssels und die Entschlüsselung mit dem privaten Schlüssel. Die Entschlüsselung ist somit nur dem Besitzer des privaten Schlüssels möglich, was diese Art von Verschlüsselung erheblich aufwendiger macht. Im Vergleich zur symmetrischen Verschlüsselung ist das Public-Key-Verfahren von Vorteil, wenn Daten mit anderen Nutzern geteilt werden müssen und keine sichere Kommunikation des Verschlüsselungsschlüssels möglich ist [55]. Ein bekannter Vertreter dieser Verschlüsselung ist das *RSA*-Verfahren, welches 1977 entwickelt wurde [55].

3.1.3 Homomorphe Verschlüsselung

Von homomorpher Verschlüsselung wird gesprochen, wenn es möglich ist, Operationen wie z. B. Abfragen [29] oder mathematische Operationen wie Additionen [34] mit verschlüsselten Daten durchführen zu können. Neben den arithmetischen Operationen haben homomorphe Verschlüsselungen zum Ziel, verschlüsselte Daten effizient abfragen und manipulieren zu können (keine Entschlüsselung notwendig) [29].

Eine Art der homomorphen Verschlüsselung ist die *Order Preserving Encryption*

(OPE). Hierbei werden Zahlen so verschlüsselt, dass man, unter dem Einsatz gleicher Schlüssel, auf diese numerische Vergleichsoperationen durchgeführt werden können. Dies erlaubt die Realisierung von Bereichsabfragen (z.B. BETWEEN, ORDER BY, MIN, MAX) und SORT-Operationen [50] auf verschlüsselte Daten [2]. Die Kernidee hinter dieser Art der Verschlüsselung ist es, Zahlen mit demselben Schlüssel, in eine verschlüsselte numerische Reihenfolge zu transformieren. Eine konkrete Implementierung von OPE und solcher Bereichsabfragen wird in CryptDB[1] eingesetzt [23].

3.1.4 Kryptographische Hash-Funktionen

Eine Hash-Funktion wird eingesetzt, um einen *Fingerabdruck* von Daten (einzigartigen Wert) zu erstellen [55]. Eine Hash-Funktion nimmt einen beliebig langen Wert entgegen und berechnet einen Hash-Wert mit einer fixen Länge. Ein bekannter Vertreter von Hash-Funktionen ist SHA-256, welcher einen Hash-Wert mit 256-bit Größe erstellt. Eine solche Hash-Funktion wird z. B. auch von Dynamo eingesetzt (vgl. Abschnitt 2.5.1). Dynamo erstellt mit dieser Hash-Funktion einen Wert, der die Speicherstelle eines Datensatzes eindeutig identifiziert und von einer fixen Größe ist.

3.2 Verschlüsselung von Datenbanken

3.2.1 Vergleichskriterien

Die Datensicherheit von Datenbanken ist kein neues Thema. Sie gewinnt jedoch durch die steigende Beliebtheit von *Cloud Computing* und den *Database-as-a-Service*-Diensten (DaaS-Dienste) an neuer Bedeutung [21]. Hierbei werden die Daten in externen Systemen gespeichert. Dies bedeutet, dass neben den allgemein bewährten Strategien zum Schutz der Daten, wie Firewalls und Zugangskontrollen, auch die Verschlüsselung relevant ist.

In diesem Abschnitt werden bereits angewandte bzw. in der Literatur diskutierte Datenbankverschlüsselungen beschrieben und verglichen. Hierbei wird ein Überblick gegeben und der in dieser Arbeit präsentierte und implementiere Ansatz von den anderen Ansätzen abgegrenzt. Die Vergleichskriterien der Literaturanalyse umfassen nachfolgende Punkte, welche anschließend noch detaillierter beschrieben werden.

Schutz - Kann die Sicherheit der Daten durch die Datenbankverschlüsselung gewährleistet werden?

Verschlüsselungsort - Wo wird die Verschlüsselung durchgeführt und wo befindet sich der Speicherort der Verschlüsselungsschlüssel?

Verschlüsselungsgranularität - Wie feingranular wird eine Verschlüsselung auf die Daten angewendet?

Austauschbarkeit des Algorithmus - Ist die eingesetzte Verschlüsselung von einem bestimmten Verschlüsselungsalgorithmus oder einer konkreten Verschlüsselungsimplementierung abhängig?

[1] CryptDB - http://css.csail.mit.edu/cryptdb/

Abfragemächtigkeit - Welche Datenbankabfragen können durchgeführt werden?

Mehrbenutzerfähigkeit - Ist trotz Verschlüsselung noch ein Mehrbenutzerbetrieb auf denselben Daten möglich?

Zugriffskontrolle - Können feingranulare Benutzerrechte durch die Verschlüsselung abgebildet werden?

Bei dem Kriterium *Schutz* wird als Bedrohungsszenario von einem *Honest But Curious*-Server ausgegangen [37]. Dies bedeutet, dass der Server alle Informationen (Daten), die er bekommt, verarbeiten kann und wird. Um den Schutz der Daten zu gewährleisten, müssen für dieses Kriterium die zwei nachfolgenden Bedingungen erfüllt sein:

- Es *können keine Rückschlüsse* durch Zugriff auf die Schlüsseltexte der verschlüsselten Datensätze gezogen werden. Rückschlüsse sind nur dann möglich, wenn innerhalb der Schlüsseltexte Muster erkannt werden (statistische Auswertungen).
- Zu keinem Zeitpunkt hat der Server *Zugriff auf die unverschlüsselten Daten*, weder durch Abfangen der Daten vor der Verschlüsselung noch durch Zugriff auf die Verschlüsselungsschlüssel.

Das zweite Kriterium des *Verschlüsselungsortes* beschäftigt sich mit dem Durchführungsort der Verschlüsselung und dem Speicherort der Verschlüsselungsschlüssel. Dieser Ort kann sowohl der Client[2], wie auch der Server sein. Biryukov et al. [22] beschreiben drei Ebenen (Speichersystem, Datenbankmanagementsystem und Applikation) in denen die Datenbankverschlüsselung durchgeführt werden kann. Zwei der Ebenen (Speichersystem und Datenbankmanagementsystem) umfassen die Datenbankverschlüsselung serverseitig und eine Ebene clientseitig (Applikation). Diese drei Ebenen der Datenbankverschlüsselung sind wie folgt charakterisiert:

Speichersystem – Diese Ebene umfasst die physische Sicherheit und die Sicherheit auf Ebene des Speichersystems/Betriebssystems. Bei der physischen Sicherheit spricht man unter anderem von der Verschlüsselung der Festplatten, um diese vor Diebstahl zu schützen. Auf Speichersystem-/Betriebssystemebene werden die einzelnen Dateien oder ganze Verzeichnisse verschlüsselt. Des Weiteren erörtern Biryukov et al. die Verschlüsselung auf Speichersystemebene aus der Datenbanksicht. Der Vorteil davon ist, dass diese Verschlüsselung transparent zur Datenbank ist. Dies bedeutet, dass keine Änderungen am Datenmodel (z. B. logische Abspeicherung von binären Daten) von der Datenbank durchgeführt werden müssen, um die Daten zu verschlüsseln. Der Nachteil ist, dass das Speichersystem keine Informationen über die Datenbankobjekte und Strukturen besitzt. Daher kann die Verschlüsselung nicht nach Benutzerrechten durchgeführt werden und ist nur auf Dateienebene (z. B. Datenbankdateien oder Log-Dateien) möglich [22]. Die Verschlüsselungsschlüssel müssen dem Speichersystem/Betriebssystem zur Verfügung stehen, um eine Verschlüsselung durchführen zu können.

Datenbankmanagementsystem – Das primäre Ziel dieser Ebene ist das sichere Speichern und Laden der Daten mithilfe des Datenbankmanagementsystems [22]. Die

[2] Anzumerken ist, dass es sich bei dem Client um einen gesicherten Ort handelt. Es kann sich hier auch um einen internen Applikationsserver handeln.

Verschlüsselung der Daten kann daher als ein Teil des Datenbankdesigns angesehen werden, wie z. B. die Definition der Benutzerrechte auf Daten. Des Weiteren erörtern Biryukov et al. dass die Verschlüsselung der Daten auf dieser Ebene in verschiedenen Granularitäten (Tabellen, Spalten, Zeilen) erfolgen kann. Sowohl die Verschlüsselung der Daten als auch die Speicherung der Schlüssel wird auf einem Datenbankserver durchgeführt [22].

Applikation – Auf dieser Ebene wird der Verschlüsselungs- und Entschlüsselungsprozess in der Applikation (clientseitig) durchgeführt. Im Vergleich zur Ebene des Datenbankmanagementsystems werden die Schlüssel nur clientseitig gespeichert. Die Daten werden bereits verschlüsselt an den Server geschickt. Der Vorteil der Verschlüsselung und Entschlüsselung der Daten auf der Clientseite ist, dass es zu einer Trennung des Speicherortes und des Verschlüsselungsortes kommt.

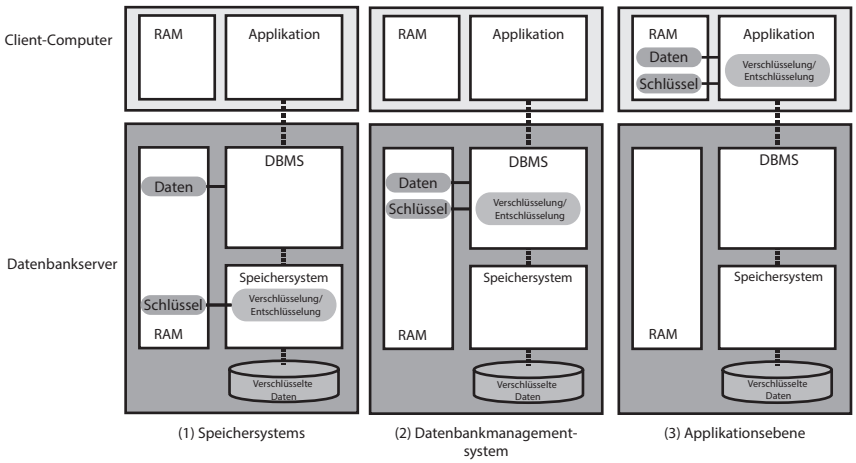

Abbildung 3.1: Abbildung zeigt die unterschiedlichen Ebenen, auf welchen Datenbankverschlüsselung durchgeführt werden kann. (adaptiert nach Biryukov et al. [22]).

In Abbildung 3.1 wird ein Überblick über die Ebenen, auf welchen Datenbankverschlüsselungen durchgeführt werden können ((1). Speichersystem, (2). Datenbankmanagementsystem, (3). Applikationsebene) gegeben. Die Abbildung zeigt, wo die Schlüssel gespeichert sind und zu welchem Zeitpunkt in weiterer Folge die Verschlüsselung der Daten stattfindet.

Der Ort der Verschlüsselung und die Speicherung der Verschlüsselungsschlüssel sind entscheidend für den Schutz der Daten. Die Speicherung der Schlüssel am Server hat den Nachteil, dass die Schlüssel von Unbefugten, z. B. Angreifern oder dem Administrator, eingesehen werden können, wodurch eine Entschlüsselung der Daten durchgeführt werden kann. Zusätzlich können die Daten zwar mithilfe von Transportverschlüsselung übertragen werden, jedoch ist nur die Übertragung verschlüsselt, anschließend werden sie am Server unverschlüsselt weiterverarbeitet. Dies ist unzureichend für ein DaaS-Szenario, da zu diesem Zeitpunkt Unbefugte wieder Zugriff auf die Daten erlangen können, bevor

diese auf der Datenbankmanagementebene erneut verschlüsselt werden. Folglich ist die reine serverseitige Verschlüsselung (Verschlüsselung auf der Ebene des Speichersystems und des Datenbankmanagementsystems) nicht hinreichend sicher.

Die Verschlüsselung auf der Clientseite eliminiert das Sicherheitsrisiko der serverseitigen Verschlüsselung, da es zu einer Trennung zwischen der Verschlüsselung, dem Speicherort des Schlüssels und der zu verschlüsselnden Daten kommt. Ist es Unbefugten möglich, Zugriff auf den Datenbankserver zu erhalten, können diese die Daten nicht entschlüsseln, da der Verschlüsselungsschlüssel auf der Clientseite gespeichert ist (Kriterium Schutz). Kommen die Daten bereits verschlüsselt am Server an, muss der Datenbankserver diese Daten nicht mehr verschlüsseln, was wiederrum bedeutet, dass diese zu keinem Zeitpunkt unverschlüsselt am Datenbankserver vorhanden sind.

Das Kriterium der *Verschlüsselungsgranularität* befasst sich mit dem Thema, wie feingranular eine Verschlüsselung auf Daten angewendet wird. Biryukov et al. erwähnen bei der Ebene des Datenbankmanagementsystems Granularitäten, die sich auf der Ebene des logischen Datenmodells befinden (Tabellen, Spalten und Zeilen). Diese werden in dieser Arbeit um die Granularität der Zellen erweitert. Des Weiteren wird bei diesem Kriterium die Ebene des Speichersystems von Biryukov et al. als Granularität erachtet. Folgende Aufzählung gibt einen zusammenfassenden Überblick über die in dieser Arbeit diskutierten Granularitäten [22, 33]:

Speichersystem - Diese Granularität umfasst die physische (z. B. Dateien) sowie die logische Abbildung (z. B. Tablespaces[3]) der Daten im Speicher der Datenbank.

Tabellen – Die Verschlüsselung der ganzen Tabellen [22].

Spalten - Eine weitere Granularität stellt die spaltenbasierte Verschlüsselung (auch *Column-level encryption* [28]) dar. Hierbei werden die Daten innerhalb einer Tabellenspalte mit demselben Schlüssel verschlüsselt.

Zeilen - Bei der zeilenbasierten Verschlüsselung werden alle Werte einer Datenbankzeile mit demselben Schlüssel verschlüsselt. Schneier [52] gibt folgendes Beispiel: Alle Elemente in einer Datenbankzeile werden mit dem Indexwert verschlüsselt; der Indexwert muss auch bekannt sein, um die Zeile zu entschlüsseln.

Zellen - Bei der zellenbasierten Verschlüsselung wird jedes Attribut, jeder Datensatz bzw. jeder Wert eines Datenbankeintrags mit einem anderen Schlüssel verschlüsselt. Dadurch ist es nicht möglich, Häufigkeitsanalysen auf die Daten durchzuführen, da jeder Wert einen anderen Schlüsseltext besitzt.

Die Granularität der verschlüsselten Daten spielt eine entscheidende Rolle für die Ausführungszeiten und Abfragefunktionalität. Die Ausführungszeiten werden durch eine feinere Granularität erhöht, da die Datensätze unterschiedlich oft entschlüsselt werden müssen (z. B. bei der Verschlüsselung der Daten auf Zellenbasis). Jedoch erhöht sich dadurch auch die Sicherheit der Daten. Ebenso wirkt sich die Granularität auf die Abfragefunktionalität der Datenbank aus, indem nur noch eine limitierte Anzahl an Abfragen möglich ist. Beispiele hierfür sind Punktabfragen[4] bei zellenbasierter Verschlüsselung

[3] Der Tablespace stellt eine logische Einheit dar, in welcher Oracle seine Daten abspeichert. [45]

[4] Punktabfrage stellen direkte Abfragen über den Primärschlüssel dar.

oder Gleichheitsabfragen[5] bei spaltenbasierter Verschlüsselung.

Das Kriterium *Austauschbarkeit des Algorithmus* beschreibt die Abhängigkeit von einem bestimmten Verschlüsselungsalgorithmus oder einer konkreten Implementierung. Der Algorithmus soll immer austauschbar sein für den Fall, dass die eingesetzte Implementierung nicht mehr hinreichend sicher sein sollte.

Das Kriterium, die *Abfragemächtigkeit*, umfasst die nachfolgendenden drei Fragen:

- Welche Art von Abfragen kann im unverschlüsselten Zustand durchgeführt werden?
- Welche Abfragen können im verschlüsselten Zustand durchgeführt werden?
- Welche Abfragen können im verschlüsselten Zustand nicht mehr durchgeführt werden?

Der vorletzte Punkt, welcher untersucht wird, ist die *Mehrbenutzerfähigkeit*. Diese befasst sich mit der Frage, ob es möglich ist, dass gleichzeitig mehrere Operationen (von unterschiedlichen Nutzern) auf verschlüsselte Daten durchgeführt werden können, ohne dass sie sich gegenseitig beeinflussen. Dieser Punkt ist abhängig von der Verschlüsselung der eingesetzten Datenbank (dem Zielsystem). Beispielsweise ist hier relevant wie nicht-idempotente Operationen, wie eine Addition auf einen Wert (z. B. `UPDATE ... SET Col = Col + 1`-Operationen), realisiert werden und wie diese auf verschlüsselte Werte durchgeführt werden bzw. möglich sein sollen.

Der letzte Punkt der *Zugriffskontrolle* beschäftigt sich mit der Frage, ob anhand der Verschlüsselung Benutzerrechte wie Nutzergruppen realisiert werden können.

In den nachfolgenden Abschnitten werden nun diverse Ansätze von Datenbankverschlüsselungen auf Basis der oben genannten Kriterien beschrieben.

3.2.2 Oracle Database

Die Datenbank von Oracle bietet die Verschlüsselung der Daten seit mehreren Versionen an. Oracle unterstützt diverse austauschbare symmetrische Blockverschlüsselungen wie z.B. DES und AES oder auch Strom-Chiffren [47]. Des Weiteren erlaubt Oracle eine Verschlüsselung der Daten in verschiedenen Granularitäten: innerhalb des logischen Datenmodells (Spalten und Zellen) und in Speichersystemnähe durch *Tablespaces*. Die Verschlüsslung von Spalten und Tablespaces ermöglicht Oracle mithilfe von *Transparent Data Encryption* (TDE) [53]. Die physische Speicherung der Daten erfolgt in *Datafiles*, ein Tablespace kann aus mehreren dieser Datafiles bestehen [45]. Die Verschlüsselung durch TDE erfolgt durch den Einsatz eines Master-Verschlüsselungsschlüssels. Dieser Schlüssel kann einerseits in einem sogenannten *Wallet* oder andererseits in einem *Hardware security module* (HSM) [49] gespeichert werden. Das Wallet ist eine Datei und befindet sich außerhalb der Datenbank [53]. Ein HSM ist ein physisches Modul, welches das sichere Speichern des Master-Verschlüsselungsschlüssels erlaubt [49]. Der im HSM gespeicherte Master-Verschlüsselungsschlüssel wird zur Verschlüsselung der eigentlichen Verschlüsselungsschlüssel der Daten am Datenbankserver eingesetzt [22]. Diese Schlüssel werden

[5] Gleichheitsabfragen sind Abfragen welche mit dem *Equal*-Operator durchgeführt werden können und eine Menge an Ergebnissen erwartet wird.

zur Ver- und Entschlüsselungszeit dynamisch vom HSM durch den Master-Schlüssel entschlüsselt. Nach erfolgreicher Durchführung des Verschlüsselungsprozesses werden die entschlüsselten Schlüssel aus dem Serverspeicher entfernt [22]. Der Verschlüsselungsprozess durch TDE wird vollständig serverseitig durchgeführt und hat nachfolgende Vorteile [53].

- Durch den Einsatz von TDE auf Tablespace-Ebene ist die Vertraulichkeit der Daten garantiert, sollte es zum Verlust der Festplatten oder physischen Dateien kommen.
- TDE ermöglicht ebenfalls die automatische Verschlüsselung sensibler Daten in Spalten, wie z. B. von Kreditkartendaten.
- Die Verschlüsselung mithilfe von TDE ist für den Anwendungsentwickler vollkommen transparent. Änderungen in den Daten werden automatisch verschlüsselt, der Entwickler muss nichts selbst implementieren, um Verschlüsselung zu realisieren.
- Nutzer der Datenbank erfahren keine Einschränkungen der Abfragemächtigkeit durch die Verschlüsselung der Tablespaces. Es sind dieselben Abfragen durch SQL möglich, wie im unverschlüsselten Zustand.
- Externe Applikationen, welche die durch TDE verschlüsselte Datenbank nutzen, müssen nicht adaptiert werden. Die Ver-und Entschlüsselung wird von der Datenbank verwaltet.

Die Abfragemächtigkeit durch die Verschlüsselung von TDE bei Spalten ist jedoch eingeschränkt. Die spaltenbasierte Verschlüsselung ist in ihren Datentypen limitiert[6] und unterstützt nur B-Tree Indizes für Gleichheitsabfragen (Equality Search) [53]. Wird eine TDE verschlüsselte Spalte als Index verwendet, verschlüsselt die Datenbank die Abfrageparameter dementsprechend und durchsucht den Index. Bereichsabfragen werden durch diese spaltenbasierte Verschlüsselung nicht unterstützt [49]. Des Weiteren reduziert der Einsatz von TDE nicht die Mehrbenutzerfähigkeit der Datenbank. Multiple Nutzer können Operationen auf die Daten durchführen, ohne dass sich diese Operationen beeinflussen können. Des Weiteren können Zugriffsrechte für Nutzer definiert werden. Nutzer können, abhängig von den definierten Zugriffsberechtigungen, Zugriff auf die verschlüsselten Daten erlangen [53].

Die letzte Granularität, welche Oracle unterstützt, ist die der Zellen. Im Vergleich zu TDE muss diese Verschlüsselung von DB-Entwicklern explizit implementiert und auf die benötigten Zellen angewendet werden. Hierzu wird die *DBMS_CRYPTO*-Schnittstelle eingesetzt [48]. Diese Schnittstelle bietet Funktionen zur Ver- und Entschlüsselung der Daten [47]. Es werden Funktionen angeboten, die den Einsatz eines Initialisierungsvektors (IV) erlauben. Beim Aufruf dieser Funktionen muss der Entwickler den Schlüssel wie auch den IV übergeben. Daher sind dem Server zum Zeitpunkt der Verschlüsselung die Schlüssel der Daten bekannt. Zusätzlich ist die Abfragemächtigkeit von Zellen eingeschränkt auf Punktvergleiche, wenn jeder Datensatz mit einem anderen IV verschlüsselt wurde. Der Vorteil dieser Verschlüsselung, in Kombination mit unterschiedlichen IVs pro Zellenwert, ist es, dass keine Muster innerhalb der Datenbank erkennbar sind. Die

[6] Es werden folgende Datentypen unterstützt: Binary_Float, Binary_Double, Char, Date, NChar, Number, NVarchar2, Raw, Timestamp, Varchar2, Blob und Clob [53].

Mehrbenutzerfähigkeit und die Zugriffskontrolle bei dieser Art von Verschlüsslung ist im Vergleich zu jener der TDE limitiert. Der Anwender dieser Schnittstelle muss explizit den Schlüssel zwischen mehreren Nutzern teilen, damit diese Zugriff auf die Daten erlangen können. Des Weiteren ist es schwer, Nutzerrechte auf Zellenbasis zu definieren, daher muss zumindest der verschlüsselte Spaltenname bekannt sein, um Rechte darauf festlegen zu können. Wird den Nutzern Zugriff auf diese Funktionen gewährt und auch definiert, auf welche Spalten er Zugriff hat, ist die Mehrbenutzerfähigkeit gegeben und die Zugriffskontrolle kann realisiert werden.

Der Schutz der Daten ist weder durch TDE (Tablespace oder spaltenbasierte Verschlüsselung) noch durch zellenbasierte Verschlüsselung gegeben. Die Verschlüsselung des Tablespace und der Zellen erlaubt zwar keine Rückschlüsse durch Muster im Schlüsseltext, jedoch ist es möglich, die Daten vor der Verschlüsselung abzufangen, da diese zum Zeitpunkt des Eintreffens am Datenbankserver unverschlüsselt und die Verschlüsselungsschlüssel dem Datenbankserver bekannt sind. Bei Oracle werden die Daten rein serverseitig verschlüsselt, auch die Verschlüsselungsschlüssel sind serverseitig oder auf einem externen serverseitigen Modul (HSM) zu finden.

3.2.3 Microsoft SQLServer

Genau wie die proprietäre Datenbank von Oracle bietet auch der Microsoft SQLServer diverse Verschlüsselungsstrategien und austauschbare Blockverschlüsselungen wie AES und DES an [28]. Auch diese Datenbank bietet die Verschlüsselung der Granularitäten auf der Ebene des logischen Datenmodells (Spalten und Zellen) sowie auch Verschlüsselung in Speichersystemnähe durch *Transparent Data Encryption* (TDE) an.

Die Verschlüsselung mithilfe von TDE unterscheidet sich jedoch funktionell von jener bei Oracle. Bei der TDE-Verschlüsselung von Microsoft wird die gesamte Datenbank (die *Datenbankpages*) inklusive der Log-Dateien verschlüsselt [28, 43]. Bei Datenbankpages handelt es sich um die Speichereinheit, in welcher der SQLServer die Daten ablegt [42]. Die Verschlüsselung mithilfe von TDE erfolgt unter Einsatz eines *Database Encryption Key* (DEK). Der DEK ist ein symmetrischer Schlüssel, der innerhalb der Datenbank gespeichert wird [43]. Dieser kann von externen Modulen wie einem HSM [22] gesichert werden. Der Einsatz von der TDE-Verschlüsselung hat keinen Einfluss auf die Abfragemächtigkeit und die Mehrbenutzerfähigkeit sowie die Zugriffskontrolle. Wie bei Oracle wird die Verschlüsselung auch bei Microsoft serverseitig durchgeführt.

Die Verschlüsselung der Daten in den Granularitäten der Spalten und Zellen kann mithilfe von Datenbankfunktionen wie *EncryptByKey* durchgeführt werden. Der Einsatz dieser Funktionen und der Zugriff auf den Verschlüsselungsschlüssel muss explizit vom Datenbank-Entwickler durchgeführt werden. Dies hat Einfluss auf die Abfragemächtigkeit, den Mehrbenutzerbetrieb sowie die Zugriffskontrolle. Die Abfragemächtigkeit wird bei der Granularität der Spalten auf Gleichheitsabfragen eingeschränkt. Bei der Granularität der Zellen sind nur Punktabfragen durchführbar. Genau wie bei Oracle kommt es hier zu einer Einschränkung des Mehrbenutzerbetriebs und der Zugriffskontrolle. Nutzer müssen explizite Rechte haben, um auf die Schlüssel und definierte Spalten der Tabellen zuzugreifen und Abfragen auf die Daten durchführen zu können. Sind diese Rechte

vorhanden, verhält sich die Datenbank bei Mehrbenutzerbetrieb wie erwartet. Multiple Nutzer können Operationen auf den Daten durchführen ohne dass sie sich gegenseitig beeinflussen.

Der Schutz der Daten beim Microsoft SQLServer ist durch TDE oder die manuelle Verschlüsselung, mithilfe von Datenbankfunktionen, basierend auf den definierten Kriterien, nicht gegeben. Die Verschlüsselung der Datenbankpages und der Zellen erlaubt zwar keine Rückschlüsse auf die Daten, jedoch sind der Datenbank die Verschlüsselungsschlüssel sowie die Daten vor der Verschlüsselung bekannt. Die Verschlüsselung wird, wie bei Oracle, rein serverseitig durchgeführt.

3.2.4 CryptDB

Die CryptDB ist ein verschlüsselter Proxy-Server, welcher auf Basis von MySQL realisiert ist. Das Ziel dieses Proxy-Servers ist es, den höchstmöglichen Schutz der Daten zu garantieren und dabei die Abfragemächtigkeit der Datenbank nicht einzuschränken.

Die CryptDB bietet zwei unterschiedliche Blockverschlüsselungen an (AES und Blowfish), ebenso wie andere Verschlüsselungsalgorithmen, welche mathematische Operationen wie Additionen und Bereichsvergleiche auf die verschlüsselten Daten (vgl. Homomorphe Verschlüsselung 3.1.3) erlauben [50]. Die CryptDB setzt großteils den AES ein, außer für Integer-Werte, hier kommt Blowfish zum Einsatz. Neben diesen Details wird die Austauschbarkeit der genannten Algorithmen von Popa et al. nicht weiter besprochen [50], jedoch ist ersichtlich, dass die Algorithmen für die homomorphe Verschlüsselung eine starke Abhängigkeit erzeugen.

Die Verschlüsselung der Datensätze ist abhängig von den gewünschten Abfragearten. Bei der Erstellung einer Tabelle muss für jede Spalte definiert werden, welche Abfragen auf diese durchgeführt werden können. Dies wird in Popa et al. als sogenannte *SQL-aware encryption strategy* bezeichnet. Dies bedeutet, die CryptDB verschlüsselt jeden Datenwert in der Datenbank in einer Art und Weise, die Abfragen darauf zulässt. Hier kommen unterschiedliche Verschlüsselungsgranularitäten zum Einsatz, wie zellenbasierte und spaltenbasierte Verschlüsselung. Eine spaltenbasierte Verschlüsselung mit gleichen Schlüsseln erlaubt z. B. die Realisierung von Joins oder Group-By Abfragen. Eine zellenbasierte Verschlüsselung bietet die höchste Sicherheitsstufe, da keine Rückschlüsse auf die Daten möglich sind, weil wiederum für jeden Wert ein anderer Schlüssel verwendet wurde. Durch diese SQL-aware encryption erlaubt CryptDB zahlreiche SQL-Operationen wie Gleichheits-, Ordnungs-, Such-, Bereichs- und Konjunktive-Abfragen sowie SQL-Funktionen (Min, Max, Sum etc.) [50]. Zeichenketten-Operationen sind auf die `like`-Operation limitiert, es können keine beliebigen regulären Ausdrücke formuliert werden. Die Realisierung von Bereichsabfragen wird mithilfe einer Order-Preserving Encrpytion Verschlüsselung (siehe OPE im Abschnitt Homomorphe Verschlüsselung 3.1.3) ermöglicht.

Die Schlüssel werden in der CryptDB auf einem eigenen Proxy-Server gespeichert. Der Proxy-Server ist auch der Applikationsserver, mit dem der Nutzer (z. B. über eine Applikation) kommuniziert [50]. Ein Teil der Schlüssel (des Passworts des Benutzers) ist clientseitig, auf der Applikationsebene, gespeichert. Auf dem Applikationsserver wird

die konkrete Schlüsselerstellung vorgenommen. Auf diesem Proxy-Server wird auch die Ver- und Entschlüsselung der Daten durchgeführt [50]. Somit führt die CryptDB die Verschlüsselung nicht direkt am Datenbankserver durch, wodurch es zu einer Trennung des Speicherortes der Daten und des Verschlüsselungsortes kommt.

In der CryptDB wurde ebenfalls ein Ansatz zur Zugriffskontrolle implementiert [50]. In diesem besitzen die Benutzer einen eigenen Schlüssel, mit dem die Daten verschlüsselt werden (Zugriffskontrolle). Um mehreren Benutzern und Benutzergruppen Zugriff zu ermöglichen, müssen *Principals* bei der Erstellung des Datenbankschemas angegeben werden. Sie regeln die Benutzerrechte pro Datensatz (Tabelle und Spalte). Der Benutzer einer Principal-Gruppe authentifiziert sich beim Proxy-Server mit seinem eigenen Passwort. Der Server validiert den Nutzer anschließend und gewährt ihm nach erfolgreicher Anmeldung Zugriff auf die Daten. Die Durchführung von Operationen auf verschlüsselte Daten erfolgt seitens Datenbankserver, daher ist auch die Mehrbenutzerfähigkeit gegeben, da sich zwei Nutzer gegenseitig bei Operationen nicht beeinflussen können.

Der Schutz der Daten innerhalb der CryptDB ist abhängig von der Konfiguration der CryptDB gegeben. Kommt es zu einer Trennung des Implementierungsorts des Proxy-Servers, auf dem die Verschlüsselung und Entschlüsselung durchgeführt wird, und des Datenbankservers, ist der Schutz der Daten gegeben.

3.2.5 Bucketing

Das Bucketing ist ein zeilenbasierter Verschlüsselungsansatz, der für relationale Datenbanken entwickelt wurde, um Abfragen auf verschlüsselte Daten zu realisieren. Die Verschlüsselung kann mit einem beliebigen symmetrischen Algorithmus durchgeführt werden. Dieser Verschlüsselungsansatz ist daher unabhängig vom eingesetzten Algorithmus. Die Kernidee dieses Ansatzes ist es, alle Einträge (Zellen) einer Datenbankzeile in eine binäre Zeichenkette zu verschlüsseln. Die Ver- und Entschlüsselung der Daten erfolgt vollkommen clientseitig. Zusätzlich wird pro Eintrag (Zelle) in der Datenbankzeile eine weitere Spalte mit Zusatzinformationen geschrieben. Die Information erlaubt die Realisierung von Abfragen, indem eine Vorselektierung am Server durchgeführt wird. Solche Datenbankzeilen sind in Abbildung 3.2 ersichtlich.

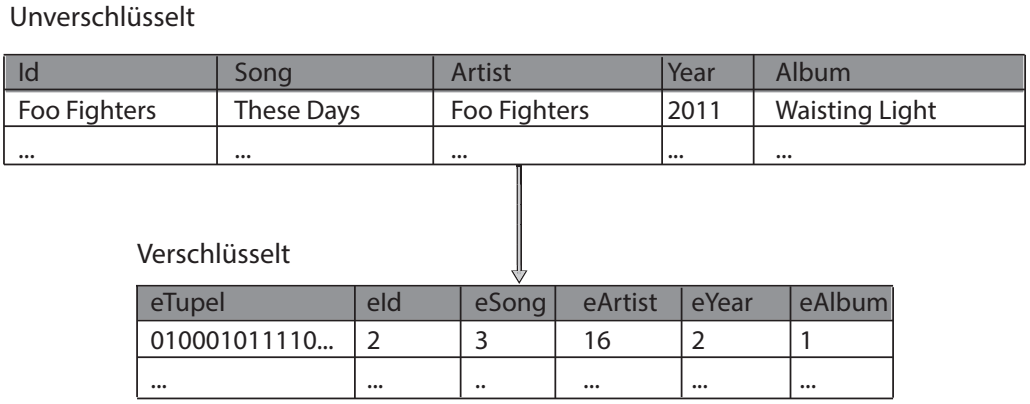

Abbildung 3.2: Der Bucketing-Ansatz nach Hacigumus et al. [36].

In der Abbildung 3.2 wird die gesamte Datenbankzeile in eine verschlüsselte Form gebracht und in der Zelle eTupel geschrieben. Pro Datenbankzelle (z. B. Song) wird eine kodierte Zusatzinformation (z. B. eSong) geschrieben. Die Zusatzinformation wird in numerischer Form abgebildet und kann daher als numerischer Index betrachtet werden. Es handelt sich hierbei um einen Wert, der den verschlüsselten Wert in einem Wertebereich abbildet. Um die Aufteilung der Daten in einem Wertebereich zu realisieren, wird eine *Partitions-Funktion* eingesetzt. Ein Wert innerhalb dieses Bereichs wird auch als *Bucket* bezeichnet. Daher wird dieser Ansatz auch als *Bucketing* bezeichnet. Die Partitions-Funktion kann auf jeder beliebigen Histogramm-Technik basieren (z. B. *Equi-Width*).

Um einen Datenwert innerhalb eines Wertebereichs zu identifizieren, wird eine *Identifikationsfunktion* eingesetzt. In Abbildung 3.3 wird eine beispielhafte Identifikationsfunktion auf einem partitionierten Domainbereich gezeigt. In dieser Abbildung ist erkennbar, dass für den Wertebereich des Attributs `Year` die Integer null, eins und zwei zugeordnet werden. Dem ersten Bereich 1900 bis 1950 ordnet die Identifikationsfunktion den Integer null zu. Beim Bereich 1950 bis 2000 ordnet sie den Integer eins zu usw.. Die Identifikationsfunktion kann mithilfe eines kollisionsfreien Hash-Werts berechnet werden.

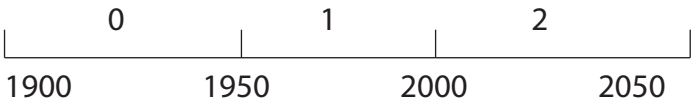

Abbildung 3.3: Die Identifikationsfunktion von Hacigumus [36] ordnet Werte einem Domain-Bereich zu.

Abfragen auf diese Daten werden mithilfe der Zusatzinformationen realisiert. Hierzu werden die Nutzerabfragen clientseitig umgeschrieben (*Query-Rewriting*), um eine Identifikation der Daten zu ermöglichen. Am Server wird aus diesem Grund eine Vorselektierung der verschlüsselten Daten vorgenommen, die anschließend am Client nach ihrer Entschlüsselung nochmals im Detail gefiltert werden. Dadurch können Operationen

wie Bereichs-, Punkt- und Konjunktiv-Abfragen durchgeführt werden, sowie Mengenoperationen (Set-Differences, Union, Projektion von Spalten), Sortierungsoperationen, Joins und auch Aggregation und Gruppierungsfunktionen, wie Min und Max, realisiert werden [36]. Zeichenketten-Operationen wie *like* sind hingegen nicht realisiert. Eine Erweiterung des Ansatzes nach Hacigumus et al.[36] wird von Lianzhong et al. [39] gezeigt. Der Ansatz wird um ein *Security Dictionary* erweitert. In diesem Dictionary sind folgende Informationen abgespeichert:

- Die Datentypen,
- die Verschlüsselungsalgorithmen,
- welche Spalten verschlüsselt worden sind (selektive spaltenorientierte Verschlüsselung) und
- die Namen der Spalten über die ein verschlüsselter Index nach Hacigumus et al.[36] erstellt worden ist.

Der Einsatz eines *Security Dictionary* erlaubt es, nur auf benötigten Spalten einen Index zu erstellen. Dadurch kommt es zu einer selektiven Auswahl an preisgegebenen Informationen, da nicht alle Spalten mit einem Bucketing-Index hinterlegt werden.

Der Bucketing-Ansatz wurde speziell für Datenbanken entwickelt, welche von Drittanbietern zur Verfügung gestellt werden, wie z. B. Cloud-Anbietern. Die Zugriffskontrolle wie auch die Realisierung der Mehrbenutzerfähigkeit ist hier abhängig von der eingesetzten Datenbank. Bei der Mehrbenutzerfähigkeit hängt es davon ab, wo die Datenbankoperationen (wie Additionen auf Daten) durchgeführt werden (am Client oder am Server) und wie die eingesetzte Datenbank diese Operationen schlussendlich durchführt. Bei der Realisierung der Zugriffskontrolle ist es entscheidend, wie die Rechte auf Datensätze abgebildet werden und wo bzw. wer Zugriff auf die Verschlüsselungsschlüssel hat.

Ein Rückschluss auf die Klartexte über die Schlüsseltexte ist nicht möglich, jedoch realisiert dieser Ansatz einen groben Index [36]. Dieser erlaubt zwar keine feingranulare statistische Auswertung, jedoch ist die Güte der Vorselektion der Daten abhängig von der Histogramm-Funktion. Ist diese zu feingranular, können detaillierte Rückschlüsse auf die Daten erfolgen. Daher ist der Schutz der Daten nur bedingt gegeben und implementierungsabhängig. Des Weiteren wird die Verschlüsselung und Entschlüsselung, sowie die Post-Filterung der durchgeführten Abfragen clientseitig durchgeführt. Der Server hat daher niemals Zugriff auf die unverschlüsselten Daten.

3.2.6 Structrue Preserving Database Encryption Scheme

Structrue Preserving Database Encryption Scheme (SPDE) ist ein zellenbasierter Ansatz, der die Zellkoordinaten (die Tabellen-ID, die Zeilen-ID und die Spalten-ID) als Ausgangsbasis für den Verschlüsselungsschlüssel einsetzt [33]. Diese Koordinaten werden einer Funktion übergeben, die einen Schlüssel daraus generiert. Dieser Ansatz kann mit unterschiedlichen symmetrischen Blockverschlüsselungen angewendet werden, wie z. B. AES oder DES und ist daher unabhängig von der eingesetzten Implementierung. In Abbildung 3.4 wird diese Idee dargestellt. Die erste Tabelle in dieser Abbildung zeigt einen

unverschlüsselten Datensatz. In der zweiten Tabelle werden die Daten nach einem naiven Ansatz verschlüsselt, gleiche Werte und Verschlüsselungsschlüssel resultieren in gleiche Schlüsseltexte. Die dritte Tabelle zeigt den SPDE-Ansatz basierend auf Zellkoordinaten. Es werden unterschiedliche Schlüsseltexte für gleiche Werte erzeugt. Um Datenintegrität zu garantieren, wird der Verschlüsselungsschlüssel, der aus den Zellkoordinaten des Datensatzes erstellt wurde, vor der Verschlüsselung mit dem Datensatz verkettet. Würde es zu einer Änderung des verschlüsselten Datensatzes kommen, oder würden zwei Datensätze ausgetauscht werden, resultiert dies in einen invaliden Datensatz. Bei der Entschlüsselung wird der auf den Zellkoordinaten basierende Verschlüsselungsschlüssel aus dem Datensatz entfernt und es wird der originale Datensatz retourniert.

Tabelle T
vor Verschlüsselung

Zeile	C
0	16
1	85
2	37
3	16
4	16
5	92
6	37
7	50
8	24
9	86

Tabelle T
nach Verschlüsselung
(naiver Ansatz)

Zeile	C
0	#$
1	]{
2	&*
3	#$
4	#$
5	^%
6	&*
7	0-
8	+=
9	@!

Tabelle T
nach Verschlüsselung
mit SPDE-Ansatz

Zeile	C
0	!#
1	:]
2	&*
3	„/
4	~?
5	|^
6	>\
7	@=
8	){
9	-+

Abbildung 3.4: Abbildung zeigt die Verschlüsselung *Structrue Preserving Database Encryption Scheme* nach Elovici et al. (adaptiert nach [33]).

Aufbauend auf dieser zellenbasierten Idee wurde von Elovici et al. ein Index entwickelt, der Punkt- und Bereichs-Abfragen ermöglicht [33]. Durch diese koordinatenorientierte Verschlüsselung wird eine Baumstruktur (B+-Baum) pro Tabellenspalte erstellt, wobei jeder Knoten auf seinen nachfolgenden Datensatz und dessen Wert verweist.

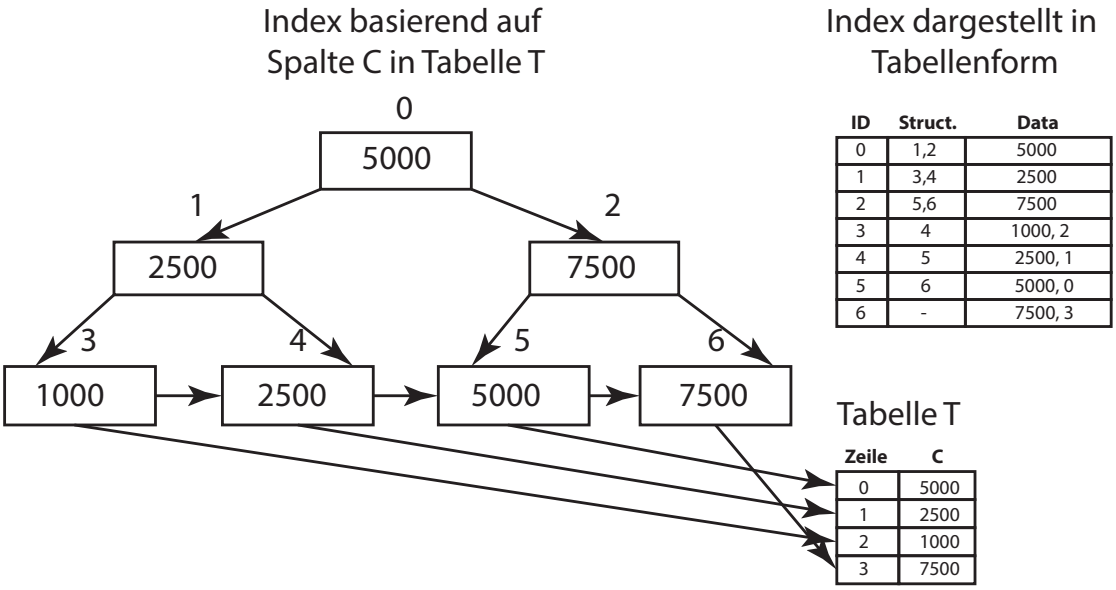

Abbildung 3.5: Index, Index-Tabelle und Referenz-Tabelle nach SPDE. Abbildung adaptiert nach [33].

Dies ist in Abbildung 3.5 ersichtlich. Die Blätter der Baumstruktur verweisen in ihren Datensätzen (in Abbildung 3.5 die Tabellenspalte *Data*) auf die Koordinaten des originalen Datensatzes, indem die referenzierte Zeilen-ID verkettet wird. Dies ermöglicht die Identifikation der referenzierten Daten. Die Werte innerhalb der Indizes werden mithilfe der Zeilen-ID des Index verschlüsselt. Somit können keine statistischen Auswertungen durchgeführt werden, da idente Datensätze in unterschiedliche Schlüsseltexte resultieren. Die Spalte *Struct* in Abbildung 3.5 verweist auf die nachfolgenden Konten des Datensatzes. Diese Spalte wird auf Basis einer Ordnungsfunktion erstellt.

Der Verschlüsselungsort und der Speicherort der Schlüssel werden in [33] nicht weiter diskutiert. Elovici et al. erörtern, dass die Struktur der Datenbank für den Einsatz dieses Verschlüsselungsansatzes nicht geändert werden muss [33]. Einzig die Implementierung der Ver- und Entschlüsselungsmethoden muss vom Nutzer vorgenommen werden. Somit entscheidet die Implementierung dieser Methoden, wo die Verschlüsselung stattfindet und wo die Schlüssel gespeichert werden.

Um Zugriffskontrolle zu ermöglichen, erörtern Elovici et al., dass die Spaltenschlüssel an die Zugriffsrechte gebunden werden können [33]. Der Ansatz geht davon aus, dass jede Spalte unterschiedliche Verschlüsselungsschlüssel verwendet. Durch die Verknüpfung gleicher Schlüssel mit den Zugriffsrechten der Nutzer kann die Anzahl verschiedener Verschlüsselungsschlüssel reduziert werden, indem sich mehrere Nutzer die gleichen Schlüssel teilen. Die Realisierung der Mehrbenutzerfähigkeit ist, wie auch beim Bucketing-Ansatz, abhängig von der eingesetzten Datenbank und dem Realisierungsort der Datenbankoperationen (beispielsweise ob die Addition auf verschlüsselte Daten client- oder serverseitig durchgeführt wird).

Der Schutz der Daten ist bei diesem Ansatz von der Implementierung der Ver- und Entschlüsselungsmethoden abhängig. Hier sind der Ort der Verschlüsselung und die Speicherung der Schlüssel entscheidend für die Sicherheit der Daten. Dies wird von Elovici et al. in [33] nicht weiter erörtert. Daher ist der Schutz der Daten implementierungsabhängig. Nichtsdestotrotz können keine Rückschlüsse auf die Daten durch die zellenbasierte Verschlüsselung und den unterschiedlichen Chiffren pro Zelle durchgeführt werden. Dies gilt sowohl für die Datenbankverschlüsselung als auch für den verschlüsselten Index.

3.2.7 Hierarchically Derived Symmetric Encryption

Die *hierarchisch abgeleitete symmetrische Verschlüsselung (hierarchically derived symmetric encryption (HDSE))* ist ein zellenbasierter Verschlüsselungsansatz, der am Institut der Johannes Kepler Universität Linz für Wirtschaftsinformatik und Data & Knowledge Engineering entwickelt wurde. Die Verschlüsselung erfolgt mithilfe von symmetrischen Blockverschlüsselungsalgorithmen. Dieser Ansatz ist daher unabhängig vom konkreten Verschlüsselungsalgorithmus. Die Verschlüsselung und Entschlüsselung der Daten wird clientseitig durchgeführt. Dieser Ansatz verschlüsselt seine Dateneinträge auf Basis ihrer Verschachtelungshierarchie und ähnelt daher dem SPDE-Ansatz aus Abschnitt 3.2.6. Der SPDE-Ansatz setzt die Speicherstruktur (Zellenkoordinaten) als Verschlüsselungsschlüssel für die Daten ein.

Die Kernidee ist es, abhängig von der Verschachtelungstiefe des Datenmodells der Datenbank, die Verschlüsselungsschlüssel für einen Datenwert abzuleiten und diesen damit zu verschlüsseln. Dadurch wird für jeden gleichen Datenwert ein unterschiedlicher Schlüsseltext erstellt. Dies wird durch die Einzigartigkeit des Primärschlüssels ermöglicht.

Um die Schlüssel unabhängig von den definierten Daten zu generieren, speichert der Benutzer clientseitig einen geheimen, zufällig generierten IV (siehe Abschnitt 3.1.1). Die Schlüssel werden mit diesem IV verknüpft. Dadurch kann nur der Besitzer des IVs die Daten verschlüsseln und entschlüsseln. Der IV erfüllt daher die Funktion des initialen Verschlüsselungsschlüssel für die erste Hierarchiestufe und muss bei der ersten Anwendung dieses Verschlüsselunganzatzes erstellt und sicher gespeichert werden.

Durch die Geheimhaltung des IV wird die Zugriffskontrolle der Datenbank eingeschränkt. Um diese zu ermöglichen, muss der private IV zentral geteilt werden. Die Daten können nur mit diesem IV entschlüsselt werden. Wird dieser geteilt, ist die Zugriffskontrolle für zahlreiche Nutzer durch die eingesetzte Datenbank realisierbar. Jedoch ist die Mehrbenutzerfähigkeit, wie auch bei Bucketing und SPDE, abhängig von der eingesetzten Datenbank und wie Operationen auf die Daten realisiert werden (z. B. ob die Datenbank Transaktionen unterstützt). Erfolgt die Durchführung von Datenbankoperationen wie z. B. Additionen auf verschlüsselten Daten clientseitig, wobei der verschlüsselte Datensatz geladen und entschlüsselt werden muss und unterstützt die Datenbank keine Transaktionen, ist die Mehrbenutzerfähigkeit nicht gegeben. Da hier eine atomare Durchführung von Operationen nicht erfolgen kann. Können diese Datenbankoperationen atomar serverseitig durchgeführt werden z. B. indem Additionen auf verschlüsselte Daten durch die Verschlüsselung unterstützt sind (Homomorphe Algorithmen) ist eine

Realisierung von Mehrbenutzerfähigkeit möglich.

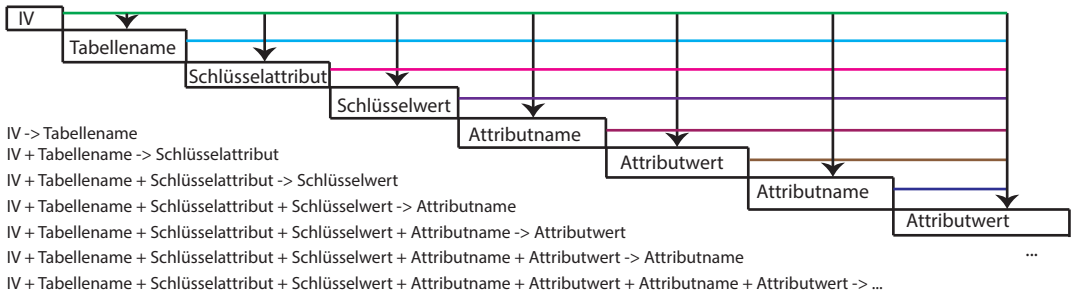

Abbildung 3.6: Eine beispielhafte Schlüsselhierarchie unter Einsatz des HDSE. Überkreuzende Linien pro Wert stellen die Schlüsselbestandteile für den resultierenden Verschlüsselungsschlüssel dar.

Die Erstellung eines solchen Verschlüsselungsschlüssels ist in Beispiel 3.6 ersichtlich. Dieses Beispiel zeigt ein einfaches auf Key-Value basierendes Datenmodell mit einer beliebigen Verschachtelungstiefe. Aus dieser Verschachtelungstiefe resultiert eine Schlüssel-hierarchie, welche aus Teilschlüsseln besteht. Durch eine Verkettung der Teilschlüssel wird der Verschlüsselungsschlüssel für einen Datensatz erstellt. Ein Teilschlüssel kann somit entweder ein Bezeichner eines Wertes (Schlüsselattribut oder Attributname), der Wert (Schlüsselwert oder Attributwert) selbst oder der Tabellenname sein. Wie in Beispiel 3.6 ersichtlich, ist die Kombination aller übergeordneten Werte der Verschlüsselungsschlüssel für den aktuellen Wert. Ein Schlüssel setzt sich beispielsweise wie folgt zusammen: der IV ist der Schlüssel für den Tabellennamen und der IV, kombiniert mit dem Tabellenamen, ist der Schlüssel für das Schlüsselattribut etc. Diese Verkettung bzw. Verschachtelung kann solange durchgeführt werden, wie es das Datenmodell der Datenbank erlaubt.

Die Realisierung von Abfragen ist abhängig vom Datenmodell. Erlaubt das Datenmodell mehrere Datensätze auf derselben Schlüsselhierarchie (wie z. B. zusammengesetzte Schlüssel der DynamoDB), wodurch gleiche Verschlüsselungsschlüssel erstellt werden können, sind Punkt- sowie Gleichheitsabfragen möglich. Ist dies nicht durch das Datenmodell gegeben, werden nur Punktabfragen sowie die selektive Auswahl von Elementen der Daten (Attributen) unterstützt. Komplexere Abfragen, wie Bereichsabfragen, sind durch die zellenbasierte Verschlüsselung nicht realisierbar.

Der Schutz der Daten ist abhängig vom verwendeten Datenmodell. Durch die Schlüsselhierarchie wird eine zellenbasierte Verschlüsselung erstellt, die abhängig vom Primärschlüssel ist. Erlaubt das Datenmodell jedoch auf der gleichen Schlüsselhierarchie gleiche Werte (z. B. zu einem Primärschlüssel gehören beliebig viele gleiche Attribute-Paare), können durch gleiche Verschlüsselungsschlüssel und Werte gleiche Schlüsseltexte entstehen. Dies erlaubt statistische Rückschlüsse auf die Daten. Erlaubt das Datenbankmodell dies nicht, sind keine Rückschlüsse auf die Daten möglich. Da die Verschlüsselung und Entschlüsselung vollkommen clientseitig durchgeführt wird, hat der Server zu keinem

Zeitpunkt Zugriff auf die unverschlüsselten Daten. Daher ist der Schutz der Daten nur bedingt gegeben, da statistische Rückschlüsse abhängig vom Datenmodell möglich sind.

3.2.8 Searchable Symmetric Encryption

Der *Searchable Symmetric Encryption* (SSE) Verschlüsselungsansatz [26] erlaubt die Erstellung eines abfragbaren verschlüsselten Index. Durch diesen Index können Konjunktiv-Abfragen auf symmetrische verschlüsselte Daten realisiert werden. Eine Erweiterung dieses Ansatzes erlaubt die Realisierung von ansonsten noch nicht unterstützten Abfragen, wie Bereichsabfragen und Substring-Matching [26].

Die Verschlüsselung der Daten und die Speicherung der Verschlüsselungsschlüssel findet clientseitig statt. Des Weiteren wird der eingesetzte symmetrische Verschlüsselungsalgorithmus vom Nutzer selbst definiert. Daher ist dieser Ansatz unabhängig von bestimmten Verschlüsselungsalgorithmen.

Die Idee ist es, eine Menge an Daten (z. B. Dokumenten oder Menge an Datenbankeinträgen) mithilfe eines Wertes zu identifizieren. Dieser Wert muss repräsentativ für den Datensatz sein, um eine Abfrage durchführen zu können. Um diese Einträge in der Datenbank darzustellen, benutzt der Ansatz sogenannte *T-Sets* (Tupel-Mengen), welche eine Liste von assoziierten Datenverweisen zu einem Schlüsselwort (Identifier) speichern. Mithilfe der T-Sets wird ein *erweiterter invertierter Index* aufgebaut. In Abbildung 3.7 wird ein Beispiel für einen invertieren Index gegeben. Der Index speichert zu einem beliebigen Wert eine Liste an Werten (z. B. Referenzen auf Datensätze). Die Datensätze (T-Sets) innerhalb eines Indexeintrages werden zeilenbasiert verschlüsselt. Der Verschlüsselungsschlüssel wird aus einem privaten clientseitigen Schlüssel in Kombination mit dem Schlüsselwort (Identifier) erstellt. Die Identifier im Index werden ebenfalls mit demselben Schlüssel im Index kodiert. Somit ist garantiert, dass der Server keine Informationen über die Daten besitzt.

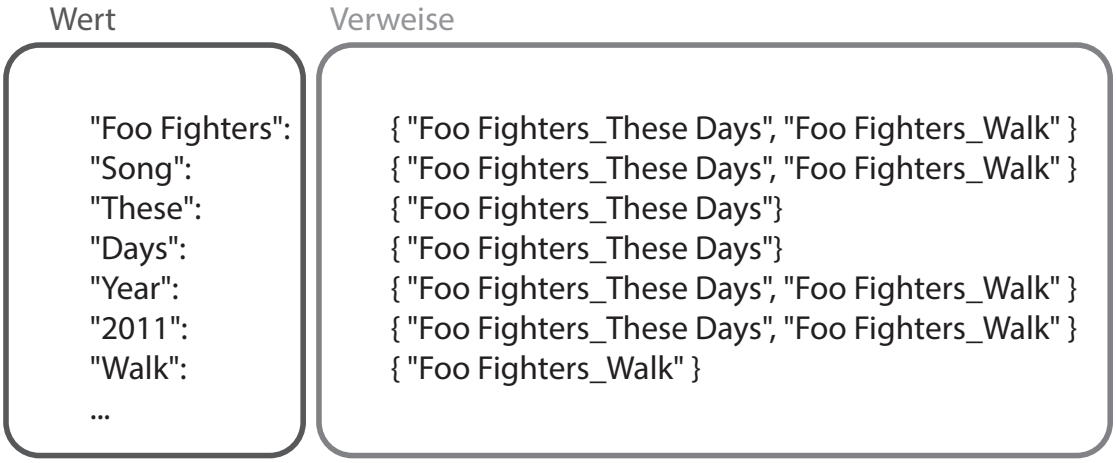

Abbildung 3.7: Die Abbildung zeigt einen invertierten Index. Jeder Wert ist ein eindeutiges Schlüsselwort, das eine Liste an Verweisen identifiziert.

Die Realisierung der Abfragen wird als Zwei-Parteien-Protokoll bezeichnet [26]. Der Server (Partei Eins) verwaltet nur die verschlüsselten Daten und Verweise auf diese und hat zu keinem Zeitpunkt Zugriff auf die unverschlüsselten Daten. Der Client (Partei Zwei) besitzt alle Schlüssel und Suchbegriffe (Werte). Er führt Abfragen auf den Index durch und lädt die Verweise auf die gefundenen Datensätze. Ein solcher Datensatzverweis wäre in Abbildung 3.7 der Wert *Foo Fighters_These Days*, welcher der Primärschlüssel ist. Nur der Client kann die verwiesenen Datensätze laden und entschlüsseln [26]. Diese Verweise zu einem Wert sind auch die Schwachstelle dieses Ansatzes. Durch die Anzahl der Verweise zu einem Schlüssel bzw. dem Wissen der Suchabfragen (meist gesuchten Werte) sind Rückschlüsse möglich. Dies ist eine allgemeine Schwachstelle aller beschriebenen Ansätze.

Die Zugriffskontrolle kann durch Erweiterungen am Ansatz realisiert werden. Dies wird ermöglicht, indem der Ersteller der Daten anderen Nutzern Abfragen erlaubt, wodurch er für bestimmte Abfragen Tokens generiert. Diese Tokens erlauben die Entschlüsselung der Daten [26]. Der SSE-Ansatz selbst erlaubt jedoch nur das Lesen und Schreiben des Schlüsselbesitzers [26]. Auch hier ist die Realisierung von Mehrbenutzerfähigkeit abhängig von der eingesetzten Datenbank und ob die Verwaltung und Aktualisierung des Index server- oder clientseitig durchgeführt wird.

Der Schutz der Daten ist bei diesem Ansatz gegeben. Zu keinem Zeitpunkt hat der Server Zugriff auf die unverschlüsselten Daten. Die Ver- und Entschlüsselung erfolgt vollkommen clientseitig. Durch eine zeilenbasierte Verschlüsselung sind statistische Rückschlüsse durch die Chiffren nur innerhalb eines Datensatzes möglich.

3.2.9 Zusammenfassung Verschlüsselung von Datenbanken

Die Tabelle 3.1 bildet einen zusammenfassenden Überblick über die Kriterien (vgl. Abschnitte 3.2.1), den Schutz der Daten, den Verschlüsselungsort, die Verschlüsselungsgranularität, die Austauschbarkeit des Algorithmus und die Mehrbenutzerfähigkeit. Das Kriterium der Abfragemächtigkeit wird in einer eigenen Tabelle 3.2 angeführt.

Verschlüsselungsansatz	Schutz	Verschlüsselungsort	Austauschbarkeit	Granularität	Mehrbenutzerfähigkeit	Zugriffskontrolle
Oracle	Nein	DBMS	Ja	Speichersystemnähe (TDE)	Ja	Ja
				Spalten (TDE)	Ja	Ja
				Zellen	Bedingt	Bedingt
Microsoft SQLServer	Nein	DBMS	Ja	Speichersystemnähe (TDE)	Ja	Ja
				Spalten	Bedingt	Bedingt
				Zellen	Bedingt	Bedingt
CryptDB	Bedingt	DBMS / Applikation	Bedingt	Spalten	Ja	Ja
				Zellen	Ja	Ja
Bucketing	Bedingt	Applikation	Ja	Zeilen	Bedingt	Bedingt
SPDE	Bedingt	NA	Ja	Zellen	Bedingt	Bedingt
HDSE	Bedingt	Applikation	Ja	Zellen	Bedingt	Bedingt
SSE	Ja	Applikation	Ja	Zeilen	Bedingt	Bedingt

Tabelle 3.1: Die Tabelle zeigt eine Übersicht über die beschriebenen Verschlüsselungsansätze und der in Abschnitt 3.2.1 beschriebenen Kriterien.

	Ja	Zutreffend
	Nein	Nicht zutreffend
Legende:	NA	Nicht angegeben
	Bedingt	Abhängig von diversen Parametern bzw. Implementierung, siehe Detailinformationen

Die in diesem Abschnitt beschriebenen Verschlüsselungsansätze werden in der Tabelle 3.2 in Bezug zur Verschlüsselungsgranularität gesetzt. Die Verschlüsselungsansätze von Oracle und Microsoft SQL Server, welche in Speichersystemnähe durchgeführt werden, erlauben SQL ohne Einschränkung.

Die CryptDB unterstützt durch ihren kombinierten Einsatz an Verschlüsselungsgranularitäten (SQL-aware encryption strategy) zahlreiche SQL-Funktionalitäten.

Das Bucketing stellt einen serverseitigen Ansatz dar, der eine grobe Filterung serverseitig und anschließend clientseitig durchführt. Dies erlaubt zahlreiche Abfragearten wie z. B. Bereichsabfragen.

Der SPDE-Index ist ein Index, welcher vorrangig für Bereichsabfragen implementiert wurde.

Der HDSE-Ansatz ist in seiner Abfragefunktionalität abhängig vom Datenmodell der Datenbank. Deshalb werden in dieser Tabelle bei HDSE nur Punktabfragen angeführt, da diese die Mindestfunktionalität ist.

Der SSE-Ansatz erlaubt serverseitige Punkt- sowie Konjunktiv-Abfragen. Der SSE ist ein Index, welcher das leichte Finden von Datensätzen mittels eines Bezeichners erlaubt.

Die Realisierung von Mehrbenutzerfähigkeit und Zugriffskontrolle ist bei den Ansätzen Bucketing, SPDE, HDSE und SSE abhängig von der eingesetzten Datenbank und der Durchführung der Operationen der Daten. Entscheidende Faktoren sind der Ort, an dem die Datenbankoperationen durchgeführt werden (Clientseite oder Serverseite), bzw. ob die Zugriffskontrolle auf die Verschlüsselungsschlüssel abgebildet werden kann, bzw. ob sich alle Nutzer einen Verschlüsselungsschlüssel teilen.

		Abfragearten								
	Granularität	**Punkt**	**Gleichheit**	**Bereich**	**Konjunktiv**	**Join**	**Mengenoperationen**	**Aggregation**	**Sortierung**	**Zeichenketten**
Oracle	Speichersystemnähe (TDE)	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja
	Spalten (TDE)	Ja	Ja	Nein	Ja	Ja	Ja	Nein	Nein	Nein
	Zellen	Ja	Nein	Nein	Nein	Nein	Nein	Nein	Nein	Nein
MS SQLServer	Speichersystemnähe (TDE)	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja
	Spalten	Ja	Ja	Nein	Ja	Ja	Ja	Nein	Nein	Nein
	Zellen	Ja	Nein	Nein	Nein	Nein	Nein	Nein	Nein	Nein
CryptDB	Spalten	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja
	Zellen	Ja	Nein	Nein	Nein	Nein	Nein	Nein	Nein	Nein
Bucketing	Zeilen	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Nein
SPDE-Index	Sym. Index	Ja	NA	Ja	NA	NA	NA	NA	NA	NA
HDSE	Zellen	Ja	Nein	Nein	Nein	Nein	Nein	Nein	Nein	Nein
SSE	Sym. Index	Ja	Ja	Nein	Ja	Nein	Nein	Nein	Nein	Nein

Tabelle 3.2: Tabelle zeigt die Übersicht über die Ansätze und die Abfragen in verschiedenen Variationen auf die Daten.

Legende:	Ja	Zutreffend
	Nein	Nicht zutreffend
	NA	Nicht angegeben

Kapitel 4

Der Verschlüsselungsclient für die DynamoDB

Die Amazon DynamoDB ist eine cloud-basierte NoSQL-Datenbank. Sie besitzt die NoSQL spezifischen Charakteristiken (nicht-relational, beinahe schemafrei, eine einfache API und *eventually consistent*). Diese Datenbank wurde bereits in Abschnitt 2.6 diskutiert. Eine clientseitige Verschlüsselung der Daten der DynamoDB wird nicht angeboten. Auf Basis der im Abschnitt 3 beschriebenen Verschlüsselungsverfahren wurde daher ein Verschlüsselungsclient – der EncryptedAmazonDynamoDbClient - entwickelt. Dieser realisiert nachfolgende Anforderungen:

- Der Server wird nicht als vertrauensvoll erachtet (*Honest But Curious* [37]). Dies bedeutet, dass der Server alle Informationen, die er bekommt, verarbeiten kann und wird.
- Die Verschlüsselung ist transparent für den Benutzer.
- Die eingesetzten Verschlüsslungen sind austauschbar.
- Der Verschlüsselungsclient erlaubt den Großteil der von der DynamoDB angebotenen Funktionalitäten, wie z.B. Bereichs- und Konjunktiv-Abfragen.

Im nachfolgenden Abschnitt wird die Architektur des Verschlüsselungsclients präsentiert. Zuerst wird gezeigt, wie der Kommunikationsablauf des unverschlüsselten DynamoDB Clients mit der Datenbank vonstatten geht. Darauf aufbauend wird erörtert, wie der Verschlüsselungsclient in diese Kommunikation eingreift. Anschließend wird erläutert, wie der Verschlüsselungsansatz HDSE auf die DynamoDB angewendet werden kann und welche Auswirkungen dieser Einsatz auf die Datenbankfunktionalität hat. Abschließend wird die Architektur des Verschlüsselungsclients präsentiert.

4.1 Konzeption

4.1.1 Kommunikation

In diesem Abschnitt wird der Kommunikationsablauf des unverschlüsselten DynamoDB Clients (AmazonDynamoDBClient) gezeigt und drauf aufbauend der Kommunikationsablauf des Verschlüsselungsclients (EncryptedAmazonDynamoDBClient) präsentiert.

Kommunikationsablauf des AmazonDynamoDBClient mit der DynamoDB

Die im Kapitel 2.6 beschriebene Datenbank Amazon DynamoDB ist eine NoSQL-Datenbank, die eine Schnittstelle namens *AmazonDynamoDBClient* für die Datenverwaltung und für Abfragen zur Verfügung stellt. Applikationen, die ihre Daten in der Amazon DynamoDB verwalten wollen, können diese DynamoDB-Schnittstelle verwenden. Sollen beispielsweise Daten in die Datenbank geschrieben werden, so wird eine Schreibeoperation in einer Applikation formuliert, welche die AmazonDynamoDBClient-Schnittstelle verwendet. Diese überträgt die Daten an die DynamoDB, die sich in der Amazon-Web-Service-Cloud (AWS-Cloud) befindet. Retourniert die Datenbank Daten, werden auch diese über den AmazonDynamoDBClient an die Applikation weitergegeben. Dieser Ablauf ist in Abbildung 4.1 ersichtlich.

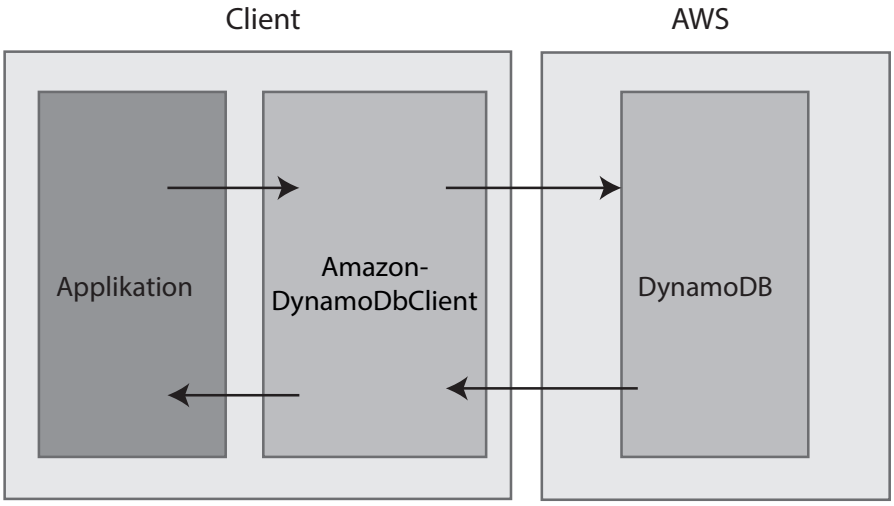

Abbildung 4.1: Überblick über die Kommunikation des *AmazonDynamoDBClient*, der Applikation und der AWS-Datenbank Amazon DynamoDB. Der Client stellt die Kommunikationsschnittstelle der Applikation und der DynamoDB Datenbank dar.

Kommunikationsablauf des EncryptedAmazonDynamoDBClient mit der DynamoDB

Der im Zuge dieser Arbeit entwickelte Verschlüsselungsclient *EncryptedAmazonDynamoDbClient* basiert auf der AmazonDynamoDBClient-Schnittstelle und ermöglicht die

Verschlüsselung von Daten. Er verwendet die HDSE-Verschlüsselung (siehe Abschnitt 3.2.7), und realisiert diese für die DynamoDB. In Abbildung 4.2 wird die Kommunikation zwischen Applikation und Datenbank mithilfe des Verschlüsselungsclients gezeigt. Im Vergleich zu Abbildung 4.1 wird die Kommunikation zur Datenbank hier direkt mit dem EncryptedAmazonDynamoDBClient und indirekt über den AmazonDynamoDBClient durchgeführt.

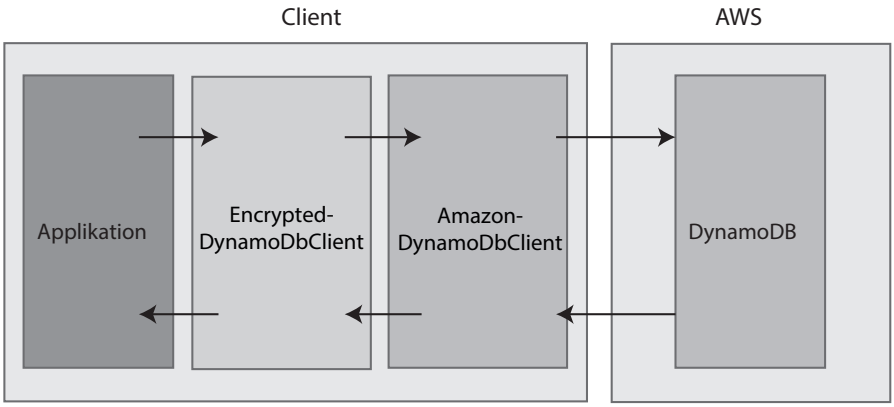

Abbildung 4.2: Überblick über die Kommunikation der Applikation, des *EncryptedAmazonDynamoDbClient*, des AmazonDynamoDBClient und der AWS-Datenbank Amazon DynamoDB. Der Verschlüsselungsclient fungiert als weiterer Mittelsmann zwischen dem AmazonDynamoDBClient und der Datenbank.

Der Verschlüsselungsclient baut somit auf der von Amazon zur Verfügung gestellten Schnittstelle auf. Die Art und Weise, wie die Daten in der DyanmoDB verschlüsselt werden, ist ausschlaggebend dafür, welche Funktionalität vom Verschlüsselungsclient angeboten werden kann. Im nachfolgenden Abschnitt wird daher gezeigt, wie der zellenbasierte Verschlüsselungsansatz HDSE auf das Datenmodell der DynamoDB angewandt wird.

4.1.2 Verschlüsselung der Daten und Indizes

Das Datenmodell der DynamoDB (siehe Abschnitt 2.6.1) besteht aus Tabellen, Datensätzen und Attributen. Der Zugriff auf diese Daten kann über zwei unterschiedliche Schlüsselschemen erfolgen: einem einfachen oder einem zusammengesetzten Schlüssel. Der Einsatz von HDSE auf das Datenmodell ist abhängig vom jeweiligen Schlüsselschema. In diesem Abschnitt werden die Schlüsselschemas anhand von Beispielen gezeigt.

Die DynamoDB verlangt bei der Tabellenerstellung die Angabe des Tabellennamens sowie des Schlüsselschemas und dessen Attribute. Bei einer Tabelle mit einem einfachen Schlüssel kann der Verschlüsselungsansatz ohne Adaptierung angewendet werden. In Abbildung 4.3 wird ein Datensatz mit einem einfachen Schlüssel gezeigt. Diese Abbildung zeigt auch die Schlüsselhierarchie für jedes Attribut.

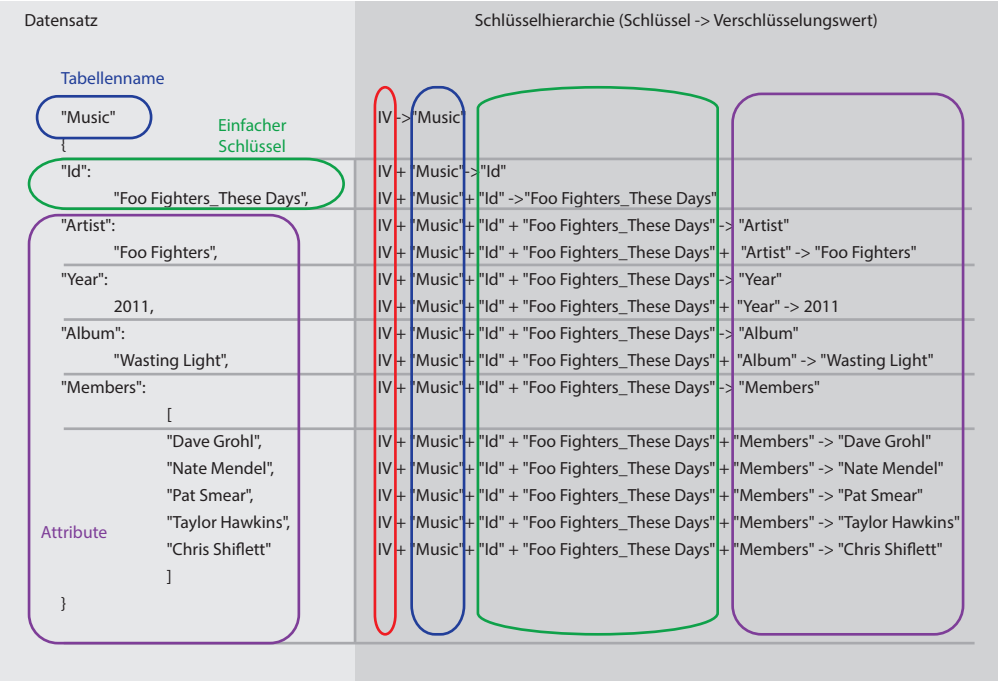

Abbildung 4.3: Schlüsselhierarchie des HDSE. Es wird ein Datensatz mit einem einfachen Schlüssel gezeigt. Die farbigen Kreise markieren die Verschlüsselungs-Bestandteile IV, Tabellenname, Primärschlüssel und Attribute.

Ein Verschlüsselungsschlüssel besteht aus Teilschlüsseln, welche, abhängig von der Hierarchieebene, der private IV, der Tabellenname, der Attributname und der Attributwert sind. Der Primärschlüssel (in Abbildung 4.3 das Attribut *Id*) ist ein fixer Schlüsselbestandteil (Teilschlüssel) für die weiteren Attribute. Durch die Einzigartigkeit des Primärschlüssels und der Datenbankeigenschaft, dass Attributduplikationen innerhalb eines Datensatzes nicht erlaubt sind, wird eine zellenbasierte Verschlüsselung auf Datensätze ermöglicht.

Im Vergleich zu einem Datensatz, der einen einfachen Primärschlüssel besitzt, kann der HDSE nicht ohne Adaptierung auf einen Datensatz mit einem zusammengesetzten Schlüssel angewendet werden. Die DynamoDB verlangt, wie bereits erörtert, bei der Tabellenerstellung die Definition der Schlüsselattribute. Der Attributname für das einfache Schlüsselschema kann ohne weitere Adaptierungen nach dem HDSE-Ansatz verschlüsselt werden (dieser ist nicht von einem übergeordneten variablen Wert abhängig). Der Schlüssel hierzu besteht aus privaten IV und den Tabellennamen. Bei einem zusammengesetzten Schlüsselschema müssen jedoch zwei Attributnamen (einfaches Schlüsselattribut und Bereichsschlüssel) bei der Tabellenerstellung definiert werden (siehe Abbildung 4.4). Hier ist entscheidend, dass der Datenbank zu jedem Zeitpunkt der Primärschlüssel des Datensatzes bekannt ist, daher wird der Bezeichner des Bereichsschlüssels auf eine andere Art und Weise kodiert. Dies und die gesamte Verschlüsselung eines Datensatzes mit zusam-

mengesetztem Schlüssel ist in Abbildung 4.4 ersichtlich. Es ist deutlich erkennbar, dass die Schlüsselhierarchie für das Attribut *Song* in diesem Beispiel angepasst wurde. Würde die Verschlüsselung des Bereichsattributes nicht auf diese Art und Weise durchgeführt werden, sondern nach dem HDSE-Ansatz, basierend seiner hierarchisch übergeordneten Werte, würde die Datenbank den Primärschlüssel (Bereichsschlüssel) nicht mehr aufgrund seiner unterschiedlichen Verschlüsselungen per Datensatz identifizieren können. Des Weiteren wäre es nicht möglich, eine Tabelle innerhalb der DynamoDB anzulegen, da diese, wie bereits erörtert, zu jedem Zeitpunkt den Primärschlüssel identifizieren können muss.

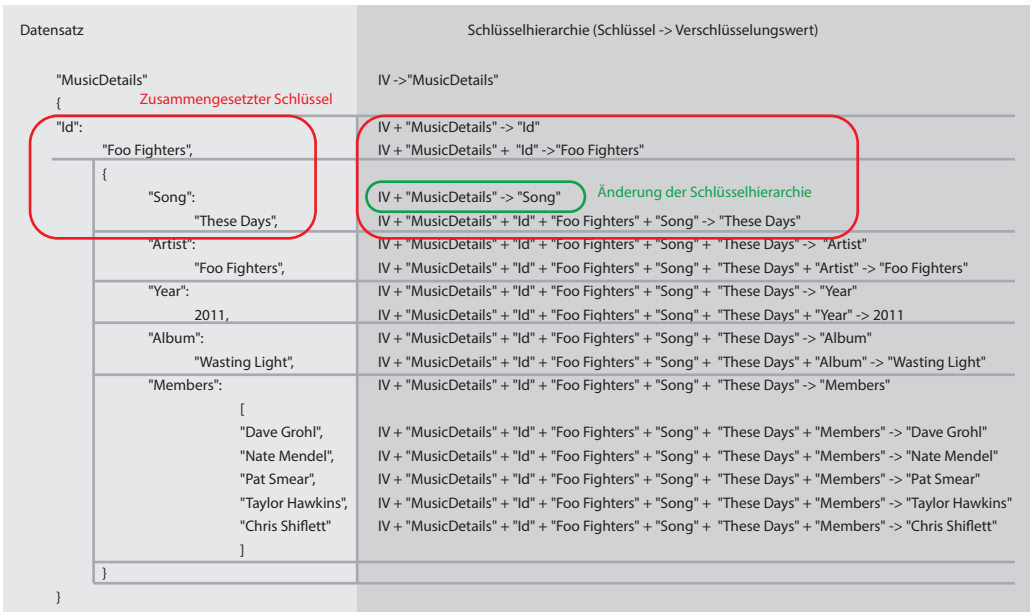

Abbildung 4.4: Schlüsselhierarchie des HDSE. Es wird ein Datensatz mit einem zusammengesetzten Schlüssel gezeigt.

Die weitere Verschlüsselung der Attribute erfolgt wie durch den HDSE definiert; auch der Wert des Bereichsschlüssels wird wie nach dem HDSE definiert verschlüsselt. Der Bereichsschlüssel (Attributname und Wert) wird als weiterer Teilschlüssel der Schlüsselhierarchie erachtet.

Bis auf die Ausnahme des Bereichsschlüsselattributnamens bei Tabellenerstellung kann der Verschlüsselungsansatz des HDSE ohne weitere Adaptionen auf das Datenmodell angewandt werden. Die Verschlüsselung hat, wie bereits erwähnt (siehe Kapitel 3), Auswirkungen auf die Funktionalität der Datenbank. In den nachfolgenden Abschnitten werden daher die Auswirkungen des Einsatzes des HDSE auf die Indizes der DynamoDB, die Datenbankoperationen und die Datenbankabfragen erörtert.

Datenbankindizes

Die DynamoDB erlaubt, wie in Abschnitt 2.6.2 beschrieben, zwei Arten von Indizes: Local Secondary Index und Global Secondary Index. Die Erstellung eines Sekundär-Index (z.B. auf ein Attribut) ist im Zuge einer Tabellenerstellung oder auf existierende Tabellen möglich. Durch den Einsatz des HDSE-Verschlüsselungsansatzes, der eine zellenbasierte Verschlüsselung vorsieht, wird jedes Attribut und sein spezifischer Wert zu einem anderen Bitmuster verschlüsselt. Daher kann der Einsatz von Sekundärindizes unter Einsatz des HDSE nicht verwendet werden, da die Datenbank den Attributnamen nicht mehr identifizieren kann. Wären die Attributnamen identifizierbar, könnte die Datenbank jedoch aufgrund der unterschiedlichen Schlüsseltexte keinerlei Vergleiche durchführen.

4.1.3 Unterstützung von Datenbankoperationen

PutItem

Die `PutItem`-Operation (vgl. `PutItem` des AmazonDynamoDBClient (in Abschnitt 2.6.4)) führt eine Schreibeoperation durch. Der Verschlüsselungsclient verwendet den Schlüssel aus dem übergebenen Datensatz als Basis für die Verschlüsselung der Attribute und ihrer Werte nach dem HDSE-Verschlüsselungsansatz. Der Schlüssel selbst wird, wie durch HDSE definiert, durch seine übergeordneten Hierarchieelemente verschlüsselt (IV, Tabellenname (Music), Schlüsselattributname (Id)). In Beispiel 4.6 ist ein unverschlüsselter Datensatz und ein verschlüsselter Datensatz ersichtlich. In diesem Beispiel ist der Primärschlüssel das Attribut *Id* mit dem Wert *Foo Fighters_These Days*.

```
1 {
2   TableName:  Music,
3   Item:
4     {
5       Year = { N: 2011, },
6       Members = { SS:
7             [
8               Dave Grohl,  Nate Mendel,
9               Pat Smear,  Taylor Hawkings,
10              Chris Shiflett
11            ],
12      },
13      Id = { S: Foo Fighters_These Days,},
14      Artist = { S: Foo Fighters,},
15      Album = { S: Wasting Light,}
16    },
17 }
```

(a) Codebeispiel einer unverschlüsselten PutItem-Operation.

```
{
  TableName: 0229e3f35599df57d239f369b72890fd,
  Item: {
      HWXzpvTnFk+qMoKJN1UusQ== = { BS:
                     [
                       44 61 76 65 20 47 72 6f 68 6c,
                       4e 61 74 65 20 4d 65 6e 64 65 6c,
                       50 61 74 20 53 6d 65 61 72,
                       54 61 79 6c 6f 72 20 48 61 77 6b 69 6e 67 73,
                       43 68 72 69 73 20 53 68 69 66 6c 65 74 74
                      ]},
      1vbVluVugzxhhOu5g+MKmw== =
        { B: 32 30 31 31 0d 0a,},
      OuvxfcUgnOg0A7abzM2/sg== =
        { B: 46 6f 6f 20 46 69 67 68 74 65 72 73 5f 54 68 65 73 65 20 44 61
     79 73,},
      tW93M07/2dXIo4LizVHEhw== =
        { B: 46 6f 6f 20 46 69 67 68 74 65 72 73 20,},
      aSqESWaLKZSpkWx5ncyV0A== =
        { B: 57 61 73 74 69 6e 67 20 4c 69 67 68 74,}
      },
}
```

(a) Codebeispiel einer verschlüsselten PutItem-Operation.

Abbildung 4.6: Request einer PutItem-Operation.

BatchWriteItem

Die `BatchWriteItem`-Operation (vgl. `BatchWriteItem` in Abschnitt 2.6.4) führt mehrere Schreibe- oder Löschoperationen auf eine oder mehrere Tabellen durch. Auch hier wird, wie bei der `PutItem`-Operation, der Schlüssel eines (zu schreibenden oder zu löschenden) Datensatzes als Basis für die Verschlüsselungshierarchie von HDSE verwendet. Anschließend wird die Verschlüsselung nach dem HDSE-Ansatz auf die Attribute des Datensatzes durchgeführt. Eine `BatchWriteItem`-Operation kann, wie bereits erwähnt, zahlreiche Schreibe bzw. Löschoperationen für mehrere Tabellen durchführen. Daher muss dieser Vorgang (der Schlüsselerstellung und der Verschlüsselung) für jede dieser Operationen pro Tabelle durchgeführt werden, da eine unterschiedliche Schlüsselhierarchie vorhanden ist.

UpdateItem

Die `UpdateItem`-Operation (vgl. `UpdateItem` in Abschnitt 2.6.4) aktualisiert einen existierenden Datensatz in der DynamoDB oder erstellt einen neuen Eintrag. Auch hier wird der Schlüssel des zu aktualisierenden Datensatzes als Basis für die Verschlüsselung nach

dem HDSE-Ansatz verwendet. Wie in Abschnitt 2.6.4 beschrieben, erlaubt diese Operation unterschiedliche Aktionen auf Datensätze (`ADD, DELETE` und `PUT`). Die `ADD`-Aktion erlaubt Addition auf numerische Werte und das Hinzufügen von neuen Elementen in Mengen. Die serverseitige Addition auf verschlüsselte (numerische) Werte kann nicht ohne zusätzliche Arbeit durchgeführt werden. Dies wird durch das clientseitige Auslesen und Entschlüsseln eines bestehenden Datensatzes und Addition der nicht verschlüsselten Werte realisiert. Additionen auf Mengen stellen durch die Eigenschaft der DynamoDB, dass innerhalb einer Menge keine Duplikate vorkommen dürfen, kein Problem dar. Dadurch ist gegeben, dass nie zwei gleiche verschlüsselte Werte in einer Menge gespeichert werden, wodurch eine Addition auf Mengen serverseitig realisierbar ist.

DeleteItem

Die `DeleteItem`-Operation (vgl. `DeleteItem` in Abschnitt 2.6.4) löscht Datensätze. Hier wird der Schlüssel des zu löschenden Datensatzes festgelegt. Dieser wird nach dem HDSE-Ansatz verschlüsselt und der Operation übergeben.

4.1.4 Unterstützung von Datenbankabfragen

GetItem

Die `GetItem`-Operation (vgl. `GetItem` in Abschnitt 2.6.5) erlaubt das Lesen eines Datensatzes aus der Datenbank. Zusätzlich ist es möglich, nur gewisse Attribute eines Datensatz zu laden. Um diesen Datensatz zu laden, wird der Primärschlüssel benötigt. Dieser Schlüssel wird nach dem Prinzip des HDSE-Ansatzes verschlüsselt. Ein Beispiel für eine verschlüsselte 4.2 und eine unverschlüsselte Abfrage 4.1 ist nachfolgend ersichtlich. Die Struktur der Abfrage wird nicht verändert, einzig die Werte werden verschlüsselt. Hier wird ebenfalls gezeigt, dass nur ein gewisses Attribut (das Attribut *Album* mit der Abfragefunktion `AttributesToGet`) des Datensatzes geladen wird. Um dieses Attribut zu laden, wird, basierend auf dem Primärschlüssel, der Verschlüsselungsschlüssel nach dem HDSE-Ansatz erstellt und der Attributname verschlüsselt.

```
1 {
2   TableName: Music,
3   Key:
4   {
5     Id =
6       {S: Foo Fighters_These Days,}
7   },
8   AttributesToGet: [Album],
9 }
```

Codebeispiel 4.1: Beispiel einer unverschlüsselten *GetItem*-Abfrage, die nach dem Attribut *Album*) über den Schlüssel `Foo Fighters_These Days` die Datenbank abfragt.

```
1 {
2   TableName:  0229e3f35599df57d239f369b72890fd,
3   Key:
4   {
5     OuvxfcUgnOg0A7abzM2/sg== =
6       { B: 4 65 20 4d 65 6e 64 65 6c 50 61 74 20 53 6d,}
7   },
8   AttributesToGet: [aSqESWaLKZSpkWx5ncyV0A==],
9 }
```

Codebeispiel 4.2: Beispiel einer verschlüsselten *GetItem*-Abfrage, die nach dem Attribut *Album* (in Verschlüsselter Variante *aSqESWaLKZSpkWx5ncyV0A==*) über den Schlüssel `Foo Fighters_These Days` die Datenbank abfragt.

Nachdem die Abfrage erfolgreich vom Server verarbeitet wurde, returniert diese den verschlüsselten Datensatz. Um diese Daten zu entschlüsseln, muss der zuvor erstellte Verschlüsselungsschlüssel wieder verwendet oder durch seine Hierarchie hergeleitet werden. Dem Verschlüsselungsclient ist zu jeder Zeit der private IV bekannt. Da dieser als erster Bestandteil der Schlüsselhierarchie verwendet wird, ist es möglich, den Tabellennamen zu entschlüsseln. Dieser Bestandteil kann anschließend zur weiteren hierarchischen Entschlüsselung verwendet werden. Die Entschlüsselung der Attribute und des Tabellennamens ist relativ einfach, da diese Werte als Zeichenkette von der DynamoDB abgespeichert werden. Die Entschlüsselung der Attributwerte stellt im Vergleich hierzu eine Herausforderung dar. Der Client muss wissen, in welchen Datentyp er die Daten entschlüsseln soll. Die DynamoDB erlaubt Binär, Zeichenketten und Numerische Datentypen. Um dies zu realisieren, muss sich der Verschlüsselungsclient merken, welche Datentypen bei welchem Attribut, zu welcher Tabelle benötigt werden. Dies wird durch ein Data-Dictionary, das clientseitig gespeichert wird, realisiert.

BatchGetItem

Die `BatchGetItem`-Operation (vgl. Abschnitt 2.6.5) ist eine Erweiterung der `GetItem`-Operation, die zahlreiche Leseoperationen auf Datensätze unterschiedlicher Tabellen durchführen kann. Ebenso wie bei der GetItem-Operation werden die übergebenen Primärschlüssel nach dem HDSE-Ansatz verschlüsselt. Wie bei der `BatchWriteItem`-Operation muss die Schlüsselerstellung und die Verschlüsselung für jeden übergebenen Primärschlüssel eines gesuchten Datensatzes der Tabelle durchgeführt werden.

Des Weiteren erlaubt diese Operation eine selektive Filterung von Attributen zahlreicher Datensätze innerhalb einer Tabelle. Diese Funktionalität kann von HDSE nicht ohne zusätzliche clientseitige Implementierungsarbeit unterstützt werden. HDSE führt, wie bereits erörtert, eine zellenbasierte Verschlüsselung der Daten durch. Diese Funktion der Operation durchsucht jedoch jeden Datensatz nach den definierten Attributen, welche durch diese Verschlüsselung nicht gefunden werden können. Gleich wie bei der `GetItem`-Operation benötigt der Verschlüsselungsclient zur Entschlüsselung der abgefragten Daten das Wissen über den Primärschlüssel und die Datentypen.

Query

Die `Query`-Operation erlaubt Abfragen auf die Schlüsselattribute (vgl. Abschnitt 2.6.5. Somit ist es möglich, Vergleichsabfragen (wie z.B. Gleichheitsabfragen oder Bereichsabfragen) auf den zweiten Bestandteil des zusammengesetzten Schlüssels durchzuführen. Bei einem zusammengesetzten Schlüssel wird, abhängig vom einfachen Schlüsselbestandteil, der Verschlüsselungsschlüssel für den Bereichsschlüssel erstellt. Dies wird auch in Beispiel 4.7 gezeigt. In diesem Beispiel sind zwei Datensätze in der Tabelle *MusicDetails* dargelegt. Beide besitzen den einfachen Schlüssel *Id: Foo Fighters* und als Bereichsschlüssel *Song:TheseDays* und *Song: Walk*. Per HDSE-Definition ist der Verschlüsselungsschlüssel für den einfachen Schlüssel (Attributname und Attribut Wert) wie auch für den Attributnamen des Bereichsschlüssels derselbe. Daher wird der gleiche Schlüssel für die Werte *Walk* und *These Days* erstellt, wodurch diese Werte vergleichbar sind. Es kommt somit implizit zu einer spaltenbasierten Verschlüsselung.

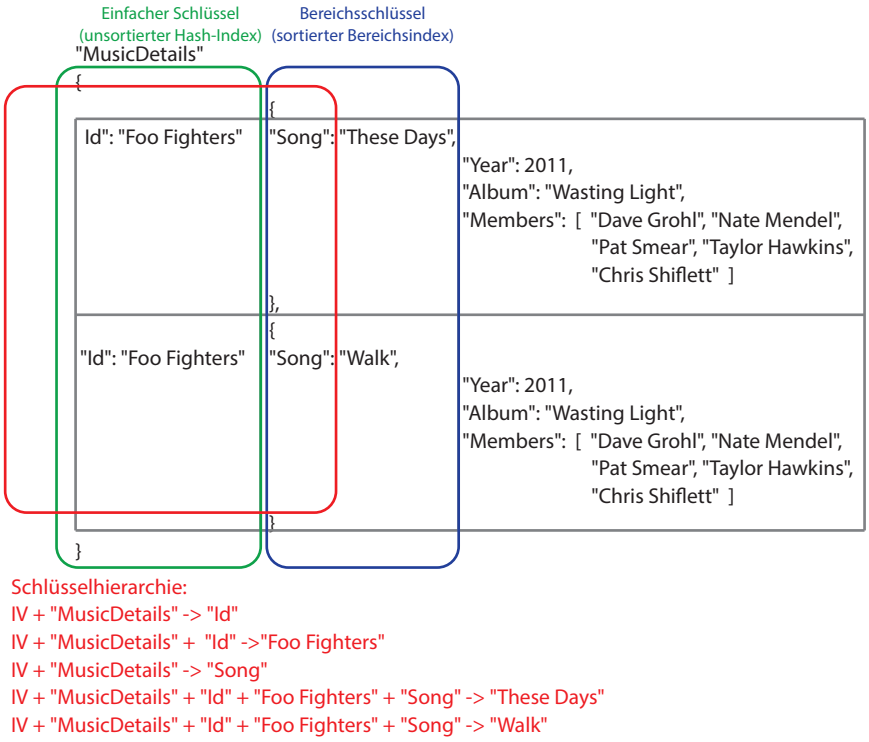

Abbildung 4.7: Schlüsselhierachie, welche der HDSE-Ansatz verwendet, wodurch die `Query`-Operation implizit unterstützt wird.

Diese einheitlichen Verschlüsselungsschlüssel erlauben somit die Realisierung von Gleichheitsabfragen. Auch hier wird, wie bei den Operationen `PutItem` oder `GetItem`, der Primärschlüssel als Basis für die Verschlüsselung verwendet. Es wird nach dem HDSE-Ansatz ein hierarchisch abgeleiteter Verschlüsselungsschlüssel zur Verschlüsselung der

Attribute und deren Werte erstellt. Der Ablauf der Verschlüsselung ist somit der gleiche wie bei den anderen Operationen und bedarf keinerlei weiterer Adaptierungen. Nach erfolgreicher Verschlüsselung kann die verschlüsselte Abfrage an die Datenbank übertragen und serverseitig ausgewertet werden.

Ein entscheidender Vorteil der gleichen Verschlüsselungsschlüssel der Bereichsschlüssel ist, dass ohne weitere Adaptierungen Gleichheitsabfragen auf den Bereichsschlüssel unterstützt werden. Diese Operation erlaubt neben Gleichheitsabfragen auch Bereichsabfragen. Eine mögliche Realisierung dieser Funktionalität wäre durch den Einsatz homomorpher Verschlüsselung (siehe Abschnitt 3.1.3) oder die Implementierung eines Indexstruktur (Baum-Indizes) in den Tabellen der DynamoDB (vergleichbar mit der Index-Idee des Structrue Preserving Database Encryption Scheme aus Abschnitt 3.2.6) möglich. Da der HDSE-Ansatz unabhängig vom eingesetzten Verschlüsselungsalgorithmus (siehe Abschnitt 3.2.7) ist, werden in dieser Arbeit die Bereichsabfragen mithilfe von homomorpher Verschlüsselung realisiert. Dies erlaubt eine serverseitige Durchführung von Bereichsabfragen auf verschlüsselte Daten.

Darüber hinaus erlaubt diese Operation eine zusätzliche Filterung nach Attributen (`QueryFilter`) (siehe Abschnitt 2.6.5). Diese wird auch vom HDSE durch seine zellenbasierte Verschlüsselung nicht unterstützt.

Die Entschlüsselung der abgefragten Daten erfolgt analog zur `BatchGetItem`-Operation.

Scan

Die `Scan`-Operation (siehe Abschnitt 2.6.5) erlaubt das Durchsuchen der Datensätze und ihrer Attribute innerhalb einer Tabelle. Zusätzlich ermöglicht diese Operation Konjunktiv-Abfragen auf Attribute. Dieses Verfahren wird ohne zusätzliche Arbeit (z. B. in Form eines Index) nicht vom Verschlüsselungsansatz HDSE unterstützt, da jeder Wert in der Datenbank unterschiedliche Bitmuster aufweist und Vergleiche somit nicht durchführbar sind.

Der Verschlüsselungsclient realisiert zwei unterschiedliche Index-Implementierungen. Diese beiden Indizes wurden ausgewählt, um unterschiedliche Ansätze zur Realisierung von serverseitigen Bereichsabfragen zu demonstrieren und ihre Laufzeit zu untersuchen. Bei der ersten Index-Implementierungen handelt es sich um einen Index nach dem Verschlüsselungsansatz Bucketing (siehe Abschnitt 3.2.5), dieser erlaubt eine grobe serverseitige Filterung der Daten und eine exakte Filterung der Daten clientseitig.

Bei der zweiten Index-Implementierung handelt es ich um einen invertierten Index nach dem Verschlüsselungsansatz Searchable Symmetric Encryption (siehe Abschnitt 3.2.8). Durch die Verwaltung eines invertierten Index, ist es möglich eine genauere serverseitige Filterung, als beim Bucketing-Index durchführen zu können. Welche dieser Index-Implementierung vom Verschlüsselungsclient eingesetzt wird, wird in der CryptoConfig definiert.

Die Erstellung und Verwaltung eines Index ist sehr aufwendig, daher sollte ein Index nur auf benötige Attribute erstellt werden. Der Client muss daher wissen, welche Attribute für den Index verwendet werden. Da sich der Client bereits die Metadaten pro Tabelle im Data-Dictionary merken muss, wird dieses um die Information, auf welchem

Attribut ein Index verwendet wird, erweitert.

Scan-Implementierung mithilfe eines Bucketing-Index

Der Bucketing-Index erweitert den mit dem HDSE-Verschlüsselten Datensatz in der DynamoDB um Zusatzinformationen pro Attribut, für welche ein Index erstellt werden soll. Diese Information muss bei jeder Schreibeoperation hinzugefügt bzw. aktualisiert werden. Die Zusatzinformation verschlüsselt den eigentlichen Attributwert und erlaubt damit die Realisierung von Bereichsabfragen. Der Attributname dieser Zusatzinformation ist für jeden Datensatz innerhalb einer Tabelle gleich und wird auch mit demselben Verschlüsselungsschlüssel verschlüsselt (spaltenbasierte Verschlüsselung). Dadurch, dass jeder Datensatz innerhalb einer Tabelle dasselbe Attribut (welches mit demselben Schlüssel kodiert ist) besitzt, kann eine Scan-Abfrage auf diese Tabelle formuliert werden. Ein Beispiel für diese Art von Index ist in Abbildung 4.8 ersichtlich. Hier wird ein Index auf das Attribut *year* erstellt. Das Indexattribut ist in diesem Beispiel das Attribut *Bucket_Year*. Der Wert des Indexattributs wird nach der Bucketing-Idee (Partition und Identifkationsfunktion), siehe Abschnitt 3.2.5, erstellt. Dieser Wert wird anschließend mit einem tabellenweit einheitlichen Schlüssel unter Einsatz des OPE-Algorithmus verschlüsselt. Daher wird z. B. das Jahr 2011 in einen Bucket-Wert und anschließend zu dem Wert 797479227844396567436413 umgewandelt.

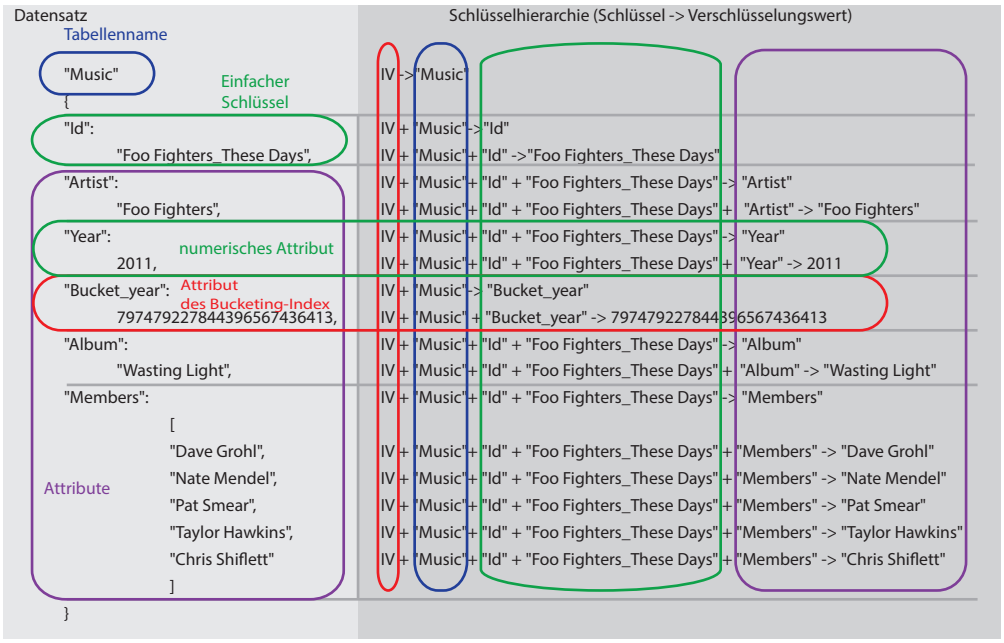

Abbildung 4.8: Änderung des Verschlüsselungsansatzes für den Bucketing-Ansatz. Es wird ein zusätzliches in der Tabelle einheitlich verschlüsseltes Attribut eingeführt.

Durch diese einheitlichen Attribute innerhalb der Tabelle können Scan-Abfragen for-

```
{
    TableName: MusicDetails,
       ScanFilter: {
         Year: {
            AttributeValueList: [
                {  N: 2010 } ,
                {  N: 2012 } ,
         ],
            ComparisonOperator: BETWEEN
        }
    }
}
```

Codebeispiel 4.3: Dieses Beispiel enthält eine beispielhafte Scan-Abfrage mit dem `Between`-Operator.

```
{
    TableName: MusicDetails,
       ScanFilter: {
         Bucket_Year: {
            AttributeValueList: [
                {  N: 79747922784439656743641 3 } ,
                {  N: 80841032786459996245446 9 } ,
         ],
            ComparisonOperator: BETWEEN
        }
    }
}
```

Codebeispiel 4.4: Dieses Beispiel enthält eine beispielhafte Scan-Abfrage nachdem der Verschlüsselungsclient das Query-Rewriting für den Bucketing-Index durchführt.

muliert werden. Hierzu wird das Wissen über das indizierte Attribut (dessen Name und in welcher Art und Weise die Werte kodiert werden) aus dem Data-Dictionary benötigt. Der Client verwendet dieses Wissen, um die Nutzerabfragen auf die indizierten Attribute umzuformulieren (z. B. indem die Nutzerabfrage auf das Attribut *year* vom Client auf das Attribut *Bucket_Year* umformuliert wird). Siehe Codebeispiele 4.3 und 4.4, das erste Beispiel zeigt die Nutzerabfrage und das zweite Beispiel die durch den Verschlüsselungsclient adaptierte Abfrage. Dies erlaubt die Realisierung von serverseitigen Abfragen.

Um auch in dieser Datenbankoperation Bereichsabfragen (wie bei der Query-Operation) zu unterstützen implementiert der Bucketing-Index nur Abfragen auf numerische Werte. Die abgefragten Ergebnissemengen werden schlussendlich vom Verschlüsselungsclient entschlüsselt und gefiltert. Der Verschlüsselungsclient führt bei einer Scan-Methode eine clientseitige Filterung der Werte durch. Diese Filterung ist dann relevant, wenn in ei-

nem Bucket unterschiedliche Wertebereiche enthalten sind. Diese zusätzliche Filterung erlaubt daher, exakte Ergebnisse zu retournieren. Die Scan-Operation erlaubt die Angabe beliebig vieler Bedingungen und somit die Realisierung von Konjunktiv-Abfragen, welche durch diese Filterung ebenfalls unterstützt werden.

Scan-Implementierung mithilfe eines verschlüsselten invertierten-Index

Die zweite Index-Implementierung ist der verschlüsselte invertierte Index. Hierzu wird pro Attribut (für welches ein Index realisiert werden soll) eine eigene Datentabelle erstellt, welche den Index beinhaltet. Diese Index-Tabellen pro zu indizierendem Attribut werden vom Verschlüsselungsclient im Zuge der Erstellung der Primärdatentabellen erstellt. Die Index-Tabellen verwenden einen zusammengesetzten Schlüssel als Primärschlüssel, wobei der einfache Schlüssel der zu indizierende Attributwert und der Bereichsschlüssel der Schlüssel vom referenzierte Datensatz ist. Durch diese Struktur kann es auch nicht zur mehrfachen Abspeicherungen von Referenzen kommen, da die DynamoDB keine Duplikate innerhalb des Bereichsschlüssels zulässt. Zusätzlich werden hierdurch DynamoDB-spezifische Limits bei der Länge von Datensätzen umgangen. Dies würde Probleme innerhalb des Index erzeugen, wenn zahlreiche Datensätze für einen Wert referenziert werden und jede Referenz als Attribut abgespeichert werden würde. Ein Beispiel für solche Index-Tabellen ist in Abbildung 4.9 ersichtlich. Es werden Index Einträge für die Attribute *Arist*, *Year* und *Members* erstellt. Diese referenzieren alle denselben Datensatz *Foo Fighters_These Days*.

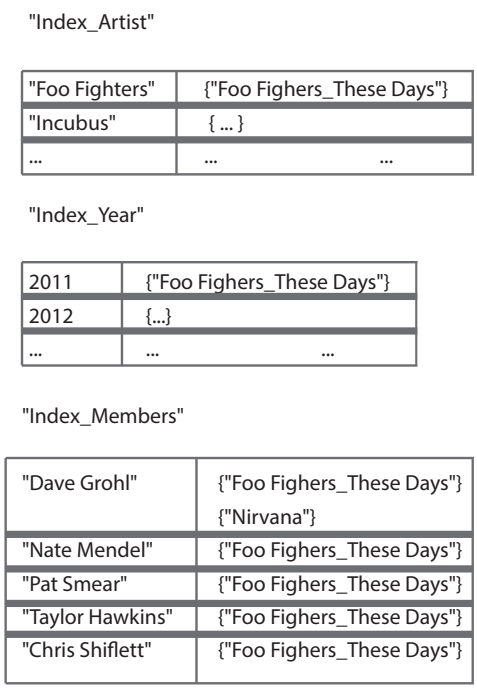

Abbildung 4.9: Abbildung zeigt die Idee des verschlüsselten invertierten Index, dargestellt durch drei DynamoDB-Tabellen für einen Datensatz.

Dieser Ansatz erlaubt es, neben den Basis-Datentypen der DynamoDB (Binär, Zeichenketten und Numerisch), auch Werte aus Mengen innerhalb des Index abzubilden. Dies erlaubt die Realisierung von Abfragen auf Mengen. Die Datensätze der Index-Tabellen werden nach dem HDSE-Verschlüsselungsansatz verschlüsselt.

Die Verwaltung und Aktualisierung der Indextabellen wird vom Verschlüsselungsclient bei jeder Schreibeoperation automatisch durchgeführt. Wird ein neuer Datensatz geschrieben, welcher zu indizierende Attribute besitzt, erstellt der Verschlüsselungsclient für jene Attribute eigene Schreibeoperationen für die jeweilige Index-Tabelle. Dies resultiert pro indiziertem Attribut in einer weiteren Schreibeoperation pro „Nutzeroperation“. Dementsprechend sind Schreibeoperationen umso teurer je mehr indizierte Attribute eine Tabelle hat. Kommt es zu einem Ausfall oder Fehlerfall des Clients, können noch nicht geschriebene Attributwerte nicht mehr in die jeweilige Index-Tabelle eingetragen werden. Um diese Probleme zu lösen, merkt sich der Client lokal die zu schreibenden Indexdaten so lange, bis das letzte Attribut in den Index-Tabellen übertragen worden ist. Wird der Client nach einem Fehlerfall wieder gestartet, wird der zuletzt zu schreiben versuchte Datensatz erneut in die jeweiligen Tabellen übertragen.

Wird nun eine Scan-Abfrage vom Nutzer formuliert, wird diese vom Verschlüsselungsclient in mehrere Scan-Abfragen pro abzufragendes Attribut für die jeweilige Index-Tabelle erstellt. Die einfache Realisierung von Abfragen wird durch die Speicherung der indizierten Werte im einfachen Schlüsselattributbestandteil ermöglicht. Alle Datenwerte

```
{
    TableName: Index_Year,
       ScanFilter: {
         AttributWert: {
            AttributeValueList: [
                {  N: 2010 } ,
                {  N: 2012 } ,
         ],
            ComparisonOperator: BETWEEN
        }
    }
}
```

Codebeispiel 4.5: Dieses Beispiel enthält eine beispielhafte Scan-Abfrage auf die Index-Tabelle Year.

innerhalb des einfachen Primärschlüsselattributs werden mit dem gleichen Verschlüsselungsschlüssel kodiert (spaltenbasierte Verschlüsselung). Wird z. B. eine Scan-Operation nach dem Attribut *Year* erstellt (siehe Scan-Beispielabfrage aus Codebeispiel 4.3), generiert der Client eine Scan-Abfrage nach dem gesuchtem Wert in der Index-Tabelle *Index_Year*. Eine solche Abfrage ist in Codebeispiel 4.5 ersichtlich.

Der retournierte Primärschlüsselwert wird vom Client verwendet, um den benötigten Datensatz aus der Primärdatentabelle zu laden und anschließend zu entschlüsseln. Werden komplexere Abfragen erstellt, z. B. Konjunktiv-Abfragen, filtert der Client lokal die geladenen Ergebnismengen auf Basis der Nutzerabfrage.

4.2 Architektur

Auf Basis des zuvor beschriebenen Konzepts wird in diesem Abschnitt nun die Architektur des Verschlüsselungsclients präsentiert. Ein grafischer Überblick ist in Abbildung 4.10 ersichtlich. Der Verschlüsselungsclient, welcher mit der Amazon DynamoDB kommuniziert, besteht aus folgenden Komponenten:

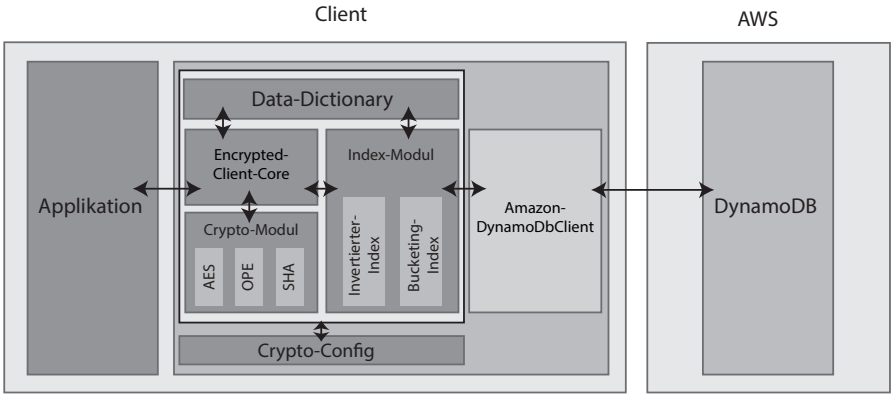

Abbildung 4.10: Überblick über die Kommunikation der Komponenten des Verschlüsselungsclient und der Interaktion mit Applikation und Datenbank.

Data-Dictionary – Das Data-Dictionary beinhaltet Informationen über die gespeicherten Datentypen, Datenstrukturen der Datensätze und ob synchrone oder asynchrone Abfragen auf die Attribute durchgeführt werden sollen[1]. Das Data-Dictionary wird vorrangig zum Entschlüsseln der Daten und zur Index-Erstellung verwendet.

Crypto-Config - Die Crypto-Config beinhaltet den privaten IV, welcher vom Verschlüsselungsansatz HDSE benötigt wird. Zusätzlich werden etwaige Konfigurationen (Verschlüsselungsalgorithmus, Index-Implementierung etc.) für die Verschlüsslungen in dieser Konfiguration gespeichert. Diese Konfigurationsdatei wird von allen anderen Modulen verwendet.

Encrypted-Client-Core - Dieses Modul transformiert jede Nutzeroperationen (Abfragen oder Datenbankoperationen wie Schreibeoperationen) in eine verschlüsselte Form (Query-Rewriting). Die Verschlüsselung erfolgt auf Basis des zellenbasierten Verschlüsselungsansatzes HDSE. Dieses Modul führt zusätzlich ggf. etwaige Filterungen und arithmetische Aktionen wie Additionen auf die Daten (z. B. durch eine UpdateItem-Operation) durch. Um die Daten in ihre richtigen Datentypen zu entschlüsseln, verwendet dieses Modul die gespeicherten Informationen des Data-Dictionarys.

Crypto-Modul – Dieses Modul beinhaltet die eingesetzten Verschlüsselungsalgorithmen des Verschlüsselungsclients. Dies sind unter anderem der symmetrische Blockverschlüsselungsalgorithmus AES und DES (vgl. Abschnitt 3.1.1), der homomorphe Verschlüsselungsalgorithmus Order Preserving Encryption (OPE) (vgl. Abschnitt 3.1.3) und eine kryptographische Hash-Funktion der SHA-256 und SHA-512(vgl. Abschnitt 3.1.4). Der Encrypted-Client-Core verwendet dieses Modul, um die Verschlüsselung der Daten bzw. das Hashen der Verschlüsselungsschlüssel durchzuführen. Es erlaubt den Austausch der Algorithmen, da die Kommunikation über eine definierte Schnittstelle erfolgt.

[1]Die Definition, welche Art von Abfragen auf das Attribut durchgeführt werden kann, wird vom invertieren Index pro Attribut eingesetzt.

Index-Modul – Dieses Modul erlaubt die Realisierung der *Scan*-Abfrage (siehe Abschnitt 2.6.5), welche durch HDSE nicht unterstützt wird. Dadurch werden Abfragen und Konjunktiv-Abfragen auf Attribute ermöglicht. Hierzu wird mithilfe dieses Moduls ein verschlüsselter Index geschaffen. Die Erstellung und die Wartung des verschlüsselten Index muss vom Verschlüsselungsclient durchgeführt werden, weshalb dieses Modul bei jeder durchgeführten Schreiboperation ausgeführt werden muss. Für welche Attribute ein Index erstellt wird, wird im Data-Dictionary definiert. Daher wird das Data-Dictonary ebenfalls vom Index-Modul verwendet. Das Index-Modul besitzt zwei unterschiedliche Index-Implementierungen: einen Bucketing-Index (siehe Abschnitt 3.2.5) und einen invertierten-Index nach dem Verschlüsselungsansatz Searchable Symmetric Encryption (siehe Abschnitt 3.2.8).

Das Zusammenspiel der beschriebenen Module (siehe Abbildung 4.10) ist wie folgt: Führt die Applikation eine Schreiboperation (z.B. ein `PutItem`) durch, wird diese vom Encrypted-Client-Core entgegengenommen. Dieser verschlüsselt mithilfe der Informationen des Data-Dictionarys und der Algorithmen aus dem Crypto-Modul die Klartexte der `PutItem`-Operation in eine nach dem HDSE-Ansatz verschlüsselte Form. Die Informationen, welcher Verschlüsselungsalgorithmus eingesetzt und welcher IV verwendet wird, werden aus der Crypto-Config vom Crypto-Modul und dem Encrypted-Client-Core entnommen. Bevor die Daten dem AmazonDynamoDBClient übergeben werden, welcher die Übertragung zur Datenbank durchführt, wird das Index-Modul durchlaufen. Dieses Modul ist verantwortlich für die Aktualisierung der verschlüsselten Indizes, um eine Scan-Operation und die damit verbundene Abfrage durchführen zu können. Nachdem die benötigten Operationen vom Index-Modul durchgeführt wurden, wird die kodierte `PutItem`-Operation dem AmazonDynamoDBClient übergeben, der die Daten an die DynamoDB überträgt.

Im Vergleich hierzu werden die Leseoperationen wie `BatchGetItem, GetItem` oder `Query` einfach vom Encrypted-Client-Core entgegengenommen, die Daten basierend auf dem HDSE-Ansatz kodiert und durch das Index-Modul geschleust. Dieses Modul führt keinerlei Anpassungen an den Abfragen durch, sondern überträgt diese sofort an die Datenbank.

Bei der Scan-Operation werden die Daten ebenfalls vom Encrypted-Client-Core verarbeitet, jedoch wird, abhängig von der verwendeten Index-Implementierung, in unterschiedliche Abfragen umgewandelt. Beim Bucketing-Index werden zum Beispiel die abgefragten Attribute auf die erstellten Zusatzattribute umformuliert (und jene Attribute dementsprechend verschlüsselt). Beim invertierten Index werden die Index-Tabellen abgefragt und die referenzierten Datensätze aus ihren Tabellen geladen.

4.3 Zusammenfassung

Zusammenfassend geht die hier beschriebene Architektur auf die am Anfang dieses Kapitels beschriebenen Anforderungen wie folgt ein:

- Der Server wird als nicht vertrauensvoll erachtet, daher realisiert der Verschlüsselungsclient eine clientseitige Verschlüsselung. Weder Daten noch ihre Metada-

ten werden im Klartext in der Datenbank gespeichert. Dies ist durch das Modul *Encrypted-Client-Core* und das *Crypto-Modul* realisiert.

- Die Verschlüsselung ist transparent für den Nutzer, der Verschlüsselungsclient implementiert den Großteil der Funktionalitäten der Amazon DynamoDB. Es wird der Großteil aller Abfrage- und Datenbankoperationen zur Verfügung gestellt. Dies ist durch das Modul *Encrypted-Client-Core* und das *Index-Modul* gegeben. Das Index-Modul erlaubt auch die Unterstützung der Scan-Abfrage, die durch den zellenbasierten HDSE nicht unterstützt wird. Die Abfragefunktionalität im Hinblick auf Bereichsabfragen und Konjunktiv-Abfragen ist ebenfalls durch den Verschlüsselungsclient unterstützt. Bereichsabfragen können mithilfe des OPE-Algorithmus (aus dem Crypto-Modul) serverseitig durchgeführt werden. Konjunktiv-Abfragen werden durch clientseitige Post-Filterungen im Encrypted-Client-Core-Modul realisiert.
- Der HDSE ist nicht abhängig von einem bestimmten Verschlüsselungsalgorithmus, daher implementiert der Verschlüsselungsclient eine Schnittstelle, welche den Austausch der Algorithmen ermöglicht. Nichtsdestotrotz wird für die Realisierung von Bereichsabfragen eine konkrete OPE-Implementierung benötigt [23], wodurch der Verschlüsselungsclient von dieser abhängig ist.

In den folgenden Tabellen ist eine Zusammenfassung der unterstützten Datenbankoperationen (siehe Tabelle 4.1) und Datenbankabfragen (siehe Tabelle 4.2) ersichtlich. Die Datenbankoperationen können für alle Datentypen (skalar und Mengen) durchgeführt werden. Der Großteil der Datenbankabfragen (Query und Scan) sind nur für numerische Werte verfügbar. Die nicht realisierten Aktionen (siehe nachfolgende Tabellen) wurden im Zuge dieser Arbeit nicht implementiert, sind jedoch nicht durch den Verschlüsselungsansatz selbst beschränkt.

Datenbankoperationen		
PutItem	Schreiben	Ja (Alle Datentypen)
	Schreiben auf Basis von Bedingungen	Ja (Alle Datentypen)
BatchWriteItem	Schreiben	Ja (Alle Datentypen)
	Löschen	Ja (Alle Datentypen)
	Aktion `UnprocessedItems` (siehe Abschnitt 2.6.4)	Nicht realisiert
UpdateItem	Schreiben	Ja (Alle Datentypen)
	Schreiben auf Basis von Bedingungen	Ja (Alle Datentypen)
	Aktion `ADD`	Ja (Alle Datentypen)
	Aktion `PUT`	Ja (Alle Datentypen)
	Aktion `DELETE`	Ja (Alle Datentypen)
DeleteItem	Löschen	Ja (Alle Datentypen)
	Löschen auf Basis von Bedingungen	Ja (Alle Datentypen)

Tabelle 4.1: Die Tabelle zeigt eine Übersicht die unterstützten Datenbankoperationen des Verschlüsselungsclients.

Datenbankabfragen		
GetItem	Lesen	Ja (Alle Datentypen)
	Lesen bestimmter Attribute	Ja (Alle Datentypen)
BatchGetItem	Lesen	Ja (Alle Datentypen)
	Lesen bestimmter Attribute	Nein (Alle Datentypen)
	Aktion `UnprocessedItems` (siehe Abschnitt 2.6.5)	Nicht realisiert
	Aktion `ExclusiveStartKey` (siehe Abschnitt 2.6.5)	Nicht realisiert
Query	Operator `EQ`	Ja (Nur numerischer Datentyp)
	Operator `LE`	Ja (Nur numerischer Datentyp)
	Operator `LT`	Ja (Nur numerischer Datentyp)
	Operator `GE`	Ja (Nur numerischer Datentyp)
	Operator `GT`	Ja (Nur numerischer Datentyp)
	Operator `BETWEEN`	Ja (Nur numerischer Datentyp)
	Operator `BEGINS_WITH`	Nein (Alle Datentypen)
	Filterfunktion auf nicht Schlüsselattribute	Nein (Alle Datentypen)
	Lesen bestimmter Attribute	Nein (Alle Datentypen)
	Aktion `ExclusiveStartKey` (siehe Abschnitt 2.6.5)	Nicht realisiert
	Abfragen auf Sekundärindizes	Nein (Alle Datentypen)
Scan (Bucketing-Index)	Operator `EQ`	Ja (Nur numerischer Datentyp)
	Operator `LE`	Ja (Nur numerischer Datentyp)
	Operator `LT`	Ja (Nur numerischer Datentyp)
	Operator `GE`	Ja (Nur numerischer Datentyp)
	Operator `GT`	Ja (Nur numerischer Datentyp)
	Operator `BETWEEN`	Ja (Nur numerischer Datentyp)
	Operator `BEGINS_WITH`	Nein (Alle Datentypen)
	Operator `NOT_NULL`	Nein (Alle Datentypen)
	Operator `NULL`	Nein (Alle Datentypen)
	Operator `NOT_CONTAINS`	Nein (Alle Datentypen)
	Operator `CONTAINS`	Nein (Alle Datentypen)
	Konjunktiv-Abfragen (AND) mit unterstützten Operatoren	Ja
	Disjunktive-Abfragen (OR)	Nein
	Aktion `ExclusiveStartKey` (siehe Abschnitt 2.6.5)	Nicht realisiert
	Aktion `LastEvaluatedKey` (siehe Abschnitt 2.6.5)	Nicht realisiert
Scan (Invertierter-Index)	Operator `EQ`	Ja (Nur numerischer Datentyp)
	Operator `LE`	Ja (Nur numerischer Datentyp)
	Operator `LT`	Ja (Nur numerischer Datentyp)
	Operator `GE`	Ja (Nur numerischer Datentyp)
	Operator `GT`	Ja (Nur numerischer Datentyp)
	Operator `BETWEEN`	Ja (Nur numerischer Datentyp)
	Operator `BEGINS_WITH`	Nein (Alle Datentypen)
	Operator `NOT_NULL`	Nein (Alle Datentypen)
	Operator `NULL`	Nein (Alle Datentypen)
	Operator `NOT_CONTAINS`	Nein (Alle Datentypen)
	Operator `CONTAINS`	Ja (Alle Mengendatentypen)
	Konjunktiv-Abfragen (AND) gleiche Operatoren wie Bucketing-Index	Ja
	Disjunktive-Abfragen (OR)	Nein
	Aktion `ExclusiveStartKey` (siehe Abschnitt 2.6.5) Aktion `LastEvaluatedKey` (siehe Abschnitt 2.6.5)	Nicht realisiert Nicht realisiert

Tabelle 4.2: Die Tabelle zeigt eine Übersicht die unterstützten Datenbankabfragen des Verschlüsselungsclients.

Kapitel 5

Implementierung des Verschlüsselungsclients

Basierend auf der zuvor beschriebenen Architektur des Verschlüsselungsclients wird in diesem Abschnitt auf die Implementierungsdetails der Module eingegangen. In einem ersten Schritt werden die benötigten Konfigurationen, das Data-Dictionary und die Crypto-Config erörtert. Anschließend wird im Abschnitt Encrypted-Client-Core gezeigt, wie der Verschlüsselungsansatz HDSE auf das Datenmodell der DynamoDB angewandt und die Realisierung der Datenbankoperationen umgesetzt wird. Abschließend werden noch das Crypto-Modul sowie das Index-Modul im Detail erörtert. Im Abschnitt Index-Modul folgen die Erörterungen der Implementierung vom Bucketing- und invertierten Index.

5.1 Data-Dictionary

Das Data-Dictionary ist eine Konfigurationsdatei des Verschlüsselungsclients und beinhaltet Informationen über die gespeicherten Datentypen, Datenstrukturen der Datensätze und darüber, wie die Daten in den Tabellen abgefragt werden können. Es wird vorrangig zum Entschlüsseln der Daten und zur Index-Erstellung verwendet. Diese Informationen werden in einem XML-Format abgespeichert. In Codebeispiel 5.1 ist eine mögliche Konfiguration ersichtlich.

```
1  <EncryptedDynamoDb>
2  <Tables>
3  <Table name="Music">
4      <attribute name="Id" keyType="HASH" type="S" queryable="true" updateBehavior="sync" />
5      <attribute name="Artist" type="S" queryable="true" updateBehavior="sync" />
6      <attribute name="Year" type="N" queryable="true" updateBehavior="sync" />
7      <attribute name="Album" type="S" queryable="true" updateBehavior="sync" />
8      <attribute name="Members" type="SS" queryable="true" updateBehavior="sync" />
9    </Table>
10 </Tables>
11 </EncryptedDynamoDb>
```

Codebeispiel 5.1: Eine Beispieltabelle innerhalb des *Data-Dictionary*.

Dieses Beispiel zeigt die Informationen für die Tabelle *Music*. Diese besitzt Datensätze mit je fünf Attributen, wobei das Attribut namens *Id* der Primärschlüssel mit einem einfachen Schlüsselschema (*keyType=HASH*) ist. Bei einer Tabelle, welche als Primärschlüssel einen zusammengesetzten Schlüssel verwendet, wird das Bereichsattribut mit dem *keyType=RANGE* markiert. Des Weiteren wird pro Attribut der Datentyp (*type*) angegeben. Hier wird die Datentyp-Notation der DynamoDB wiederverwendet (siehe Tabelle 2.3 in Abschnitt 2.6.1). In diesem Beispiel ist das Attribut *Artist* vom Datentyp Zeichenkette (String). Der Bezeichner *queryable* gibt an, dass für dieses Attribut ein Index erstellt werden soll, welcher, wie bereits erörtert, für Scan-Operationen benötigt wird. Dieses Attribut erlaubt nur die Boolean-Werte *true* oder *false*. Das letzte Attribut *updateBehavior* wird nur beim invertierten verschlüsselten Index verwendet. Da hier Datensätze auf diversen Index-Tabellen geschrieben werden, ist es möglich, diese Schreiboperationen auch asynchron durchzuführen (Werte *sync* oder *async*), im Normalfall wird jedoch immer eine synchrone Schreiboperation durchgeführt.

Das Wissen über den Primärschlüssel und die Datentypen erlaubt dem Verschlüsselungsclient die Daten dementsprechend zu entschlüsseln. Um die Anzahl an Ver- und Entschlüsselungen zu reduzieren, wird während der Ausführungszeit gleichzeitig eine verschlüsselte und unverschlüsselte Variante des Data-Dictionarys im flüchtigen Speicher gehalten. Dies ist besonders für die Primärschlüssel relevant, da der Attributname des Primärschlüssels in verschlüsselter Form immer dieselbe kodierte Form annimmt. Dies erlaubt es, dass man bei einem verschlüsselten Datenbankergebnis den Primärschlüssel, welcher zur Entschlüsselung des Datensatzes benötigt wird, direkt identifizieren kann.

5.2 Crypto-Config

Neben der Konfiguration der Daten im Data-Dictionary erlaubt die Crypto-Config die Konfiguration der Verschlüsselungsdetails, wie z.B. den eingesetzten Verschlüsselungsalgorithmus, in Form eines XML-Dokuments. Eine mögliche Konfiguration ist in Beispiel 5.2 ersichtlich.

```
1 <cryptoData>
2   <algorithm length="128" name="AES" padding="PKCS5Padding"
3     operationMode="CBC" hashProvider="SHA256" />
4   <iv>ZFfd5+hkCsuQfp13jDEnJQ==</iv>
5   <keyGeneration  useMode="Chaining" />
6   <scanOperation useMode="NoScanOperationPossible">
7   </scanOperation>
8   <opeEncryption plaintextSizes="64" ciphertextSizes="128"/>
9 </cryptoData>
```

Codebeispiel 5.2: Die Crypto-Config des Verschlüsselungsclient.

Dieses Beispiel beinhaltet die Konfigurationsmöglichkeiten. Jede Konfigurationsmöglichkeit der gezeigten XML-Elemente wird in den nachfolgenden Abschnitten im Detail erörtert.

5.2.1 Konfiguration der Blockverschlüsselung

Das Element *algorithm* aus Beispiel 5.2 erlaubt die Angabe der zu verwendenden Blockverschlüsselung wie z. B. den AES und DES (siehe Abschnitt 3.1.1). Diese Entwicklungsbibliothek verlangt bei der Anwendung einer Verschlüsselung, dass der Nutzer neben der Bit-Länge (Attribut *length*) der Verschlüsselungsblöcke (siehe Abschnitt 3.1.1) auch das Padding (Attribut *padding*) und den Mode of Operation (Attribut *operationMode*) angibt. Gültige Werte für den AES sind beispielsweise bei der Schlüssellänge 128 Bit, für das Padding PKCS5Padding und der Mode of Operation CBC. Zusätzlich können in diesem Element auch die kryptographischen Hashfunktionen (Attribut *hashProvider*) definiert werden. Die Hashfunktionen werden zur Erstellung der Verschlüsselungsschlüssel eingesetzt. Der Verschlüsselungsclient unterstützt die Hashfunktionen SHA256 und SHA512.

5.2.2 Konfiguration des Initialisierungsvektors

Der Initialisierungsvektor (Element *iv* in Beispiel 5.2) wird vom Verschlüsselungsclient beim ersten Aufruf erstellt und in Crypto-Konfigurationsdatei gespeichert. Die Erstellung erfolgt über die `SecureRandom`-Klasse, welche Java zur Verfügung stellt. Die Länge ist abhängig von der Blocklänge des eingesetzten Verschlüsselungsalgorithmus. Generell verwendet der Verschlüsselungsclient AES-128 mit PKCS-Padding. Somit wird standardmäßig ein IV mit einer Größe von 128 Bit erstellt.

5.2.3 Konfiguration der Schlüsselerstellung

Die Konfiguration der Schlüsselerstellung wird durch das Element *keyGeneration* in Beispiel 5.2 ermöglicht. Der hier definierte Wert gibt an, wie die Teilschlüssel des Verschlüsselungsschlüssels miteinander verkettet werden, um in einen Schlüssel für einen Datenwert zu resultieren. Der Verschlüsselungsclient implementiert zwei unterschiedliche Strategien:

- Die erste Verkettungsart *CBCChaining* ist angelehnt an die Idee des *Cipher block Chaining-Mode of Operation* (vlg. Abschnitt 3.1.1), indem der erste Teilschlüssel der Schlüsselhierarchie mit dem Initialisierungsvektor durch eine XOR-Operation verbunden wird. Dies erlaubt die Einbindung des IV in den resultierenden Verschlüsselungsschlüssel. Anschließend werden alle weiteren Teilschlüssel mit diesem Ergebnis ebenfalls durch diese Operation verknüpft. Bevor es zu einer Verkettung der Teilschlüssel (z. B. Tabellenname oder Attributname) kommt, werden sie mithilfe der zuvor angegebenen kryptographischen Hashfunktion (z. B. SHA-256) in eine einheitliche binäre Form und Größe gebracht.
- Die zweite Verkettungsart *Chaining* ist einfacher. Die Teilschlüssel werden mithilfe der Java-Klasse `StringBuilder` durch eine Zeichenketten-Konkatenation verbunden. Anschließend wird diese neue Zeichenkette mithilfe der zuvor angegebenen kryptographischen Hashfunktion in eine einheitliche Größe gebracht.

Bei den beiden Ansätzen fließen alle Schlüsselbestandteile vollständig in die Generierung des Verschlüsselungsschlüssels ein. Dies erfolgt entweder durch die XOR-Operation oder durch den Einsatz der kryptographischen Hashfunktion auf den gesamten Schlüssel.

5.2.4 Konfiguration von Scan-Operationen

Scan-Operationen werden über das Element *scanOperation* aus Beispiel 5.2 konfiguriert. Wie bereits erörtert, implementiert der Verschlüsselungsclient zwei unterschiedliche Index-Lösungen. In dieser Konfigurationsdatei kann somit entweder der Bucketing-Index (*PartitionAndIdentification*), der verschlüsselte invertierte Index (*EncryptedIndex*) oder gar kein Index (*NoScanOperationPossible*) konfiguriert werden. Da eine Index-Erstellung mit erhöhtem Aufwand verbunden ist, kann das Index-Modul somit auch vollkommen deaktiviert werden.

Wird vom Nutzer der Bucketing-Index gewählt, muss dieser die Histogramm-Funktion des Bucketing-Index definieren. Der Bucketing-Index erstellt auf Basis einer Histogramm-Funktion (Element *HistogramConstructionTechnique*) jene Zusatzinformation, über die Abfragen durchgeführt werden können. Der Verschlüsselungsclient unterstützt die Technik *EquiWith*. Bei dieser Methode muss der Wertebereich (Element *DomainRange* mit den Attributen *min* und *max* für Minimal- Maximal-Werte) sowie die Anzahl der Buckets definiert werden (*DataAmount*). Diese Werte werden am Beginn der Index-Erstellung definiert. Eine mögliche Konfiguration ist in Beispiel 5.3 ersichtlich.

```
1 <cryptoData>
2 ...
3   <scanOperation useMode="PartitionAndIdentification" >
4     <HistogramConstructionTechnique use="EquiWidth">
5       <DomainRange min="1922" max="2015" />
6       <DataAmount value="1000"/>
7     </HistogramConstructionTechnique>
8   </scanOperation>
9 ...
10 </cryptoData>
```

Codebeispiel 5.3: Dieser Ausschnitt der Konfigurationsdatei stellt die Erweiterung des Beispiels 5.2 dar. Es wird dabei mithilfe des Modus `PartitionAndIdentification` die Bucketing-Funktionalität aktiviert. Des Weiteren werden die Parameter für die *Equi-Width*-Funktion definiert.

5.2.5 Konfiguration der Order-Preserving-Encryption

Durch das Element *opeEncryption* ist es möglich, Order-Preserving-Encryption zu konfigurieren. OPE wird dazu eingesetzt, um Ordnungsrelationen zwischen den Klartexten bei der Verschlüsselung bzw. durch die Verschlüsselung hindurch zu wahren. Zusätzlich muss in der Crypto-Config für OPE die Klartext und Schlüsseltextlänge (Attribute *plaintextSizes* und *ciphertextSizes*) angegeben werden (siehe Beispiel 5.2). In dieser Arbeit wird OPE standardmäßig mit einem Klartextwertebereich von 64-Bit und einem Schlüsseltextwertebereich von 128-Bit verwendet.

5.3 Encrypted-Client-Core

Der *Encypted-Client-Core* ist der Kern des Verschlüsselungsclients. Dieses Modul transformiert jede vom Benutzer abgesetzte Datenbankoperation in ihre verschlüsselte Form. Es wird somit ein Query-Rewriting durchgeführt. Die Verschlüsselung der Daten erfolgt, wie bereits erörtert, auf Basis des zellenbasierten Verschlüsselungsansatzes HDSE. Jede Datenbankoperation, ob Abfrage-, Schreibe- oder auch Schemaoperation[1], wird in diesem Modul separat behandelt. Dies ermöglicht es, auf die jeweiligen Funktionalitäten (wie Filterungen) der Nutzeroperation gesondert einzugehen. Nichtsdestotrotz werden alle Daten, die übertragen werden, auf die gleiche Art und Weise ver- und entschlüsselt. In den nachfolgenden Codebeispielen wird die Verschlüsselung (siehe Beispiel 5.4) und die Entschlüsselung (siehe Beispiel 5.5) der Daten in Pseudocode gezeigt.

```
1  DynamoDBTable plainTableName = getTableName();
2  IV iv = getIV();
3  DynamoDBData plainRecord = getCurrentDataRecord();
4  DynmoDBKeys plainRecordKeys = extractKeys(plainRecord);
5
6  Key metaKey = chainKeys(iv, plainTableName, plainRecordKeys);
7  DynamoDBData encRecord = new DynamoDBData();
8
9  for(Attribut plainAttribut : plainRecord){
10   String encAttributName = encryptMetaData(metaKey, plainAttribut. AttributName);
11   Key valueKey = chainKeys(iv, plainTableName, plainRecordKeys, plainAttribut.AttributName);
12   AttributeValue encAttributeValue = encryptAttributeValue(valueKey,
       plainAttribut.AttributeValue);
13   Attribute encAttribut = new Attribute(encAttributName, encAttributeValue);
14   encRecord.add(encAttribut);
15 }
16 return encRecord;
```

Codebeispiel 5.4: Dieser Pseudocode zeigt den Ablauf der Verschlüsselung eines Datensatzes basierend auf dem DynamoDB-Datenmodell.

```
1  DynamoDBTable plainTableName = getTableName();
2  IV iv = getIV();
3  DynamoDBData encRecord = getCurrentDataRecord();
4  DynmoDBKeys plainRecordKeys = extractKeys(encData);
5
6  Key metaKey = chainKeys(iv, plainTableName, plainRecordKeys);
7  DynamoDBData plainRecord = new DynamoDBData();
8
9  for(Attribut encAttribut : encRecord){
10   String plainAttributName = decryptMetaData(metaKey, encAttribut.AttributName);
11   Key valueKey = chainKeys(iv, plainTableName, plainDataKeys, plainAttributName);
12   AttributeValue plainAttributeValue = decryptAttributeValue(valueKey,
       encAttribut.AttributeValue);
13   Attribute plainAttribut = new Attribute(plainAttributName, plainAttributeValue);
14   plainData.add(plainAttribut);
15 }
16 return plainData;
```

Codebeispiel 5.5: Dieser Pseudocode zeigt den Ablauf der Entschlüsselung eines Datensatzes basierend auf den DynamoDB-Datenmodell.

[1] Der Verschlüsselungsclient implementiert auch Datenbankschemaoperationen wie `CreateTable` und `UpdateTable`, auf diese wird jedoch nicht weiters eingegangen.

Zur weiteren Verdeutlichung des Verschlüsselungsprozesses wird dieser Vorgang in nachfolgender Abbildung 5.1 anhand eines Beispieldatensatzes gezeigt.

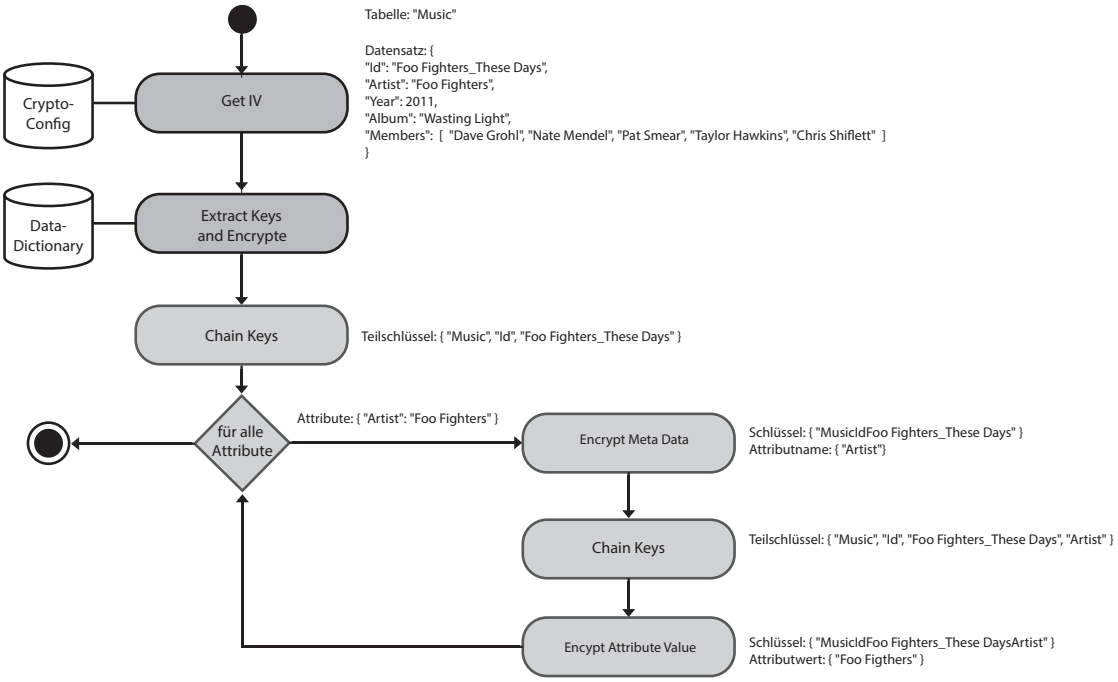

Abbildung 5.1: Ablauf der Verschlüsselung eines Datensatzes.

Der zu kodierende Datensatz stammt aus der Tabelle *Music* und hat den Primärschlüssel *Id* mit dem Wert *Foo Fighters_ These Days* sowie diverse andere Attribute. Basierend auf dem präsentierten Verschlüsselungsprozess wird zuerst der Primärschlüssel aus dem Datensatz extrahiert und entschlüsselt. Hierzu werden das Data-Dictionary und die Crypto-Konfiguration konsultiert. Nach diesem Schritt verfügt der Verschlüsselungsclient über die Informationen für die ersten Teilschlüssel der Verschlüsselungshierarchie, den Tabellennamen *Music*, den Attributnamen des Primärschlüssels *Id* und den Wert des Primärschlüssels *Foo Fighters_ These Days*. Diese Teilschlüssel bilden des Weiteren die Grundlage für den Verschlüsselungsschlüssel aller Attributnamen dieses Datensatzes. Abschließend erfolgt die Verkettung der Teilschlüssel zum finalen Verschlüsselungsschlüssel. Welche Art von Verkettung durchgeführt wird, ist in der CryptoConfig definiert, siehe Abschnitt 5.2.3. In jedem Fall wird der IV in die Verkettung der Primärschlüssel mit eingebunden. Anschließend werden für alle weiteren nicht-Primärschlüssel-Attribute folgende Schritte durchgeführt:

- Verschlüsselung der Attributnamen. Die DynamoDB erlaubt für Attributnamen nur den Datentyp Zeichenkette. Daher wird der Attributname in einer Base64-kodierten Variante abgespeichert. Der eingesetzte Verschlüsselungsalgorithmus wurde in der CryptoConfig (siehe Abschnitt 5.2.1) festgelegt.

- Erstellung eines neuen Verschlüsselungsschlüssels (durch Verkettung) basierend auf dem IV, Tabellennamen, Primärschlüssel (dem Attributnamen und Wert des Primärschlüssels) und dem jeweiligen Attributnamen.
- Verschlüsselung des Attributwerts mit dem in der CryptoConfig (siehe Abschnitt 5.2.1) festgelegten Verschlüsselungsalgorithmus. Ist der zu verschlüsselnde Attributwert ein numerisches Schlüsselattribut, wird dieser Wert mit OPE verschlüsselt. Die Verschlüsselung der numerischen Schlüsselattribute[2] mit OPE erlaubt dem Verschlüsselungsclient die Realisierung von Bereichsabfragen auf Schlüsselattribute.
- Erstellung eines neuen Attributs mit den verschlüsselten Werten. Dies resultiert in eine verschlüsselte Version des übergebenen Datensatzes.

Analog zur Verschlüsselung erfolgt die Entschlüsselung des Datensatzes in Abbildung 5.2. Beim Entschlüsselungsprozess muss jedoch immer berücksichtigt werden, dass zuerst die Teilschlüssel der Verschlüsselungsschlüssel entschlüsselt werden müssen, um einen validen Verschlüsselungsschlüssel zu ergeben. Es muss zuerst der Attributname entschlüsselt werden und anschließend der Wert. Die Entschlüsselung des Wertes ist abhängig vom definierten Datentyp im Data-Dictionary (Verschlüsselungsclient muss wissen in welchen Datentyp er den Wert dekodierten muss, dies ist vor allem relevant bei verschlüsselten Zeichenketten und Binrädaten).

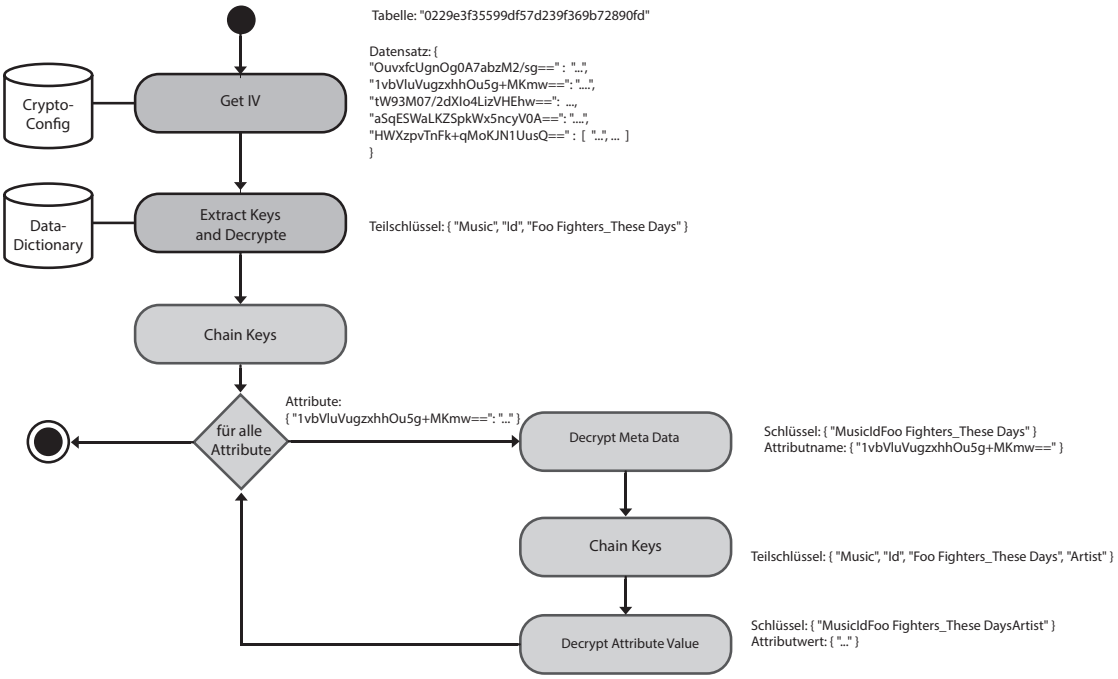

Abbildung 5.2: Ablauf der Entschlüsselung eines Datensatzes.

[2]OPE wird sowohl bei dem einfachen als auch beim zusammengesetzten Schlüsselschema angewendet.

Darüber hinaus ist in Beispiel 5.2 ersichtlich, dass sich die Darstellung der Verschlüsselung des Tabellennamens von jener der Attribute und Werte unterscheidet. Die DynamoDB erlaubt, wie bereits in Abschnitt 2.6.3 erörtert, nur einen gewissen Zeichensatz für Tabellennamen. Die Base64-Kodierung beinhaltet Sonderzeichen wie „=“, welche im Tabellennamen nicht erlaubt sind. Aus diesem Grund kodiert der Verschlüsselungsclient die Tabellennamen nach ihrer Verschlüsselung in eine hexadezimale Darstellung.

5.4 Crypto-Modul

Das Crypto-Modul stellt eine Sammlung von diversen Verschlüsselungsalgorithmen und kryptographischen Hashfunktionen zur Verfügung, die über eine einheitliche Schnittstelle angesprochen werden können. In dieser Sammlung sind unter anderem AES und DES enthalten. Die unterstützten Hashfunktionen umfassen die Hashalgorithmen SHA256 und SHA512.

Neben diesen Standardalgorithmen wird ein *Order-Preserving-Encryption*-Algorithmus (OPE), vgl. Abschnitt 3.1.3, angeboten. Er erlaubt es, Integer-Werte in einen verschlüsselten Integer umzuwandeln. Dieser Algorithmus wurde von CryptDB, siehe Abschnitt 3.2.4, implementiert und von dort entnommen. Er wurde in C++ verfasst und von der CryptDB Build-Umgebung (GCC) nach Windows Visual C++ portiert, um damit Windows Binarys kompilieren zu können. Der hier implementiere Verschlüsselungsclient wurde mit der Programmiersprache Java auf Windows entwickelt, weswegen der C++-Code über die JNI-Schnittstelle[3] angebunden werden musste. Die Inputparameter des OPE-Algorithmus sind der Verschlüsselungsschlüssel, der zu verschlüsselnde Wert und die Klartext- und Schlüsseltextlänge, die in der CryptoConfig (siehe Abschnitt 5.2.1) definiert worden sind. Die Klartext- und Schlüsseltextlänge wirken sich auf die Ausführungszeit des OPE aus, da größere Werte mehr Berechnungsschritte benötigen. In dieser Arbeit wird OPE standardmäßig mit einer Klartextgröße von 64-Bit und einer Schlüsseltextlänge von 128-Bit verwendet.

OPE retourniert einen verschlüsselten Integer-Wert. Bei der Konfiguration des OPE muss darauf geachtet werden, dass die eingesetzte Schlüsseltextgröße den maximalen numerischen Wertebereich der DynamoDB nicht überschreitet (vgl. Abschnitt 2.6.1).

Da der Einsatz von OPE sehr ressourcenintensiv ist, wird dieser nur für Schlüsselparameter bzw. für indizierende Attribute eingesetzt. Zusätzlich ist ein clientseitiger Cache für OPE implementiert. Dieser speichert die Input- und Output-Werte sowie deren Schlüssel, um redundante Berechnungen zu vermeiden. Der Benutzer kann den Cache per API-Aufruf clientseitig je nach Wunsch leeren. Der Cache behält alle Werte im Speicher, bis dieser explizit geleert wird. Die Konfiguration der Verschlüsselung und insbesondere des OPE erfolgt in der CryptoConfig-Konfigurationsdatei, siehe Abschnitte 5.2.1 und 5.2.5.

[3] Java Native Interface (JNI)

5.5 Index-Modul

Die Durchführung von *Scan*-Abfragen (siehe Abschnitt 2.6.5) ist auf HDSE verschlüsselten Daten nicht möglich. Um diese Funktionalität dennoch zur Verfügung zu stellen, wird ein Index auf die Daten erstellt. In dieser Arbeit werden zwei unterschiedliche Indizes implementiert. Der erste in diesem Abschnitt beschriebene Index ist der Bucketing-Index. Der zweite Index ist ein verschlüsselter invertierter Index. Welche Implementierung zum Einsatz kommt, wird, wie bereits erörtert, in der Crypto-Konfigurationsdatei (siehe Scan-Konfiguration in Abschnitt 5.2.4) festgelegt. Der Verschlüsselungsclient kann immer nur eine Art von Index für alle Tabellen einsetzen. Für welche Attribute ein Index erstellt wird, wird im Data-Dictionary festgelegt.

5.5.1 Bucketing-Index

Die ursprüngliche Idee des Bucketing-Verschlüsselungsansatzes (siehe Abschnitt 3.2.5) ist es, einen kompletten Datensatz zu verschlüsseln und pro verschlüsselter Datenbankspalte unverschlüsselte und somit indizierbare Zusatzinformation zu speichern. Diese Zusatzinformation repräsentiert den Inhalt der originalen Datenbankspalte in einer numerischen Form. Dies erlaubt die serverseitige Realisierung von Abfragen auf verschlüsselte Daten, im Besonderen von Bereichs- und Konjunktion-Abfragen, welche auf die numerische Zusatzinformation abgesetzt werden können.

Diese Idee wurde für den Verschlüsselungsclient adaptiert, sodass auch Bereichsabfragen auf mit HDSE verschlüsselte Daten durchgeführt werden können. Diese Art von Index wird in der vorliegenden Arbeit nur für numerische Werte verwendet. Die Datensätze werden, wie bereits erörtert, nach dem HDSE-Ansatz verschlüsselt (siehe Abschnitt 5.3), aber die Zusatzinformation des zu indizierendem numerischen Attributs wird entsprechend des Bucketing-Index geschrieben. Die Zusatzinformation kann nicht nach dem HDSE verschlüsselt werden, da dies durch die unterschiedlichen Verschlüsselungsschlüssel in nicht vergleichbare Werte resultiert (zellenbasierte Verschlüsselung). Daher wird für diese Zusatzinformation über die gesamte Datentabelle (aber nur für diese „Spalte“) ein identischer Verschlüsselungsschlüssel eingesetzt (spaltenbasierte Verschlüsselung) welcher es erlaubt, Bereichsabfragen zu realisieren. Dieser Verschlüsselungsschlüssel basiert auf folgenden verketteten Teilschlüsseln: dem IV, dem Tabellennamen und dem Attributnamen der Zusatzinformation.

Die Zusatzinformation, wie bereits in Abschnitt 3.2.5 erörtert, bildet den eigentlichen Wert durch eine Partitions-Funktion in einem vergleichbaren Wertebereich ab. Die realisierte Partitionsfunktion basiert bei dieser konkreten Implementierung auf der Histogramm-Technik *Equi-Width*. Diese Parameter werden in der Crypto-Config (siehe Abschnitt 5.2.4) definiert. Die Histogramm-Technik resultiert in einem Bereich (Bucket), zu welchem der eigentliche Wert zugeordnet wird. Dieser wird schlussendlich mithilfe einer Identifikationsfunktion durch einen numerischen Wert abgebildet. Die Identifikationsfunktion des Verschlüsselungsclients verwendet hierzu die native Java HashCode-Implementierung für Integer-Werte. Dieser resultierende Identifikationswert wird abschließend unter Einsatz des Verschlüsselungsschlüssels der Zusatzinformation mit dem

OPE verschlüsselt. Die Verschlüsslung mit OPE wird durchgeführt, um keinerlei Rückschlüsse (durch Domainwissen) auf die Buckets und somit die Wertebereiche zu ermöglichen.

Die Pflege des Index erfolgt durch den Verschlüsselungsclient bei jeder Schreibeoperation auf ein indiziertes Primärdatenattribut. Ein Beispiel, wie ein Datensatz in der DynamoDB unter Einsatz dieser Index-Lösung aussieht, wird in Abbildung 5.3 gezeigt. In diesem Beispiel wird ein Index für das Attribut *Year* erstellt. Das korrespondierende Index-Attribut ist *Prefix_Year*.

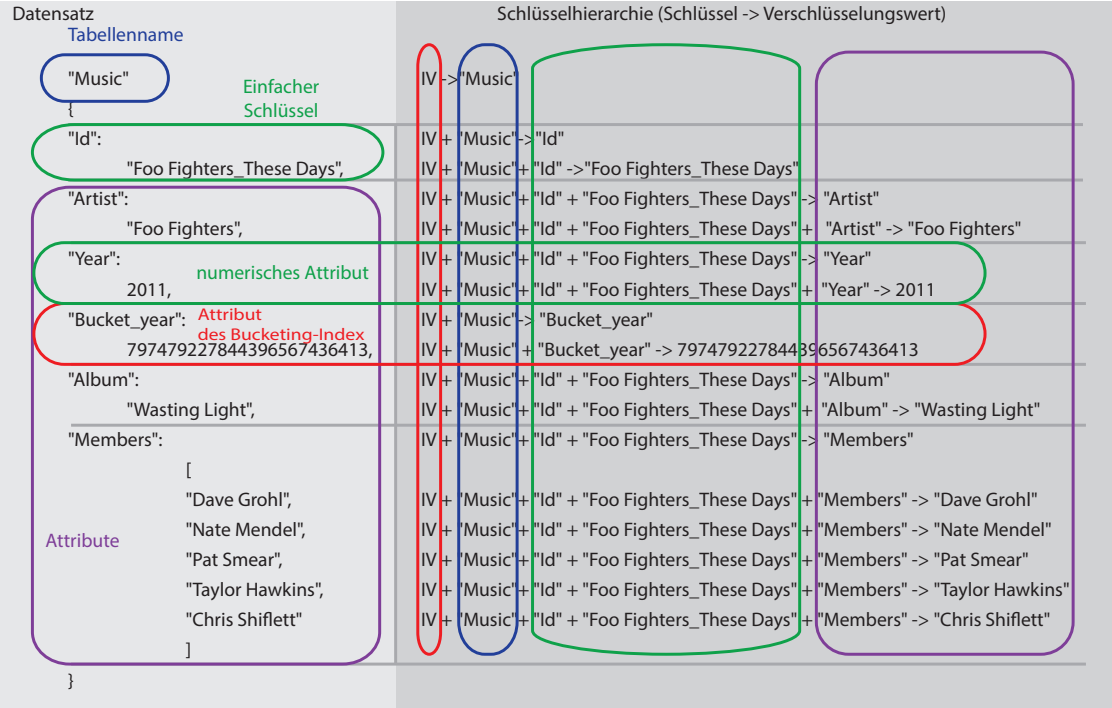

Abbildung 5.3: Ein beispielhafter Datensatz in der DynamoDB, welcher ein Attribut nach dem Bucketing-Index beinhaltet.

Die Zusatzinformation beinhaltet die Daten für das Jahr *2011*,wandelt diese basierend auf der Bucketing-Idee (Partition- und Identifikationsfunktion) in einen interpretierbaren Wert (z. B. 1234) um und verschlüsselt diesen mit OPE in *97479227. . . 6567436413*.

Diese durch den OPE verschlüsselte Zusatzinformation ermöglicht nun serverseitige Scan-Abfragen. Daher können auch Bereichs- und Konjunktiv-Abfragen durchgeführt werden. Ein typisch kodierter Datensatz ist in Beispiel 5.6 ersichtlich. Um diese Zusatzinformation interpretieren zu können, muss der Verschlüsselungsclient die Nutzerabfragen in die gleiche Form transformieren, um die Daten nach den übergebenen Bereichen zu selektieren. Hierzu wird derselbe Prozess wie zur Erstellung der Zusatzinformation durchgeführt. Würde nun der Nutzer nach dem Jahr 2011 eine Abfrage

erstellen, würde der Bucketing-Wert (inkl. OPE-Verschlüsselung mit demselben Schlüssel) *9747922784439656743641 3* und Attributname *F8dugX+qi/QkkzUgKYvYLhL7kFzX qcJh2V3s1iKq6+k=* erstellt werden. Diese sind diegleichen wie in Beispiel 5.6. Nach den eben genannten Schritten wird die Nutzer-Scan-Abfrage in eine für die Datenbank interpretierbare Form gebracht.

```
1 {
2   TableName:  0229e3f35599df57d239f369b72890fd,
3   Item:  {
4       BS:
5                       [
6                         44 61 76 65 20 47 72 6f 68 6c,
7                         4e 61 74 65 20 4d 65 6e 64 65 6c,
8                         50 61 74 20 53 6d 65 61 72,
9                         54 61 79 6c 6f 72 20 48 61 77 6b 69 6e 67 73,
10                        43 68 72 69 73 20 53 68 69 66 6c 65 74 74
11                      ]},
12      1vbVluVugzxhhOu5g+MKmw== =
13        { B: 32 30 31 31 0d 0a,},
14      OuvxfcUgnOg0A7abzM2/sg== =
15        { B: 46 6f 6f 20 46 69 67 68 74 65 72 73 5f 54 68 65 73 65 20 44 61
      79 73,},
16      tW93M07/2dXIo4LizVHEhw== =
17        { B: 46 6f 6f 20 46 69 67 68 74 65 72 73 20,},
18      aSqESWaLKZSpkWx5ncyV0A== =
19        { B: 57 61 73 74 69 6e 67 20 4c 69 67 68 74,}
20      },      F8dugX+qi/QkkzUgKYvYLhL7kFzXqcJh2V3s1iKq6+k= =
21        {S: 9747922784439656743641 3,}
22      },
23 }
```

Codebeispiel 5.6: Codebeispiel des verschlüsselten Datensatzes (aus Abbildung 5.3), welcher ein Bucketing-Attribut besitzt.

Dieser Bucketing-Index erlaubt nur eine grobe serverseitige Filterung, weswegen der Verschlüsselungsclient die geladenen Daten zusätzlich filtern muss (Post-Filtering). Es können nur folgende Scan-Operationen auf numerische Daten angewendet werden: `equals` (EQ), `not equals` (NE), `less equals` (LE), `less than` (LT), `greater equals` (GE), `greater than` (GT) und `between` (BETWEEN). Zusätzlich realisiert der Verschlüsselungsclient bei dieser Index-Implementierung die Anwendung des logischen Operators *AND* und somit die Durchführung von Konjunktion -Abfragen durch die clientseitige Filterung der Daten.

5.5.2 Verschlüsselter invertierter Index

Das Konzept des verschlüsselten invertierten Index basiert auf der Idee der *Searchable Symmetric Encryption*, welche in Abschnitt 3.2.8 beschrieben wird. Die Idee dahinter ist es, für eine beliebig verschlüsselte Datenbank einen eigenen verschlüsselten invertierten

Index zu erstellen. Das Grundkonzept eines invertierten Index ist es, eine Menge an Daten (z. B. Datensätze) anhand eines indizierten Wertes zu identifizieren.

Der Verschlüsselungsclient erstellt eine eigene invertierte Index-Tabelle pro zu indizierendem Attribut unter dem Einsatz eines zusammengesetzten Schlüsselschemas in der DynamoDB. Dieses Schlüsselschema erlaubt die direkte Realisierung eines solchen Index unter Einsatz des DynamoDB-Datenmodells und der bereits implementierten Verschlüsselungsfunktionalität durch den Verschlüsselungsclient.

Der Gedanke hinter der eigenen Index-Tabelle pro zu indizierendem Attribut ist, dass die Realisierung des invertierten Index unter Einsatz des zusammengesetzten Schlüssels für die Index-Tabelle einfach durchführbar ist. Der zu indizierende Wert wird hierbei als einfacher Schlüssel verwendet und der Primärschlüssel des indizierten Datensatzes als Bereichsschlüssel. Ein Beispiel hierzu (welches nachfolgend auch in Codebeispiel 5.4b gezeigt wird) ist eine Index-Tabelle für das Attribut *Artist* mit dem Namen *Index_Artist*. Diese Tabelle besitzt einen Datensatz, bestehend aus dem einfachen Schlüssel *Foo Fighters*, der auf den Datensatz mit dem Schlüssel *Foo Fighters_These Days* in der originalen Tabelle *Music* referenziert. Der Vorteil ist, dass keine Duplikate innerhalb eines Datensatzes (z. B. Duplikationen von Referenzen) möglich sind. Dies ist garantiert durch den Bereichsschlüssel der DynamoDB, nachdem keine Duplikate von Primärschlüsseln erlaubt sind. Ein weiterer Vorteil dieser Lösung ist, dass eine beliebige Anzahl an Referenzen zu einem Wert gespeichert werden können, ohne eine Einschränkung der DynamoDB zu erfahren. Würde man stattdessen zu einem indizierten Wert alle Referenzen innerhalb eines Datensatzes speichern, würde dieser anhand der DynamoDB-Datensatz-Größe von 64 KB (siehe Abschnitt 2.6.3) beschränkt sein. Nichtsdestotrotz verlangt die DynamoDB, dass jeder Datensatz mindestens ein nicht-Schlüsselattribut besitzt. Die soeben beschriebene Lösung benötigt jedoch nur die Schlüsselparameter, daher wird zu jedem Datensatz ein Dummy-Attribut geschrieben. Ein vollständiges Beispiel, ausgehend von einen Beispieldatensatz, und die Erstellung von Indextabellen ist in Codebeispiel 5.4b ersichtlich.

```
{
  TableName:  Music,
  Item:
    {
      Year = { N: 2011, },
      Members = { SS:
            [
              Dave Grohl,  Nate Mendel,
              Pat Smear,  Taylor Hawkings,
              Chris Shiflett
            ],
      },
      Id = { S: Foo Fighters_These Days,},
      Artist = { S: Foo Fighters,},
      Album = { S: Wasting Light,}
    },
}
```

(a) Codebeispiel eines DynamoDB-Datensatzes in der Tabelle Music.

```
{
  TableName: Index_Artist
  Item:
    {
      EinfacherSchluessel = { S: Foo Fighters },
      BereichsSchluessel = { S: Foo Fighters_These Days },
      DummyAttribute = { S: Dummy Wert }

    }
}
```

(b) Codebeispiel der Index-Tabelle *Index_Artist* für das Attribute *Artist*.

```
{
  TableName: Index_Year
  Item:
    {
      EinfacherSchluessel = { N: 2011 },
      BereichsSchluessel = { S: Foo Fighters_These Days },
      DummyAttribute = { S: Dummy Wert }

    }
}
```

(c) Codebeispiel der Index-Tabelle *Index_Year* für das Attribute *Year*.

Im Vergleich zur Bucketing-Lösung erlaubt dieser invertierte Index die Indizierung

```
{
  TableName: Index_Members
  Item:
    {
      EinfacherSchluessel = { S: Dave Grohl },
      BereichsSchluessel = { S: Foo Fighters_These Days },
      DummyAttribute = { S: Dummy Wert }

    },
  {
      EinfacherSchluessel = { S: Nate Mendel },
      BereichsSchluessel = { S: Foo Fighters_These Days },
      DummyAttribute = { S: Dummy Wert }

    },
  {
      EinfacherSchluessel = { S: Pat Smear },
      BereichsSchluessel = { S: Foo Fighters_These Days },
      DummyAttribute = { S: Dummy Wert }

    },
  {
      EinfacherSchluessel = { S: Taylor Hawkings },
      BereichsSchluessel = { S: Foo Fighters_These Days },
      DummyAttribute = { S: Dummy Wert }

    },
  {
      EinfacherSchluessel = { S: Chris Shiflett },
      BereichsSchluessel = { S: Foo Fighters_These Days },
      DummyAttribute = { S: Dummy Wert }

    }

}
```

(d) Codebeispiel der Index-Tabelle *Index_Members* für das Attribut *Members*.

Abbildung 5.4: Beispiel eines Datensatzes in der DynamoDB und dessen verschlüsselten Index-Tabellen.

von Attributen beliebiger Datentypen (Zeichenketten, Numerische Werte und Binärdaten) sowie Mengen. Eine mögliche Anwendung für einen Index auf eine Menge ist in Beispiel 5.4d ersichtlich.

Ein weiterer Vorteil der Erstellung eigener Index-Tabellen ist, wie zuvor beschrieben, die Wiederverwendung der bereits implementierten Verschlüsselungsfunktionalität durch

den Verschlüsselungsclient. Jede Indextabelle wird nach dem HDSE-Ansatz durch die existierende Implementierung verschlüsselt.

Die Erstellung der Index-Tabellen erfolgt implizit mit der Erstellung der Primärdatentabellen. Zu diesem Zeitpunkt muss bereits bekannt sein, welche Attribute zu indizieren sind. Die Verwaltung der Index-Tabellen erfolgt durch das Index-Modul, indem bei jeder Schreiboperation eigene Operationen für die Index-Tabellen erstellt und ausgeführt werden. Diese Index-Operationen sind:

DeleteItem – Zuerst wird versucht, einen eventuell vorhandenen und veralteten Indexeintrag zu löschen.

PutItem – Das Schreiben des Index-Datensatzes. Die DynamoDB erlaubt keine Updates auf den Primärschlüssel, daher muss explizit eine PutItem-Operation ausgeführt werden.

Das bedeutet, dass pro Schreibeoperation immer zwei weitere Operationen für eine Tabelle pro indiziertem Attribut ausgeführt werden. Diese zahlreichen Schreibeoperationen auf unterschiedlichen Tabellen können zu Inkonsistenzen führen, falls Fehler auftreten und manche Operationen nicht erfolgreich abgeschlossen werden können. Wie in Abschnitt 2.6 besprochen, bietet die DynamoDB keine ACID-Transaktionen an, um die Effekte dieser Transaktionen zu isolieren und ggf. zurückzurollen. Um der Problematik eines möglichen Ausfalles und Index-Inkonsistenzen entgegenzuwirken, wurde ein Log-Recovery-Mechanismus implementiert. Hierbei werden die ausgeführten Operationen (z. B. die `PutItem-Operation` auf die Primärdatentabelle) in ihrer Ausführungsreihenfolge in Log-Dateien geschrieben. Diese Dateien werden von einem `Backgroundworker` im Hintergrund geladen und in die Datenbank implementiert. In anderen Worten, es wird zuerst der originale Datensatz geschrieben und anschließend der Index. Kommt es zu einem Ausfall des Clients, werden die Daten nachträglich, nach Reaktivierung des Clients, in die Daten propagiert. Eine Log-Datei wird erst nach erfolgreicher Durchführung gelöscht. Im Fall eines Ausfalls wird die letzte ausgeführte Operation durch die Log-Datei wiederholt. Der `Backgroundworker` wird solange im Hintergrund ausgeführt(selbst nach Beendigung der Client-Applikation), bis die letzten Daten im Index geschrieben worden sind. Der `Backgroundworker` realisiert daher im wesentlichen das Konsistenzmodell BASE, indem der Index *irgendwann* einen konsistenten Zustand erreicht. Anzumerken ist, dass in dieser Arbeit angenommen wird, dass der temporäre clientseitige Speicherort der Log-Dateien hinreichend *ausfallsicher* ist, und nur eine kleine Datenmenge am Client gehalten wird bevor diese zur DynamoDB übertragen wird.

Der `Backgroundworker` besitzt die Möglichkeit, die Operation auf die Index-Tabellen mithilfe von asynchronen Operationen auf die DynamoDB durchzuführen. Dies ist im Data-Dictionary konfigurierbar, standardmäßig werden synchrone Operationen dafür verwendet.

Erstellt der Nutzer nun eine Scan-Abfrage auf eine Tabelle, wandelt der Verschlüsselungsclient diese in je eine verschlüsselte Scan-Abfrage auf eine der Index-Tabellen um. Anschließend werden diese Index-Referenzen entschlüsselt und zu BatchGetItem-Abfragen weiterverarbeitet. Diese BatchGetItem-Abfragen erlauben es, zahlreiche Datensätze aus der Primärdatentabelle über ihre Schlüssel zu laden. Die DynamoDB lädt

somit nur die ersten 1 MB oder die ersten 100 Datensätze aus der Primärdatentabelle. Findet die DynamoDB mehr Resultate, wird eine Referenz auf den zuletzt durchsuchten Datensatz zum Client retourniert. Die Verarbeitung weiterer Ergebnisse ist im Verschlüsselungsclient nicht implementiert. Abschließend werden die geladenen und entschlüsselten BatchGetItem-Ergebnisse nochmals clientseitig gefiltert.

Die clientseitige Filterung erlaubt die Kombination von mehreren Scan-Bedingungen auf Attribute (und die unterschiedlichen Ergebnismengen der Index-Tabellen) und somit die Realisierung von Konjunktiv-Abfragen. Hierzu wird die gleiche Filterfunktionalität für Konjunktiv-Abfragen wie beim Bucketing-Index implementiert: `equals` (EQ), `not equals` (NE), `less equals` (LE), `less than` (LT), `greater equals` (GE), `greater than` (GT) und `between` (BETWEEN). In dieser Implementierung wird ebenfalls der logische Operator *AND* für Konjunktiv-Abfragen realisiert. Wird vom Nutzer eine Abfrage auf eine Menge erstellt, wird diese, bedingt durch das Schema der Indextabellen, in eine Gleichheitsabfrage (GE) umgewandelt. Dies erlaubt implizit die Realisierung von `contains` (CONTAINS).

Kapitel 6

Laufzeittests

Die Implementierung des Verschlüsselungsclients wird mit einer repräsentativen Datenmenge getestet, um die Auswirkung der Verschlüsselung und der Index-Implementierung auf die Laufzeit zu eruieren. Die Messung der Laufzeit erfolgt modulabhängig, basierend auf der Architektur des Verschlüsselungsclients. In dieser Arbeit werden drei unterschiedliche Tests durchgeführt, welche in diesem Kapitel erörtert werden. In den weiteren Abschnitten dieses Kapitels werden die verwendeten Testdaten und der Aufbau der Testkonfiguration beschrieben. Abschließend werden die Ergebnisse präsentiert.

6.1 Testablauf

In diesem Abschnitt wird der allgemeine Testablauf beschrieben und erörtert, wie die Daten gemessen wurden.

6.1.1 Allgemeiner Ablauf

In dieser Arbeit werden drei unterschiedliche Tests durchgeführt. Diese sollen zeigen, welche Auswirkungen die Verschlüsselung auf die Laufzeit der Datenbankoperationen hat. Wie bereits in den vorhergegangenen Abschnitten erörtert, hat die Verschlüsselung Einfluss auf die Datenbankfunktionalität, daher ist die Scan-Operation nur mit zusätzlichen Indexstrukturen realisierbar. Aus diesem Grund wird im ersten Testfall der *Standardbetrieb* des Verschlüsselungsclients ohne zusätzliche Index-Strukturen untersucht. Die weiteren Testfälle untersuchen die Laufzeit des Verschlüsselungsclients bei Verwendung der Index-Strukturen. In jedem Testfall werden die Ergebnisse des Verschlüsselungsclients mit jenen des unverschlüsselten Clients verglichen, um aufzuzeigen, wie sehr sich die Verschlüsselung auf die Laufzeit auswirkt.

Jeder dieser Testfälle, der Datenmanipulationen und Leseoperationen realisiert, führt diegleichen Operationen 1000-Mal durch, um nicht durch Laufzeitschwankungen beeinflusst zu werden. Die Query- und Scan-Operationen werden im Vergleich nur 100-Mal ausgeführt. Die Tests werden alle mithilfe des synchronen AWS DynamoDBClients durchgeführt. Des Weiteren wird der Cache des OPE-Moduls nach jedem Testlauf und

nach jeder Ausführung einer Scan- und Query-Operation geleert.

Testfall 1: Standardbetrieb

Der Standardbetrieb untersucht die Laufzeit der Datenbankfunktionalität ohne zusätzlich benötigte Index-Strukturen für die Scan-Operation. Es werden die nachfolgenden Datenbankoperationen unter Einsatz beider Schlüsselschemata, dem einfachen Schlüssel und dem Bereichsschlüssel, durchgeführt. Bei diesem Testfall werden die Operationen PutItem, UpdateItem, DeleteItem, GetItem und Query untersucht. Die Query-Operation erlaubt Bereichsabfragen auf Schlüsselparameter, daher wird diese nur auf eine Tabelle mit zusammengesetzten Schlüsseln durchgeführt. Bei dieser Operation werden die nachfolgenden Vergleichsoperatoren eingesetzt: `equals` (EQ), `less equals` (LE), `less than` (LT), `greater equals` (GE), `greater than` (GT) und `between` (BETWEEN)).

Testfall 2: Verschlüsselungsclient unter Einsatz des Bucketing-Index

Der zweite Testfall untersucht den Einfluss des Bucketing-Index auf die Laufzeit des Verschlüsselungsclients. Daher werden in diesem Testfall ebenfalls die Laufzeiten der Operationen PutItem, UpdateItem, DeleteItem und GetItem analysiert. Diese Operationen werden auf eine Tabelle mit einfachem Schlüsselschema durchgeführt.

Durch den Einsatz des Bucketing-Index ist es möglich, Scan-Abfragen durchzuführen. Daher werden in diesem Testfall Scan-Abfragen auf alle numerischen Attribute des Datensatzes durchgeführt. Es werden die Vergleichsoperationen (`equals` (EQ), `less equals` (LE), `less than` (LT), `greater equals` (GE), `greater than` (GT) und `between` (BETWEEN)) mit der Scan-Operation untersucht.

Des Weiteren erlaubt die Scan-Operation auch Konjunktiv-Abfragen. Bei diesem Durchlauf werden Gleichheits- (EQ) und Bereichsabfragen (BETWEEN) mit dem logischen And-Operator, verknüpft auf numerische Attribute, durchgeführt. Die Tests für Konjunktiv-Abfragen beginnen bei einem Abfragekriterium (einfache Abfrage) und verknüpfen pro Testablauf ein zusätzliches Abfragekriterium damit. Dies wird so lange durchgeführt, bis alle numerischen Attribute des Datensatzes verwendet werden. Somit ist es möglich, die Laufzeit für komplexere Konjunktiv-Abfragen zu eruieren.

Die Ergebnisse dieses Testfalls werden mit jenen des ersten Testlaufs verglichen. Ebenso erfolgt ein Vergleich der Laufzeit der Scan-Operation, durchgeführt mit dem unverschlüsselten DynamoDBClient, und den Laufzeitergebnissen aus diesem Testfall.

Testfall 3: Verschlüsselungsclient unter Einsatz des invertierten Index

Der letzte Testfall wiederholt die Abläufe des zweiten Tests unter Einsatz des verschlüsselten invertierten Index. Auch hier werden die Ergebnisse der Datenbankoperationen mit dem unverschlüsselten und verschlüsselten Client verglichen. Es kommt ebenfalls zum Vergleich der unterschiedlichen Index-Implementierungen.

Wie die Messung der Ausführungszeiten der Testfälle vonstatten geht, wird im nachfolgenden Abschnitt erörtert.

6.1.2 Messung der Ausführungszeiten

Die Messung der Ausführungszeiten der Datenbankoperation wird pro durchlaufenem Modul, basierend auf der Architektur des Verschlüsselungsclients (vgl. Abschnitt 4.2), gemessen. Die Laufzeit wird mithilfe des Java ETM-Tools[1] gemessen. Dieses Tool zählt die Anzahl der Aufrufe und misst die gesamte Ausführungszeit, die Durchschnitts-, Minimum- und Maximum-Ausführungszeit in Millisekunden. Hierzu wurden die Module der Implementierung des Verschlüsselungsclients mithilfe des ETM-Tools instrumentiert.

Das ETM-Tool misst die nachfolgenden Bestandteile:

Encrypted-Client-Core - Es wird die Laufzeit eines Aufrufes mit dem Verschlüsselungsclient gemessen.

Crypto-Modul - Es wird die Laufzeit des eingesetzten Verschlüsselungsalgorithmus (z. B. AES oder OPE) zur Ver- und Entschlüsselung gemessen.

Amazon DynamoDbClient - Diese API stellt die Kommunikations-Schnittstelle zur Datenbank dar. Das ETM-Tool misst die Laufzeit der aktuell ausgeführten Operation. Diese Laufzeit inkludiert die Übertagungszeit zum Server, die Verarbeitungszeit am Server und die Antwortzeit des Servers.

Index-Modul - Die Messung des Index ist abhängig von der eingesetzten Implementierung. Der invertierte Index ist komplexer und wird daher im Detail gemessen. Der Mehraufwand der Laufzeit vom Bucketing-Index besteht hingegen fast ausschließlich aus der Laufzeit der OPE Operationen, weswegen hier die Laufzeit vom Bucketing-Index nur indirekt über die OPE Laufzeit gemessen wird.

In diesem Abschnitt wird der Testablauf erörtert, sowie die Messung der Daten. Im nachfolgenden Abschnitt werden die eingesetzten Testdaten gezeigt.

6.2 Daten und Datenaufbau

Als für die Laufzeittests repräsentative Datenquelle wird der Datensatz des *Millon Song Dataset* [58] verwendet. Hierbei handelt es sich um eine frei verfügbare Sammlung von Daten von diversen Künstlern. Diese Daten enthalten Detailinformationen über Höhen und Tiefen der Lieder sowie Meta-Informationen wie Genre und ähnliche Interpreten (siehe Tabellen 6.1 und 6.2). Der Datenbestand beinhaltet Lieder ab dem Jahr 1922 [58]. Ein Lied besteht aus zahlreichen Attributen, welche Werte über den Künstler und das Lied selbst beinhalten. Zusätzlich ermöglichen es diese Testdaten, alle Datentypen und Mengen, welche in der DynamoDB verfügbar sind, einzusetzen (siehe Abschnitt 2.6.1). Da jedoch der OPE-Algorithmus nur Integer-Werte erlaubt und auf numerische Attribute ein Index erstellt wird, werden Double-Werte gerundet. Um auch binäre Datentypen in den Tests einzusetzen, werden akustische Informationen des Liedes (Pitches, Timbre und Loudness) in eine binäre Form umgewandelt und anschließend in der DynamoDB abgelegt.

[1] Java Execution Time Measurement Library - http://jetm.void.fm/

Diese Daten werden in den Tabellen der DynamoDB entsprechend ihrer unterschiedlichen Schlüsselschemata abgespeichert. Der Primärschlüssel des einfachen Schlüsselschemas verwendet die Informationen über den Künstler und den Namen des Liedes, (`ArtistName` und `SongName`) indem er diese beiden Werte miteinander zu einer Zeichenkette verkettet. Dieser resultierende Wert wird unter dem Attributnamen `ArtistKey` verwaltet (vgl. Tabelle 6.1). Beim zusammengesetzten Schlüssel wird der einfache Schlüssel `ArtistKey` um den Bereichsschlüssel `year`, dem Veröffentlichungsjahr des Liedes, erweitert (vgl. Tabelle 6.2).

In den Tabellen 6.1 und 6.2 werden die eingesetzten Datensätze mit ihren Datentypen (siehe Tabelle 2.3 für Beschreibung der Datentypen) nach Schlüsselschema angeführt.

Attributname	Datentyp
`ArtistKey`	S
year	N
artist_name	S
songTitle	SS
artistLocation	S
artistLatitude	N
artistLongitude	N
artist_familarity	N
artistID	S
artistMbid	S
artistPlaymeid	N
artistdigitalId	N
similarArtists	SS
artistTerms	SS
artistTermFreq	NS
artistTermWeight	NS
artist_hotness	N
acousticFeatures	B

Abbildung 6.1: Datensatzschema für die Tabelle mit einfachem Schlüssel. Das Schlüsselattribut ist `ArtistKey`.

Attributname	Datentyp
`artist_name`	S
`year`	N
songTitle	SS
artistLocation	S
artistLatitude	N
artistLongitude	N
artist_familarity	N
artistID	S
artistMbid	S
artistPlaymeid	N
artistdigitalId	N
similarArtists	SS
artistTerms	SS
artistTermFreq	NS
artistTermWeight	NS
artist_hotness	N
acousticFeatures	B

Abbildung 6.2: Datensatzschema für die Tabelle mit einem zusammengesetzten Schlüssel. Das Attribut `artist_name` ist der Hash-Wert und das Attribut `year` der Bereichsschlüssel.

Diese Testdaten wurden erstellt, um Forschern eine große Datenmenge für Forschungsalgorithmen (z. B. in den Bereichen Skalierung und Data-Mining) zur Verfügung zu stellen. Der Datensatz ist insgesamt 300 GB groß und bietet auch eine Untermenge, die aus 10.000 Liedern und Künstlerinformationen besteht. Basierend auf den definierten Tests aus Abschnitt 6.1.1 werden aus dieser Untermenge in der vorliegenden Arbeit die ersten 1000 Lieder verarbeitet.

6.3 Testkonfiguration

In diesem Abschnitt wird die Konfiguration der Testumgebung beschrieben. Im darauffolgenden Teilbereich wird die Konfiguration der Datenbank gezeigt. Diese ist für alle Testfälle gleich. Im Unterschied zur Datenbank-Konfiguration sind das Data-Dictionary und die Crypto-Config abhängig vom Testfall. Abschließend wird noch erörtert, wie die beschriebenen Daten für die Testfälle eingesetzt werden.

6.3.1 Konfiguration der Amazon DynamoDB-Datenbank

Die DynamoDB erlaubt es, Metriken wie die Lese- und Schreibkapazitäten per Tabelle zu konfigurieren. Diese Metriken werden auch *Read-*[2] und *Write-Throughput*[3] genannt und werden in Kapazitäten definiert. Für weitere Informationen siehe Abschnitt 2.6.6. Ein zu niedriger Wert hat Auswirkung auf die Ausführungszeit der Datenbank selbst, da es zu einer Drosselung der Lese- und Schreibkapazitäten seitens der Datenbank kommen kann. Daher wurden nachfolgende Konfigurationen durchgeführt:

Tabelle 6.1: Lese- und Schreibkapazitäten für die Primärdatentabelle.

	Kapazitäten
Lesekapazität	80 Einheiten[2]
Schreibkapazität	80 Einheiten[3]

Tabelle 6.2: Lese- und Schreibkapazitäten für die Index-Tabellen.

	Kapazitäten
Lesekapazität	8 Einheiten[2]
Schreibkapazität	12 Einheiten[3]

Die eingesetzte Amazon DynamoDB befindet sich im europäischen Datenzentrum von Amazon in Irland (Region *EU_WEST_1*[4]).

6.3.2 Konfiguration des Verschlüsselungsclients

Die Konfiguration des Verschlüsselungsclients ist abhängig von den verwendeten Testdaten und vom konkreten Testlauf. In diesem Abschnitt wird die Basiskonfiguration des Clients gezeigt. In den Testfällen wird nur der Einsatz des Index in der Konfiguration adaptiert. Alle weiteren Werte bleiben wie hier beschrieben. Die Crypto-Config ist in Codebeispiel 6.1 ersichtlich. Es wird ein Blockverschlüsselungsalgorithmus AES-128 und PKC5Padding angewendet (vgl. Abschnitt 3.1.1). AES wird mit dem Mode of Operation CBC (vgl. Abschnitt 3.1.1) durchgeführt. Des Weiteren wird zur Schlüsselverkettung die Option Chaining aus dem Abschnitt 5.2.3 verwendet. OPE wird auf eine Schlüsseltextgröße von 128-Bit und eine Klartextgröße von 64-Bit konfiguriert. Die Konfiguration der

[2] Anzahl der zu lesenden Datensätze pro Sekunde x 4 KB Datensatzgröße [14]
[3] Anzahl der zu schreibenden Datensätze pro Sekunde x 1 KB Datensatzgröße [14]
[4] Die DynamoDB erlaubt es, die Daten in unterschiedlichen Rechenzentren der Welt (Amerika, Asien etc.) abzuspeichern. Um etwaige Latenz bzgl. weiterer Distanzen ausschließen zu können, wird das Rechenzentrum in der EU (Irland) verwendet.

```
<cryptoData>
  <algorithm length="128" name="AES" padding="PKCS5Padding"
    operationMode="CBC" hashProvider="SHA256" />
  <iv>ZFfd5+hkCsuQfp13jDEnJQ==</iv>
  <keyGeneration useMode="Chaining" />
  <opeEncryption plaintextSizes="64" ciphertextSizes="128"/>
  <!-- Testfall 1:  Standardbetrieb -->
  <scanOperation useMode="NoScanOperationPossible">
  </scanOperation>

  <!-- Testfall 2: Bucketing-Index -->
  <scanOperation  useMode="PartitionAndIdentification">
      <HistogramHistogramConstructionTechnique use="EquiWidth">
      <DomainRange min="1922" max="2015" />
      <DataAmount value="1000"/>
    </HistogramHistogramConstructionTechnique>
  </scanOperation>

  <!-- Testfall 3: Inverted encrypted Index -->
  <scanOperation  useMode="EncryptedIndex" >
    <logRecovery file="C:\Path\To\log\recovery\files" />
  </scanOperation>
</cryptoData>
```

Codebeispiel 6.1: Die Crypto-Konfiguration ohne `Scan`-Funktionalität.

`Scan`-Funktionalität ist abhängig vom durchgeführten Testfall. Die diversen Konfigurationen pro Testfall werden ebenfalls in diesem Beispiel gezeigt.

Die Daten im *Data-Dictionary* müssen, abhängig vom Testfall, definiert werden. In den entsprechenden Tests werden diese Daten abhängig von der benötigten Anzahl an indizierten Attributen adaptiert (durch das Data-Dictionary-Attribut *queryable* siehe Abschnitt 5.1 für weitere Informationen). In Beispiel 6.2 ist die eben genannte Konfiguration ersichtlich. In diesem Beispiel werden zwei unterschiedliche Tabellen (einfacher und zusammengesetzter Schlüssel) basierend auf den Testdaten aus Abschnitt 6.2 konfiguriert.

Die Hardware, auf welcher der Verschlüsselungsclient ausgeführt wird, ist wie folgt: Intel Core i3-3220 CPU @ 3.30 GHz, 8,00 GB RAM, 250 GB SSD auf Windows 7 64-Bit.

6.3.3 Konfiguration der Testdurchführung

Die Tests werden nach dem Ablauf in Abschnitt 6.2 und basierend auf den Daten von Abschnitt 6.2 durchgeführt.

Die Testabläufe für PutItem, GetItem, UpdateItem und DeleteItem sind immer wie im Pseudocode-Beispiel 6.3 beschrieben aufgebaut. In den ersten Schritten wird die Zeitmessung durch den ETM-Monitor gestartet, das Data-Dictionary gelesen und ein Objekt des Verschlüsselungsclients erstellt, das mit der DynamoDB-Region *EU_WEST_1* kommuniziert.

Anschließend wird ein Datensatz aus den Daten geparst und in die dementsprechende Datenbankabfrage formuliert. Wie die Daten der Abfrage übergeben bzw. angepasst werden, ist abhängig vom eingesetzten Schlüsselschema (siehe Abschnitt Datenaufbau

```
<EncryptedDynamoDb>
<Tables>
  <Table name="Artists_HashTableSmall">
    <attribute name="ArtistKey" keyType="HASH" type="S"  />
    <attribute name="artist_name" type="S" />
    <attribute name="year" type="N" />
    <attribute name="songTitel" type="SS" />
    <attribute name="artistLocation" type="S"  />
    <attribute name="artistLatitude" type="N"  />
    <attribute name="artistLongidue" type="N"  />
    <attribute name="artist_familiarity" type="N" />
    <attribute name="artistID" type="S" />
    <attribute name="artistMbid" type="S" />
    <attribute name="artistPlaymeid" type="N"  />
    <attribute name="artistdigitalId" type="N"  />
    <attribute name="similarArists" type="SS"/>
    <attribute name="artistTerms" type="SS" />
    <attribute name="artistTermFreq" type="NS" />
    <attribute name="artistTermWeight" type="NS" />
    <attribute name="artist_hotness" type="N"  />
    <attribute name="acousticFeatures" type="B" />
  </Table>
  <Table name="Artists_HashRangeTableSmall">
    <attribute name="artist_name" keyType="HASH" type="S"  />
    <attribute name="year" keyType="RANGE" type="N" />
    <attribute name="songTitel" type="SS"  />
    <attribute name="artistLocation" type="S"  />
    <attribute name="artistLatitude" type="N" />
    <attribute name="artistLongidue" type="N" />
    <attribute name="artist_familiarity" type="N"  />
    <attribute name="artistID" type="S"  />
    <attribute name="artistMbid" type="S" />
    <attribute name="artistPlaymeid" type="N" />
    <attribute name="artistdigitalId" type="N" />
    <attribute name="similarArists" type="SS"/>
    <attribute name="artistTerms" type="SS"/>
    <attribute name="artistTermFreq" type="NS"  />
    <attribute name="artistTermWeight" type="NS"  />
    <attribute name="artist_hotness" type="N" />
    <attribute name="acousticFeatures" type="B" />
  </Table>
</Tables>
</EncryptedDynamoDb>
```

Codebeispiel 6.2: Das eingesetzte *Data-Dictionary* beschreibt zwei unterschiedliche Tabellen. Es basiert auf dem Datensatz aus Abschnitt 6.2.

6.2) und der aktuellen Abfrage. Bei der PutItem- und UpdateItem-Operation werden die Daten einfach ohne Anpassung in die Abfrage übernommen. Bei der GetItem-Abfrage und DeleteItem-Operation wird nur der Primärschlüssel für die Abfrage eingesetzt.

Nach jeder Ausführung einer Operation wird die benötigte Zeit vom ETM-Tool gemessen und gespeichert. Am Ende des Testfalls werden die Ergebnisdateien geschrieben und der OPE-Cache geleert.

Der Testaufbau für die Query- und Scan-Operation ist etwas anders als der Testaufbau beschrieben in Beispiel 6.3. Auch hier wird am Anfang die Zeitmessung gestartet und die Initialisierung des Data-Dictionary und des Clientobjekts durchgeführt. Abhän-

```
1  startTimeMeasure();
2  boolean isHashKey = true; //true for range key
3  string tablename = "Music";
4
5    for (File musicEntry : folder.listFiles()) {
6      Data data = ParseDataFile();
7      //Version PutItem
8      PutItemRequest request = CreatePutItemRequest(data, tablename, isHashKey);
9      //Version GetItem
10     GetItemRequest request = CreateGetItemRequest(data, tablename, isHashKey);
11     //Version UpdateItem
12     UpdateItemRequest request = CreateUpdateItemRequest(data, tablename, isHashKey);
13     //Version DeleteItem
14     DeleteItemRequest request = CreateDeleteItemRequest(data, tablename, isHashKey);
15
16     EtmPoint measurePoint = etmMonitor.createMeasurePoint();
17     try {
18       //Version PutItem
19       dynamoDB.putItem(request);
20       //Version GetItem
21       dynamoDB.getItem(request);
22       //Version UpdateItem
23       dynamoDB.updateItem(request);
24       //Version DeleteItem
25       dynamoDB.deleteItem(request);
26     } finally {
27       measurePoint.collect();
28     }
29   }
30
31 initDynamo();
32 StopTimeMeasure();
33 OPECache.cleanCache();
```

Abbildung 6.3: Standardablauf eines Testfalls. Abhängig von der durchgeführten Operation wird der dementsprechende Datenbankaufruf durchgeführt.

gig vom Testfall wird ein anderer Vergleichsoperator ausgewählt, in Beispiel 6.4 wurde der Gleichheitsoperator ausgewählt. Anschließend werden die Daten aus dem Datensatz geparst und abhängig von der durchgeführten Operation (Query oder Scan) eine Abfrage erstellt. Bei der Query- und Scan-Operation werden Abfragen nur für das Attribut *Year* erstellt, hierzu wird das aktuelle Jahr des gelesenen Datensatzes verwendet. Bei der Scan-Operation, die eine Konjunktiv-Abfrage durchführt, wird, wie in Testlauf zwei erörtert, pro Testlauf eine weitere Bedingung hinzugefügt. Jede Bedingung basiert auf einem numerischen Attribut des Datensatzes. Bei den Konjunktiv-Abfragen werden nur EQ- und BETWEEN-Vergleichsoperationen durchgeführt. Auch hier wird als Input der aktuelle Datenwert des benötigten Attributes aus dem zurzeit geparsten Datensatz verwendet.

Die Between-Abfragen (Konjunktiv- sowie einfache Scan-Abfragen) müssen in diesem Testlauf gesondert verarbeitet werden. Auch hier wird als Input der aktuell geparste Datensatz und der benötigte Attributwert verwendet. Um jedoch Ober- und Untergrenzen für die Between-Abfrage zu bestimmen, werden die existierenden Datenwerte (z. B.

```
1 startTimeMeasure(); //init ETM-Monitoring Process and Start Time Measure for entire test
2 string tablename = "Music";
3 ComparisonOperator comp = ComparisonOperator.EQ; //differs per testrun
4 Map<String, Condition> conj; //Conjunction Conditions - differs per testrun
5
6   for (File musicEntry : folder.listFiles()) { //read all data files from data set folder
7     Data data = ParseDataFile();
8
9     //Version Query
10    QueryRequest request = null;
11    If(comp.equals(ComparisonOperator.BETWEEN)){
12      request = CreateQueryBetweenRequest(data, tablename);
13    }else{
14      request = CreateQueryRequest(data, tablename, comp);
15    }
16
17    //Version Scan
18    ScanRequest request = null;
19    If(comp.equals(ComparisonOperator.BETWEEN)){
20      request = CreateScanBetweenRequest(data, tablename);
21    }else{
22      request = CreateScanRequest(data, tablename, comp);
23    }
24
25    //Version Scan with Conjunction
26    ScanRequest request = CreateScanConjunctionRequest(data, tablename, conj);
27    EtmPoint measurePoint = etmMonitor.createMeasurePoint(); //start time measure for current
       query
28    try {
29      //Version Query
30      dynamoDB.query(request);
31      //Version Scan
32      dynamoDB.scan(request);
33    } finally {
34      measurePoint.collect(); //collect time measure for current query
35      OPECache.cleanCache();
36    }
37  }
38
39 initDynamo();
40 StopTimeMeasure();
41 OPECache.cleanCache();
```

Abbildung 6.4: Testablauf der Abfragen Query, Scan und Konjunktion-Abfragen.

das Year-Attribut) mithilfe von Zufallszahlen adaptiert. Es werden zwei Zufallszahlen mithilfe des *Java-Pseudo-Random-Generator* im Bereich von eins bis fünf erstellt und zum gegebenen Datenwert addiert bzw. subtrahiert. Beispielweise kann so eine Bereichsabfrage basierend auf einem Datensatz aus dem Jahr 2012 mit den Parametern 2010 als Untergrenze und 2014 als Obergrenze erstellt werden.

Nach der Ausführung der Abfrage wird die aktuelle Dauer der Durchführung gemessen und der OPE-Cache geleert, sowie die Ergebnisdateien der Zeitmessung geschrieben.

6.4 Ergebnisse

Nachfolgend werden die Ergebnisse der beschriebenen Testfälle aus Abschnitt 6.1.1 präsentiert.

6.4.1 Testfall 1: Standardbetrieb

Der Testfall 1 (siehe Abschnitt 6.1.1) umfasst den Standardbetrieb des Verschlüsselungsclients, indem keine zusätzlichen Index-Strukturen benötigt werden. Es werden PutItem, GetItem, UpdateItem, DeleteItem-Operationen auf Tabellen mit beiden Schlüsselschemata durchgeführt, sowie die Query-Abfrage auf einer Tabelle mit zusammengesetztem Schlüssel. Die Query-Abfrage wird mit den nachfolgenden Vergleichsoperatoren durchgeführt: `equals` (EQ), `less equals` (LE), `less than` (LT), `greater equals` (GE), `greater than` (GT) und `between` (BETWEEN)).

Dieser Standardbetrieb des Verschlüsselungsclients wird ebenfalls mit dem unverschlüsselten DynamoDBClient und den gleichen Operationen und Daten durchgeführt. Dies soll zeigen, in wieweit die Ausführungszeit des verschlüsselten Clients im Vergleich zum unverschlüsselten Client steigt. Die Tests werden, wie zuvor erörtert, 1000-mal durchgeführt.

In den nachfolgenden Tabellen werden die Ergebnisse (Durchschnitts-, Minimum- und Maximum-Ausführungszeit in Millisekunden) der PutItem, GetItem, UpdateItem, DeleteItem-Operation gezeigt. Diese Operationen werden auf einer Tabelle mit einem *einfachen Schlüssel*-Schema durchgeführt. In Tabelle 6.3 sind die Ergebnisse des unverschlüsselten Clients sichtbar und in Tabelle 6.4 jene des Verschlüsselungsclients.

Operation	**Ø(ms)**	**Min(ms)**	**Max(ms)**
putItem	63,81	55,90	547,65
getItem	60,40	50,96	619,74
updateItem	57,53	52,96	168,00
deleteItem	64,27	58,91	273,6

Tabelle 6.3: Datenmanipulation und Leseoperationen, durchgeführt auf eine unverschlüsselte Tabelle mit einem einfachen Schlüssel.

Die Ausführungszeiten des Verschlüsselungsclients werden, wie in Abschnitt 6.1.2 erörtert, nach Modulen gemessen. Dies ist auch in Tabelle 6.4 ersichtlich. Die Operationen mit dem Prefix *V-*, wie zum Beispiel *V-PutItem*, beinhalten die gesamte Ausführungszeit der Operation (inklusive Verschlüsslung bzw. Entschlüsselung der Daten und Übertragung zur Datenbank). Die Ver- und Entschlüsselung (AES-Enc und AES-Dec) mithilfe des AES wird in dieser Tabelle gesondert angeführt.

Operation	Aufrufe	Ø(ms)	Min(ms)	Max(ms)
putItem				
V-PutItem	1000	77,12	63,61	2.166,48
AES-Enc	164320	0,04	0,02	209,93
AES-Dec	2	0,63	0,45	0,81
AWS-Client	1000	68,07	57,78	417,99
getItem				
V-GetItem	1000	71,14	64,13	208,13
AES-Enc	3000	0,10	0,03	0,77
AES-Dec	163319	0,03	0,03	8,88
AWS-Client	1000	65,04	59,32	199,81
updateItem				
V-UpdateItem	1000	67,08	58,74	271,51
AES-Enc	166319	0,03	0,02	8,20
AWS-Client	1000	61,16	54,23	264,46
deleteItem				
V-DeleteItem	1000	62,29	57,74	180,26
AES-Enc	3000	0,10	0,03	1,120
AWS-Client	1000	61,76	57,43	177,20

Tabelle 6.4: Datenmanipulationen und Leseoperationen, durchgeführt mit dem Verschlüsselungsclient auf eine Tabelle mit einem einfachen Schlüssel.

In den Laufzeitergebnissen (vgl. Tabelle 6.4) ist auch die Zeit für die Entschlüsselung von genau zwei Datensätzen enthalten, weil bei der Datenbank die Informationen über das aktuelle Schlüsselschema der Tabelle abgefragt werden. Es werden daher zwei Entschlüsselungen für den Tabellenamen und den Bezeichner des Schlüsselattributes durchgeführt. Diese Informationen werden im flüchtigen Speicher gehalten. Dies wird bei der allerersten Operation durchgeführt. Der Großteil der benötigten Laufzeit wird jedoch vom Query-Rewriting benötigt.

Hier können bereits erste Aussagen über die Ausführungszeiten des Verschlüsselungsclients im Vergleich zum DynamoDbClient getroffen werden. Diese Ausführungszeiten werden in der Abbildung 6.5 verglichen. Die Daten entstammen den Tabellen 6.3 und 6.4. Diese Tests werden auf eine Tabelle mit einem einfachen Schlüsselschema durchgeführt. Es ist zu erkennen, dass die Verschlüsselung durch den HDSE-Ansatz mehr Zeit benötigt als jene Operationen, wo diese nicht verwendet wird.

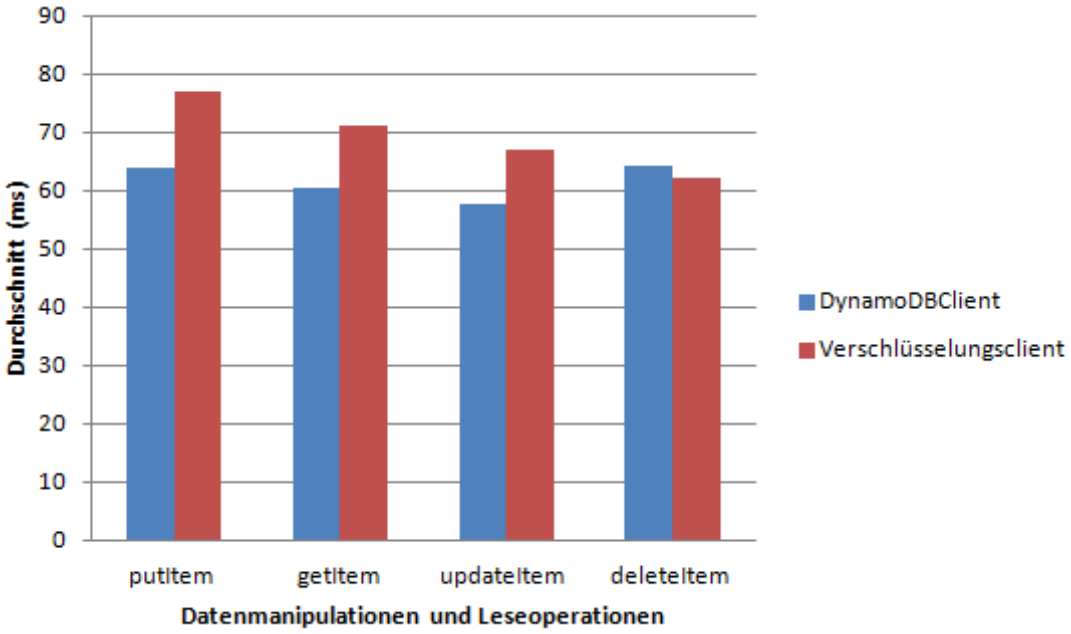

Abbildung 6.5: In diesem Diagramm werden die Ergebnisse der unverschlüsselten und verschlüsselten Tests auf Datensätze mit einem einfachen Schlüssel gezeigt. Die Werte basieren auf den Tabellen 6.3 und 6.4.

Der gleiche Testfall wurde für die Tabelle mit einem *zusammengesetzten Schlüssel* durchgeführt. In den nachfolgenden Tabellen werden die Ausführungszeiten der PutItem-, GetItem-, UpdateItem-, DeleteItem-Operation gezeigt. Anzumerken ist, dass in dieser Tabelle zum ersten Mal der OPE-Algorithmus (OPE-Enc und OPE-Dec) zum Einsatz kommt. Der Verschlüsselungsclient verwendet diesen Algorithmus immer für numerische Schlüsselparameter. In diesem Testfall wird OPE konkret für den Bereichsschlüssel eingesetzt. In Tabelle 6.5 sind die Ergebnisse des unverschlüsselten Clients sichtbar und in Tabelle 6.6 jene des Verschlüsselungsclients.

Operation	**Ø(ms)**	**Min(ms)**	**Max(ms)**
putItem	86,30	61,72	808,94
getItem	76,92	57,64	670,39
updateItem	76,87	54,34	301,94
deleteItem	67,77	50,73	303,96

Tabelle 6.5: Datenmanipulation und Leseoperationen, durchgeführt auf eine unverschlüsselte Tabelle mit einem zusammengesetzten Schlüssel.

Operation	Aufrufe	Ø(ms)	Min(ms)	Max(ms)
putItem				
V-PutItem	1000	202,09	161,51	2.120,1
AES-Enc	161320	0,040	0,025	177,27
AES-Dec	3	0,37	0,27	0,42
AWS-Client	1000	78,28	57,47	1.940,32
OPE-Enc	1000	114,86	98,60	675,83
getItem				
V-GetItem	1000	341,26	258,36	34.896,59
AES-Enc	4000	0,11	0,03	2,04
AES-Dec	160288	0,036	0,025	32,29
AWS-Client	1000	99,14	53,11	34.665,37
OPE-Enc	1000	117,61	98,64	298,55
OPE-Dec	1000	117,23	98,92	298,04
updateItem				
V-UpdateItem	1000	165,06	63,06	808,96
AES-Enc	164319	0,030	0,025	24,14
AWS-Client	1000	61,86	55,37	236,90
OPE-Enc	903	107,20	98,76	317,99
OPE-Cache-Read	2000	0,00	0,00	0,18
deleteItem				
V-DeleteItem	1000	152,64	50,78	800,15
AES-Enc	4000	0,10	0,03	0,66
AWS-Client	1000	54,83	50,36	177,42
OPE-Enc	903	106,84	98,77	271,77
OPE-Cache-Read	1000	0,00	0,00	0,06

Tabelle 6.6: Datenmanipulationen und Leseoperationen, durchgeführt mit dem Verschlüsselungsclient auf eine Tabelle mit einem zusammengesetzten Schlüssel.

Auch bei dieser Tabelle 6.6 werden beim ersten Aufruf die Schlüsseltypen am Server abgefragt. In diesem Test wurde ein zusammengesetzter Schlüssel eingesetzt, weswegen bei der ersten Operation, PutItem, drei Werte entschlüsselt werden. Es werden die Schlüsselschema-Informationen vom Server geladen, daher werden der Tabellenname und die Bezeichner des einfachen Schlüssels und des Bereichsschlüssels entschlüsselt. Die Anzahl der Aufrufe der OPE-Verschlüsselung (903) bei der UpdateItem- und DeleteItem-Operation ist darauf zurückzuführen, dass diese OPE-Werte bereits im OPE-Cache vorhanden waren, was wiederum nur dann möglich ist, wenn zu einem Datensatz mehrere Werte für dasselbe Jahr vorhanden sind (ein Künstler hat mehrere Lieder in einem Jahr veröffentlicht).

In Abbildung 6.6 ist ein Vergleich dieser Ergebnisse des Verschlüsselungsclients und des DynamoDbClient ersichtlich. Dieser Vergleich basiert auf den Tabellen 6.5 und 6.6.

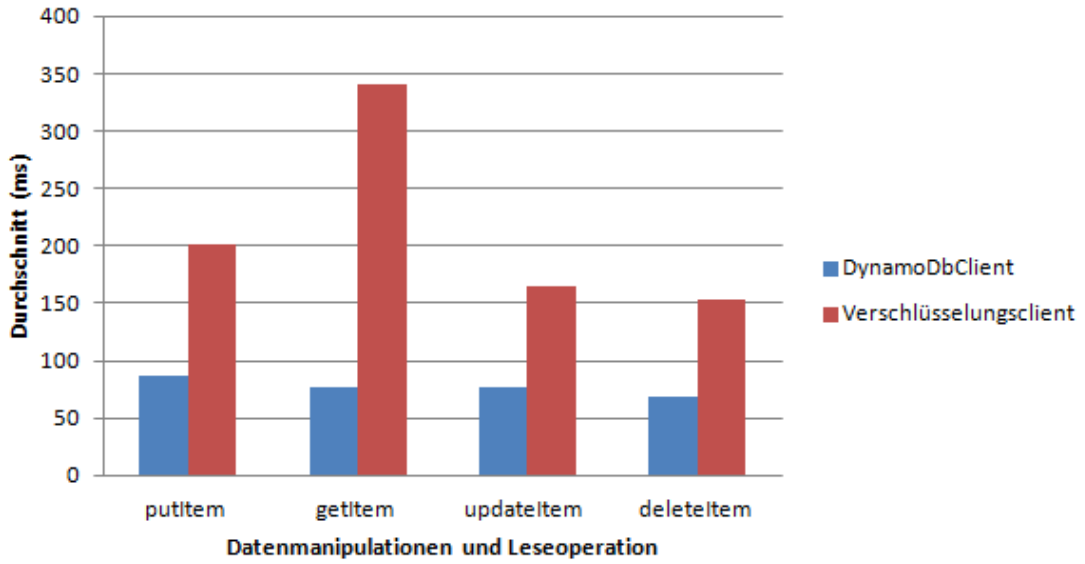

Abbildung 6.6: In diesem Diagramm werden die Ergebnisse der unverschlüsselten und verschlüsselten Tests auf Datensätze mit einem zusammengesetzten Schlüssel gezeigt.

Die Ergebnisse der Operationen auf Tabellen mit einfachen (vgl.Tabelle 6.4) und zusammengesetzten Schlüsselschemas, vgl. Tabelle 6.6), weisen unterschiedliche Ausführungszeiten auf. Die verschlüsselten Operationen auf einer Tabelle mit einem zusammengesetzten Schlüssel (siehe Tabelle 6.6) benötigen fast die doppelte Ausführungszeit. Dies ist dadurch bedingt, dass mit dem Schlüsselparameter ein numerisches Attribut zum Einsatz kommt. Durch das explizite Anführen der Ausführungszeiten der Verschlüsselungsclient-Komponenten wird gezeigt, dass der OPE-Algorithmus entscheidend mehr Zeit benötigt als die Kommunikation und Verarbeitung der Daten am Datenbankbankserver. Daraus lässt sich ableiten, dass die Laufzeit durch den Einsatz von OPE steigt.

Ein weiterer Bestandteil des ersten Testfalls ohne zusätzliche Index-Strukturen ist die Durchführung einer Query-Operation auf eine Tabelle mit einem zusammengesetzten Schlüssel. Auch hier wird ein Vergleich zwischen unverschlüsseltem und verschlüsseltem Client präsentiert.

Operation	Ø(ms)	Min(ms)	Max(ms)
Equals	64,43	58,66	177,60
Greater Equals	90,05	53,12	247,74
Greater Than	69,38	56,92	198,26
Less Equals	62,85	57,11	121,11
Less Than	150,10	57,19	320,96
Between	66,04	57,52	120,63

Tabelle 6.7: `Query`-Operation, angewandt auf unverschlüsselten Daten.

Die Ergebnisse der Ausführung der Query-Operation mithilfe des unverschlüsselten Clients sind in Tabelle 6.7 ersichtlich, ebenso wie die Ergebnisse mit dem verschlüsselten Client in Tabelle 6.8. Diese Ergebnisse werden in Abbildung 6.7 grafisch gegenübergestellt.

Operation	Aufrufe	Ø(ms)	Min(ms)	Max(ms)
Equals				
V-Query	100	277,16	158,07	496,90
AES-Enc	400	0,12	0,03	2,77
AES-Dec	9078	0,042	0,025	11,44
AWS-Client	100	67,05	52,39	192,35
OPE-Enc	100	128,82	100,94	231,78
OPE-Dec	58	131,26	101,74	250,13
Greater Equals				
V-Query	100	303,63	259,99	572,50
AES-Enc	400	0,107	0,027	0,57
AES-Dec	18560	0,03	0,025	4,35
AWS-Client	100	59,36	52,95	131,95
OPE-Enc	100	109,10	100,40	241,46
OPE-Dec	116	110,40	99,42	219,81
Greater Than				
V-Query	100	243,79	159,86	440,02
AES-Enc	400	0,12	0,03	0,98
AES-Dec	10170	0,03	0,03	1,50
AWS-Client	100	64,30	57,23	176,39
OPE-Enc	100	108,55	100,44	260,05
OPE-Dec	62	107,81	101,36	134,87
Less Equals				
V-Query	100	257,59	160,53	570,67
AES-Enc	400	0,11	0,03	0,91
AES-Dec	10244	0,03	0,03	6,44
AWS-Client	100	65,82	57,71	171,99
OPE-Enc	100	112,60	100,71	169,52
OPE-Dec	65	114,53	100,11	376,02
Less Than				
V-Query	100	178,35	151,85	292,96
AES-Enc	400	0,11	0,03	0,63
AES-Dec	995	0,03	0,03	0,42
AWS-Client	100	62,85	50,15	170,09
OPE-Enc	100	108,12	100,17	181,16
OPE-Dec	6	105,91	100,31	110,78
Between				
V-Query	100	391,12	269,08	610,88
AES-Enc	400	0,11	0,03	0,58
AES-Dec	17388	0,03	0,03	4,48
AWS-Client	100	60,41	53,70	107,62
OPE-Enc	188	109,53	99,98	235,46
OPE-Dec	109	108,29	99,38	165,77

Tabelle 6.8: Die `Query`-Operation, durchgeführt mit verschlüsselten Daten.

In Abbildung 6.7 ist erkennbar, dass sich die durchschnittlichen Ausführungszeiten bei verschlüsselten Datensätzen um ein Vielfaches erhöht haben. Ausschlaggebend für die Ausführungszeiten ist die Anzahl an retournierten Werten aus der Datenbank und die Entschlüsselung dieser.

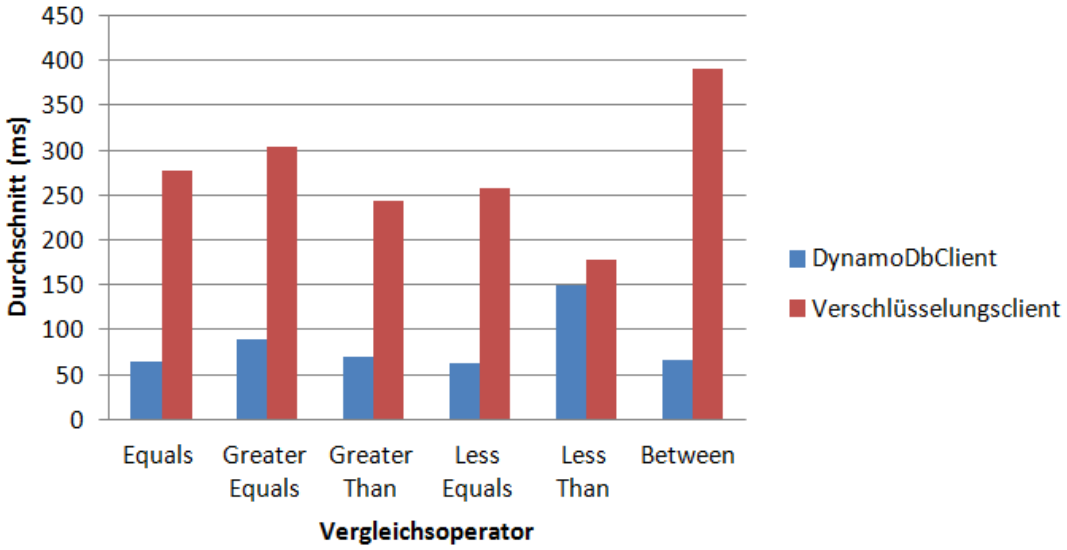

Abbildung 6.7: Diagramm zeigt einen grafischen Vergleich der unverschlüsselten und verschlüsselten Ergebnisse der `Query`-Operation (vgl. die Tabellen 6.7 und 6.8). Die Daten verwenden einen zusammengesetzten Schlüssel.

6.4.2 Testfall 2: Verschlüsselungsclient unter Einsatz des Bucketing-Index

Der zweite Testfall (siehe Abschnitt 6.1.1) untersucht den Einfluss des Bucketing-Index auf die Laufzeit des Verschlüsselungsclients. Es werden, wie im ersten Testfall, die Operationen PutItem, UpdateItem, DeleteItem und GetItem analysiert sowie der Einfluss des Bucketing-Index auf die Ausführungszeiten dieser Operationen. Es wird ein Bucketing-Index auf alle numerischen Attribute des Datenschemas (insg. sieben Attribute) erstellt. Diese Tests werden nur auf eine Tabelle mit einem einfachen Schlüsselschema durchgeführt, da hier der Fokus auf die Ausführungszeit der Scan-Operation liegt. Daher wird in diesem Testfall keine Query-Operation durchgeführt, welche eine Tabelle mit zusammengesetztem Schlüsselschema benötigt.

Die Scan-Operation wird mit folgenden Vergleichsoperationen `equals` (EQ), `less equals` (LE), `less than` (LT), `greater equals` (GE), `greater than` (GT) und `between` (BETWEEN) durchgeführt. in diesem Testfall werden ebenfalls Konjunktiv-Abfragen mit der Scan-Operation untersucht.

Die Ergebnisse dieses Testfalls werden mit jener des ersten Testlaufs (Standardbetrieb ohne Index und unverschlüsselt) verglichen.

Die nachfolgende Tabelle 6.9 beinhaltet die Ausführungszeiten der Operationen PutItem, UpdateItem, DeleteItem und GetItem unter Einfluss des Bucketing-Index. Diese Tests werden wie beim ersten Testfall ebenfalls 1000-Mal auf dieselben Daten einer Tabelle mit einfachem Schlüsselschema ausgeführt.

Operation	Aufrufe	Ø(ms)	Min(ms)	Max(ms)
putItem				
V-PutItem	1000	957,20	820,43	5.763,10
AES-Enc	171320	0,05	0,02	503,77
AES-Dec	2	4,15	0,59	7,71
AWS-Client	1000	75,90	65,11	430,13
OPE-Enc	7000	123,61	101,16	541,82
getItem				
V-GetItem	1000	864,36	609,0	1.786,54
AES-Enc	3000	0,08	0,03	0,72
AES-Dec	177299	0,04	0,03	8,07
AWS-Client	1000	64,93	59,00	213,96
OPE-Dec	6980	113,12	101,65	348,80
updateItem				
V-UpdateItem	1000	225,84	63,08	2.651,84
AES-Enc	173320	0,04	0,02	250,81
AWS-Client	1000	62,444	56,133	447,598
OPE-Enc	1383	110,15	101,83	227,28
deleteItem				
V-DeleteItem	1000	57,92	52,76	362,10
AES-Enc	3000	0,08	0,03	0,89
AWS-Client	1000	57,52	52,45	361,51

Tabelle 6.9: Datenmanipulationen und Leseoperationen, durchgeführt unter Verwendung des Bucketing-Ansatzes.

In Tabelle 6.9 ist erkennbar, dass diese Operationen sehr zeitintensiv sind. Im Vergleich zur verschlüsselten PutItem-Operation des Testfall 1 aus Tabelle 6.4 benötigt diese Operation mit dem Bucketing-Ansatz das 12-fache an Ausführungszeit. Anzumerken ist, dass dieser Testfall jeweils einen Index auf alle sieben numerischen Attribute erstellt, daher kommt es auch zu mehr OPE-Aufrufen (vor allem bei PutItem und GetItem), was die hohen Ausführungszeiten erklärt. Die Operationen UpdateItem und DeleteItem haben fast bzw. keine OPE-Aufrufe, wodurch sie entscheidend schneller sind.

Darüber hinaus muss die Ausführungshäufigkeit der AES-Entschlüsselung bei der PutItem-Operation erklärt werden. Auch hier lädt der Verschlüsselungsclient beim ersten Aufruf einer Operation Tabelleninformationen und entschlüsselt den Bezeichner des Schlüsselattributs und den Tabellennamen.

In Abbildung 6.8 ist ein grafischer Vergleich der Datenmanipulationen und Leseoperationen der Ergebnisse aus Testfall 1 und des Verschlüsselungsclients unter Einsatz des Bucketing-Index aus Testfall 2 ersichtlich. Es ist deutlich erkennbar, dass sowohl das Lesen als auch das Schreiben teure Operationen sind. Aktualisieren und Löschen sind günstigere Operationen. Dies ist jedoch wieder bedingt durch den Einsatz von OPE.

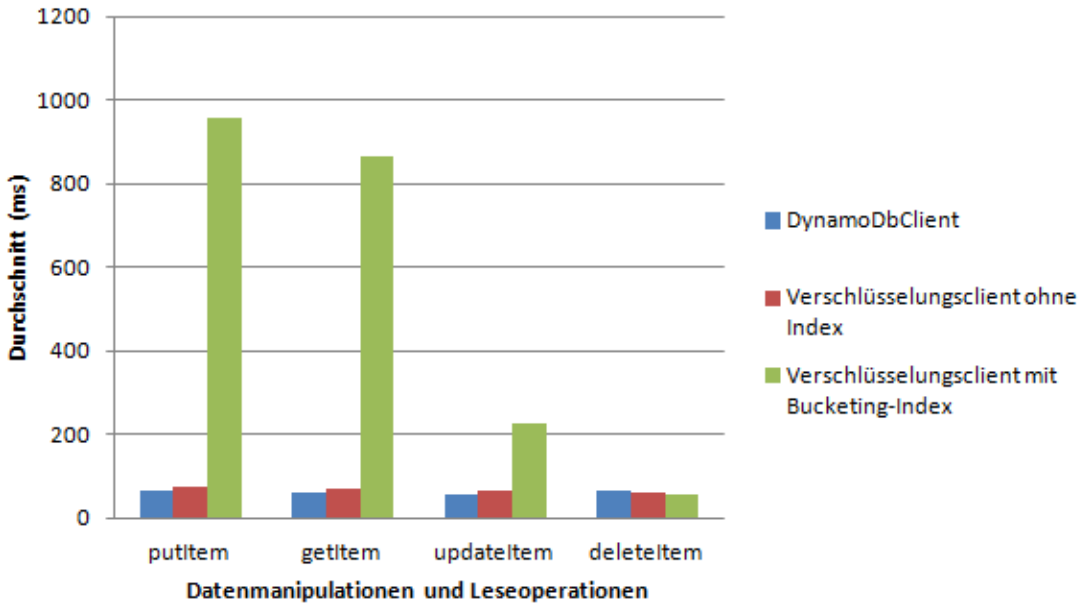

Abbildung 6.8: Vergleich der Datenmanipulationen und Leseoperationen des unverschlüsselten Client, des verschlüsselten Client ohne Scan-Funktionalität und des verschlüsselten Clients mit dem Bucketing-Index.

In diesem Testfall wird zum ersten Mal in dieser Arbeit die Scan-Operation verwendet. Daher wird die Scan-Operation mit dem unverschlüsselten Client ausgeführt (siehe Ausführungszeiten in Tabelle 6.10), um einen Vergleichswert für die Durchführungszeit zu erhalten. Anzumerken ist, dass in diesen Scan-Testfällen keine weiteren Referenzen auf zusätzliche Ergebnismengen der Scan-Operation (siehe Abschnitt 2.6.5) geladen werden. In Tabelle 6.11 werden die Ausführungszeiten mit dem Verschlüsselungsclient unter Einsatz des Bucketing-Index präsentiert. Die Scan-Operation wird in diesem Test 100-Mal ausgeführt.

Operation	Ø(ms)	Min(ms)	Max(ms)
Equals	339,88	69,20	670,84
Greater Equals	740,63	90,78	1.356,21
Greater Than	465,83	70,59	784,58
Less Equals	811,33	613,49	1.249,67
Less Than	543,28	65,14	1.325,67
Between	418,72	72,55	744,13

Tabelle 6.10: `Scan`-Operation, angewandt auf unverschlüsselte Daten.

Operation	Aufrufe	Ø(ms)	Min (ms)	Max (ms)
Equals				
V-Scan	100	11.944,07	237,90	24.326,26
AES-Enc	300	0,15	0,08	0,85
AES-Dec	834480	0,035	0,025	14,56
AWS-Client	100	400,63	65,51	864,16
OPE-Enc	100	110,88	106,91	124,42
OPE-Dec	10112	108,88	101,80	263,94
OPE-Cache-Read	33181	0,002	0,00	0,23
Greater Equals				
V-Scan	100	27.948,80	2.146,69	46.548,55
AES-Enc	300	0,14	0,09	0,43
AES-Dec	1916324	0,04	0,03	113,36
AWS-Client	100	914,00	90,25	1.673,78
OPE-Enc	100	111,75	107,20	133,42
OPE-Dec	23997	108,53	101,64	256,78
OPE-Cache-Read	23997	0,01	0,00	0,43
Greater Than				
V-Scan	100	18.366,06	248,46	27.062,84
AES-Enc	300	0,15	0,09	1,35
AES-Dec	1081844	0,04	0,03	56,94
AWS-Client	100	551,66	68,09	1.156,30
OPE-Enc	100	111,61	106,81	130,54
OPE-Dec	15614	109,96	101,98	274,65
OPE-Cache-Read	42385	0,00	0,00	0,33
Less Equals				
V-Scan	100	30.905,72	23.450,03	46.313,71
AES-Enc	300	0,15	0,09	0,80
AES-Dec	2224856	0,03	0,03	46,20
AWS-Client	100	1.065,47	770,21	2.354,20
OPE-Enc	100	111,28	106,55	124,86
OPE-Dec	26501	108,39	101,66	264,95
OPE-Cache-Read	88115	0,00	0,00	0,31
Less Than				
V-Scan	100	19.630,99	238,87	45.299,18
AES-Enc	300	0,15	0,09	0,56
AES-Dec	1390376	0,04	0,03	29,80
AWS-Client	100	689,77	67,46	1.542,57
OPE-Enc	100	111,97	107,13	183,13
OPE-Dec	16751	108,58	101,52	249,06
OPE-Cache-Read	54934	0,00	0,00	0,29
Between				
V-Scan	100	14.560,56	1.131,72	30.283,12
AES-Enc	300	0,17	0,09	0,83
AES-Dec	961398	0,04	0,03	18,62
AWS-API	100	498,97	75,76	1.194,57
OPE-Enc	189	114,06	107,54	187,35
OPE-Dec	11944	111,84	101,90	327,46
OPE-Cache-Read	38137	0,00	0,00	2,89

Tabelle 6.11: Die Bereichsabfragen durchgeführt mithilfe der `Scan`-Operation mithilfe des Bucketing-Index.

In der Tabelle 6.11 ist deutlich erkennbar, dass diese Operation sehr zeitintensiv ist. Nichtsdestotrotz ist ersichtlich, dass der Einsatz des OPE-Cache hier zugunsten der Ausführungszeit wirkt. Dies wird anhand der Anzahl der Aufrufe innerhalb der Spalte **OPE-Cache-Read** gezeigt.

In Abbildung 6.9 werden die Ausführungszeiten der unterschiedlichen Vergleichsoperatoren der `Scan`-Operation des unverschlüsselten DynamoDBClients und des Verschlüsselungsclients gezeigt. Die Ausführungszeit steigt beim Einsatz der verschlüsselten Scan-Operation. An dieser Stelle sind auch die unterschiedlichen Häufigkeiten der Entschlüsselungen anzumerken. Die DynamoDB beschränkt ihre Ausführungen und ihre Filterungen auf eine Menge von 1 MB an Daten. Durch die Verschlüsselung der Daten wird diese Grenze schneller erreicht. Es werden im Vergleich zum unverschlüsselten Client weniger Ergebnisse zurückgeliefert und demnach auch weniger entschlüsselt.

An dieser Stelle muss man anmerken, dass die hier präsentierten Laufzeiten abhängig sind von der verwendeten Datenmenge. In dieser Arbeit werden Scan-Operationen mithilfe von Index-Implementierung auf ihre Laufzeit untersucht. Dennoch kann es sein, dass ab einer gewissen Datenmenge die verschlüsselten Scan-Operationen mithilfe der Index-Strukturen schneller sein können als jene unverschlüsselter Scan-Operationen.

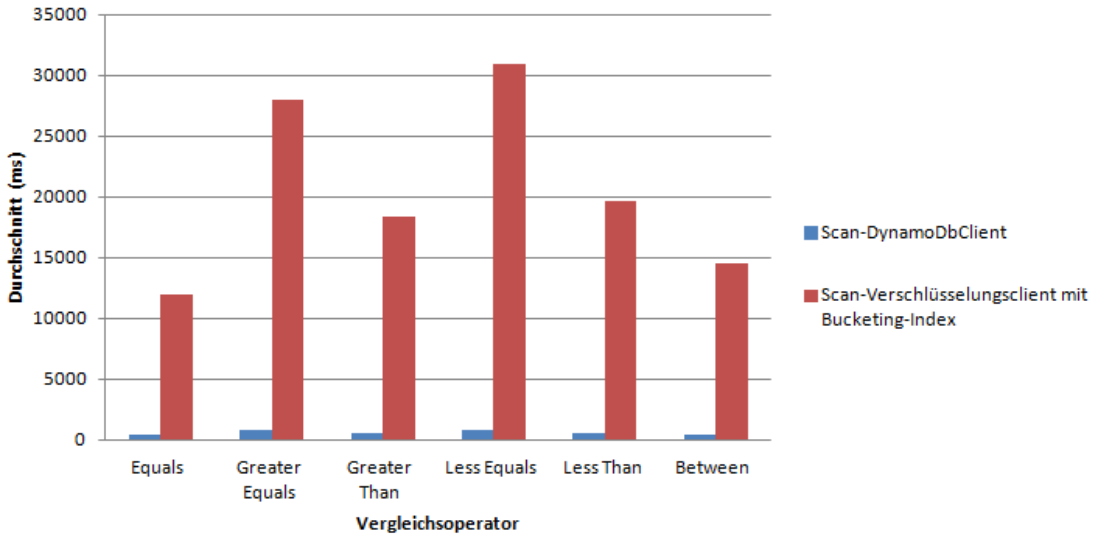

Abbildung 6.9: Das Diagramm zeigt einen Ergebnisvergleich der unverschlüsselten und der verschlüsselten `Scan`-Operation unter Einsatz des Bucketing-Index.

Weiters erfolgt in diesem Testfall die Durchführung der Scan-Operation mit Konjunktiv-Abfragen. Es werden hierbei die Vergleichsoperationen EQ und BETWEEN mit dem logischen And-Operator verknüpft und auf numerische Attribute durchgeführt. Die Tests für Konjunktiv-Abfragen beginnen bei einem Abfragekriterium (einfache Abfrage) und verknüpfen pro Testablauf ein zusätzliches Abfragekriterium damit. Dies wird so lange durchgeführt, bis alle numerischen Attribute des Datensatzes verwendet werden (der

hier verwendete Datensatz besitzt sieben numerische Attribute). Auch diese Tests werden 100-Mal durchgeführt. Sie werden ebenfalls mit dem unverschlüsselten Client wie auch mit dem Verschlüsselungsclient durchgeführt und anschließend verglichen. In der Tabelle 6.12 werden die Ausführungszeiten des unverschlüsselten Clients und in Tabelle 6.13 jene des Verschlüsselungsclients dargestellt. Des Weiteren wird in Tabelle 6.12 in der Spalte **# num Werte** gezeigt, wie viele numerische Werte beim aktuellen Testlauf miteinander verknüpft worden sind. Die gewählten Vergleichsbedingungen (EQ und BETWEEN) sind für alle Tests dieser Art gleich.

Scan-Operator	**# num Werte**	**Ø(ms)**	**Min(ms)**	**Max(ms)**
EQ	1	420,49	75,42	2.025,77
	2	821,56	66,49	3.938,35
	3	967,38	65,50	4.032,52
	4	970,40	64,66	3.773,27
	5	927,34	71,30	3.638,77
	6	969,07	66,30	3.738,22
	7	949,50	64,77	3.729,60
Between	1	1.024,50	81,43	4.236,14
	2	954,68	74,56	3.831,66
	3	949,90	71,66	4.040,16
	4	986,19	71,96	4.025,97
	5	963,79	65,01	3.750,07
	6	829,84	66,80	10.563,14
	7	1.005,04	66,93	4.058,01

Tabelle 6.12: Numerische Konjunktiv-Abfragen, durchgeführt mithilfe der unverschlüsselten `Scan`-Operation.

Scan-Operator	**# num Werte**	**Ø(ms)**	**Min (ms)**	**Max (ms)**
Equals	1	12.828,60	249,47	30.796,64
	2	4.955,70	354,56	14.535,67
	3	5.069,39	462,26	14.914,21
	4	3.124,23	555,66	7.738,63
	5	813,42	658,05	1.689,20
	6	1.034,52	783,02	2.509,15
	7	1.095,40	898,27	2.427,09
Between	1	16.295,59	1.120,60	41.343,49
	2	6.422,08	340,00	18.748,84
	3	7.038,89	467,21	23.601,04
	4	5.638,90	792,37	12.871,33
	5	1.980,79	1.006,28	5.754,38
	6	1.467,96	1.118,33	2.385,29
	7	1.666,94	1.232,78	2.484,51

Tabelle 6.13: Konjunktions -Abfragen mithilfe der `Scan`-Operation unter Verwendung der Bucketing-Technik.

In den Abbildungen 6.10 und 6.11 werden die verschlüsselten Ergebnisse aus Tabelle 6.13 den unverschlüsselten aus Tabelle 6.12 grafisch gegenübergestellt. Es werden die Durchschnittswerte in Millisekunden angezeigt. Erkennbar ist, dass die Abfragen mit weniger Bedingungen langsamer sind als jene mit mehreren Bedingungen. Die Erklärung kann aus der Menge an retournierten Ergebnissen abgeleitet werden. Diese sinkt mit steigender Anzahl an Bedingungen.

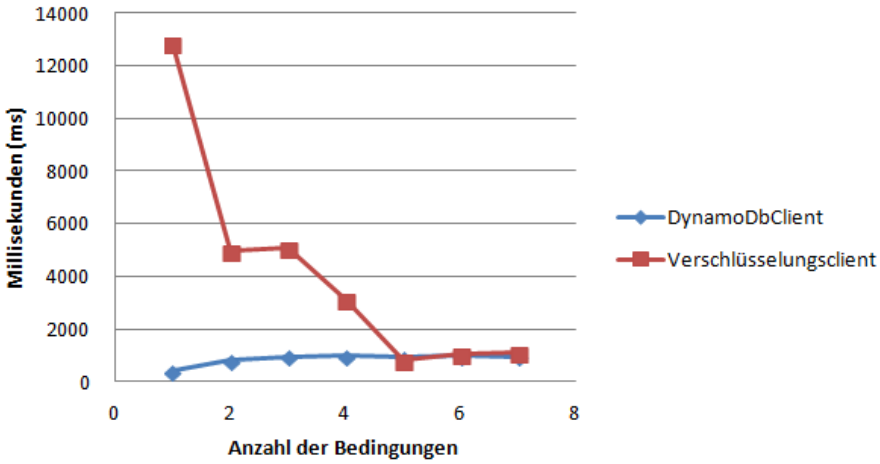

Abbildung 6.10: Vergleich der unverschlüsselten und verschlüsselten Konjunktiv-Abfragen (`Equals`) mithilfe der Bucketing-`Scan`-Operation (vgl. die Tabellen 6.12 und 6.13).

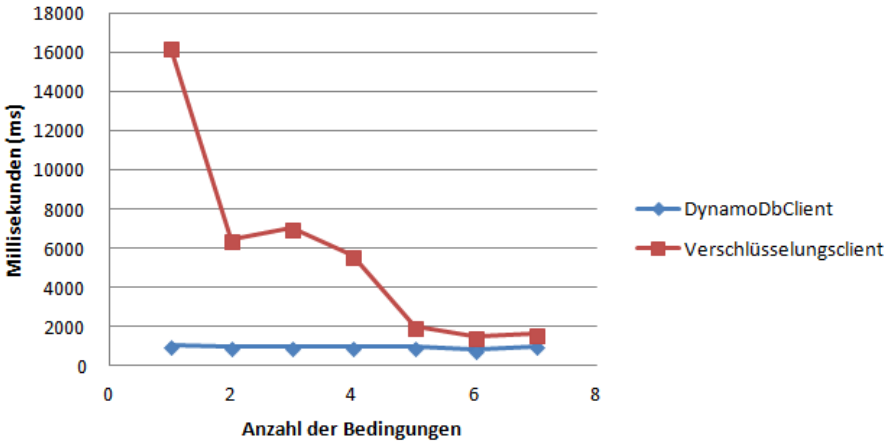

Abbildung 6.11: Das Diagramm zeigt einen Vergleich der unverschlüsselten und verschlüsselten Konjunktiv-Abfragen (`Between`) mithilfe der Bucketing-`Scan`-Operation (vgl. die Tabellen 6.11 und 6.10).

6.4.3 Testfall 3: Verschlüsselungsclient unter Einsatz des invertierten Index

Testfall drei (siehe Abschnitt 6.1.1) wiederholt die Abläufe des zweiten Tests unter Einsatz des verschlüsselten invertierten Index. Hier werden die Ergebnisse der Datenbankoperationen mit dem unverschlüsselten und verschlüsselten Client verglichen.

Die Ausführungszeiten der Operationen PutItem, UpdateItem, DeleteItem und GetItem unter Einfluss des invertierten Index sind in Tabelle 6.14 ersichtlich. Diese Index-Implementierung ist komplexer als jene des Bucketing-Index, da diese auf eigenen Index-Tabellen durch den BackgroundWorker (siehe Abschnitt 5.5.2) verwaltet wird. Da dieser Testfall ebenfalls einen Index auf alle sieben numerischen Attribute erstellt, werden insg. sieben Index-Tabellen verwendet. Die Ergebnisse der Schreibeoperationen in Tabelle 6.14 sind wie folgt zu lesen:

Der beschriebene Operationsname z.B. `AWS-PutItem` steht im Bezug zur Zeile `V-PutItem`. AWS-* oder V-* ist jene Ausführungszeit des Verschlüsselungsclients, die er benötigt, bis er retourniert. Diese Ausführungszeit beinhaltet das Schreiben des Datensatz in die Primärdatentabelle und das Schreiben der Log-Recovery-Datei. Die Ausführungszeiten der Indexoperationen (durch den Backgroundworker) sind wie folgt zu verstehen:

Index-Operation - Die Index-Implementierung des invertierten Index wird mithilfe eines *Backgroundworkers* durchgeführt (vgl. Abschnitt 5.5.2). Um diese Operationen messen zu können, wird jede Indexoperation einzeln gemessen (z. B. `Index-Update` oder `Index-Delete`), da in diesen Operationen auch die Logik implementiert ist.

Index-Gesamt-Operation - Die gesamte Ausführungszeit der Index-Operation (z. B. `Index-Gesamt-PutItem`) inkludiert das Lesen der Log-Datei und die Verarbeitung dieser am Index (löschen, aktualisieren, etc.).

Operation	Aufrufe	Ø(ms)	Min (ms)	Max (ms)
putItem				
V-PutItem	1000	220,79	163,33	1.980,89
V-GetItem	1000	85,95	64,41	9.326,89
AES-Enc	251320	0,04	0,024	192,07
AES-Dec	184321	0,05	0,025	150,23
AWS-PutItem	1000	77,92	63,09	326,33
AWS-GetItem	1000	77,68	58,99	9.315,31
OPE-Enc	2409	163,24	101,38	1.246,80
OPE-Cache-Read	14000	0,004	0,00	0,66
Index-DeleteItem	7000	62,05	56,09	326,40
Index-UpdateItem	7000	69,53	57,28	42.908,72
Index-Gesamt-PutItem	1000	1.453,58	893,14	44.289,86
getItem				
V-GetItem	1000	62,32	57,63	163,65
AES-Enc	3000	0,08	0,03	0,56
AES-Dec	163319	0,03	0,03	6,03
AWS-GetItem	1000	56,99	52,82	157,65
updateItem				
V-UpdateItem	1000	218,16	149,50	424,93
V-GetItem	1000	77,70	57,45	253,20
AES-Enc	253319	0,04	0,02	146,50
AES-Dec	184319	0,05	0,03	182,86
AWS-UpdateItem	1000	86,47	55,68	252,86
AWS-GetItem	1000	70,21	52,64	220,92
OPE-Enc	1383	129,98	101,53	914,43
OPE-Cache-Read	14000	0,004	0,001	0,195
Index-DeleteItem	7000	60,48	51,66	283,46
Index-UpdateItem	7000	60,23	51,46	268,58
Index-Gesamt-UpdateItem	1000	1.147,90	822,13	3.582,74
deleteItem				
V-DeleteItem	1000	230,09	165,42	692,47
AES-Enc	45000	0,05	0,03	9,85
AES-Dec	170319	0,04	0,02	152,45
AWS-DeleteItem	1000	82,28	54,23	333,41
OPE-Enc	1383	155,96	101,58	1.919,68
Index-DeleteItem	7000	69,59	51,91	368,48
Index-Gesamt-DeleteItem	1000	726,29	385,05	4.041,62

Tabelle 6.14: Die Datenmanipulationen und Leseoperationen mit dem Verschlüsselungsclient bei Verwendung des invertierten Index auf einer Tabelle mit einem einfachen Schlüssel.

Anhand der Daten aus Tabelle 6.14 ist eindeutig erkennbar, dass diese Technik komplexer ist als jene des Bucketing-Index und auch mehr Zeit benötigt. Es werden hier die Methoden des Verschlüsselungsclients (V-GetItem, V-PutItem und V-DeleteItem) zur Wartung und Erstellung des Index eingesetzt. Die Laufzeit der GetItem-Operation selbst ist vergleichbar mit den Laufzeitwerten aus dem ersten Testfall, da hier keine Index-Struktur benötigt wird und kein zusätzlicher Aufwand (wie beim Bucketing-Index) zur Entschlüsselung des Bucketing-Index innerhalb des Datensatzes notwendig ist.

In Abbildung 6.12 werden diese Daten mit den zuvor beschriebenen Datenmanipulations- und Leseoperations-Tests verglichen. In diesem Diagramm sind für den Index zwei Balken zum Vergleich eingetragen. Der Balken **Verschlüsselungsclient mit invertiertem Index** umfasst die jeweilige Operation der Verschlüsselungsebene (Operation auf Tabelle und das Schreiben der Log-Recovery-Daten) und der Balken **Verschlüsselungsclient mit invertiertem Index Backgroundworker** umfasst die gesamten Operationen, welche vom Backgroundworker ausgeführt werden, um den Index zu verwalten.

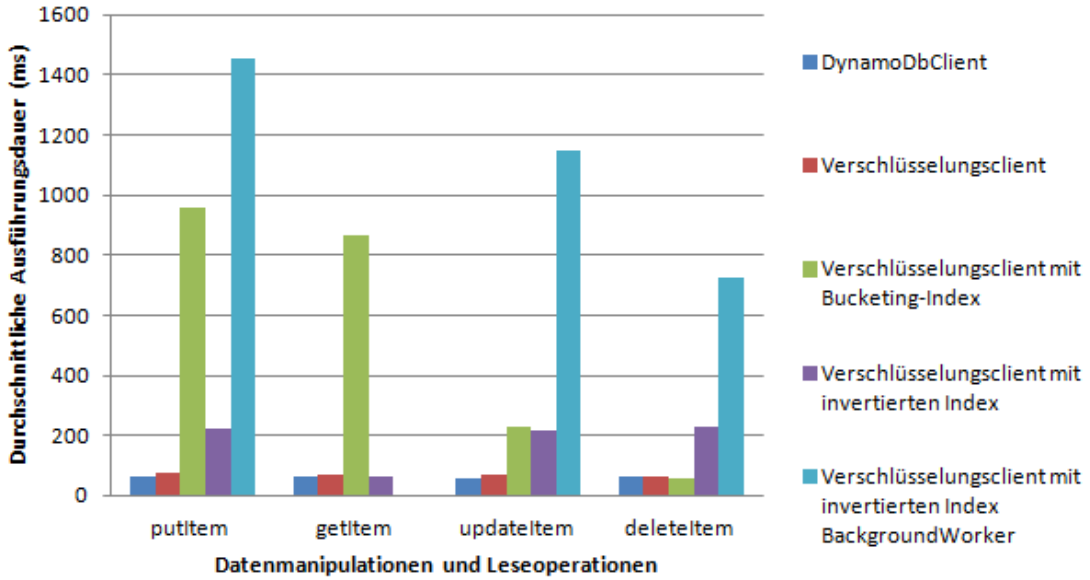

Abbildung 6.12: Die Abbildung zeigt einen Vergleich der verschiedenen Tests der Datenmanipulationen und Leseoperationen (vgl. Tabelle 6.14).

Wie in Testfall zwei wird auch hier die Scan-Operation unter Einsatz des invertierten Index 100-Mal ausgeführt. Die Tabelle 6.15 zeigt die Ausführungszeiten dieses Tests. Die gesamte Laufzeit aller hier aufgeführten Operationen ist in der jeweiligen Zeile von **V-Scan** ersichtlich. Die `V-BatchGetItem`-Operation wird von der Index-Operation `ReadTable` (vgl. Zeile **Index-ReadTable**) verwendet, um die referenzierten Datensätze aus der Tabelle zu laden und zu filtern (vgl. Abschnitt 5.5.2 für den Ablauf der invertierten Index-Lösung). Die Zeilen **Index-Scan** beinhalten die Ausführungszeiten aller durchgeführten `Scan`-Aufrufe des AWS-DynamoDBClient auf die Index-Tabellen. Die Zeilen **Index-ProcessScanResult** wiederum stellen die Extraktion der binär gespeicherten Referenzen aus den Ergebnissen der **Index-Scan**-Abfragen auf die Index-Tabelle dar.

Operation	Aufrufe	Ø(ms)	Min (ms)	Max (ms)
Equals				
V-Scan	100	3.441,30	461,64	28.532,09
V-BatchGetItem	100	867,01	140,50	2.170,06
AES-Enc	11969	0,03	0,03	6,69
AES-Dec	1045959	0,03	0,03	36,32
OPE-Enc	100	108,39	102,65	238,55
OPE-Dec	100	108,38	102,17	150,88
OPE-Cache-Read	25004	0,00	0,00	0,50
Index-Scan	100	2.201,27	59,57	26.538,72
Index-ProcessScanResult	100	205,70	103,49	965,63
Index-ReadTable	100	869,45	140,58	2.173,77
Greater Equals				
V-Scan	100	6.983,57	1.100,96	34.574,33
V-BatchGetItem	100	1.390,48	578,30	2.227,10
AES-Enc	19284	0,03	0,03	8,91
AES-Dec	1825309	0,03	0,03	20,85
OPE-Enc	100	108,36	103,19	132,10
OPE-Dec	2981	106,71	102,04	263,31
OPE-Cache-Read	58564	0,00	0,00	0,14
Index-Scan	100	2.043,17	81,24	27.068,61
Index-ProcessScanResult	100	3.372,61	219,15	6.160,19
Index-ReadTable	100	1.394,70	578,53	2.229,96
Greater Than				
V-Scan	100	6.343,75	561,51	35.686,45
V-BatchGetItem	100	1.404,00	185,44	11.419,43
AES-Enc	18282	0,03	0,03	42,97
AES-Dec	1630623	0,03	0,03	21,23
OPE-Enc	100	109,69	102,53	164,69
OPE-Dec	2881	109,07	101,91	344,67
OPE-Cache-Read	33560	0,00	0,00	0,38
Index-Scan	100	1.503,73	67,30	27.038,67
Index-ProcessScanResult	100	3.260,01	105,47	10.360,47
Index-ReadTable	100	1.406,47	185,53	11.422,76
Less Equals				
V-Scan	100	6.669,56	2.321,64	34.320,48
V-BatchGetItem	100	1.446,01	1.389,43	1.845,54
AES-Enc	20600	0,03	0,03	8,66
AES-Dec	1954084	0,03	0,03	18,40
OPE-Enc	100	107,29	102,34	114,26
OPE-Dec	2219	105,67	101,90	160,12
OPE-Cache-Read	66440	0,00	0,00	0,26
Index-Scan	100	2.492,82	440,19	27.095,72
Index-ProcessScanResult	100	2.553,50	253,11	5.656,46
Index-ReadTable	100	1.450,87	1.392,92	1.848,85
Less Than				
V-Scan	100	6.335,94	233,41	26.437,69
V-BatchGetItem	100	795,62	0,00	2.223,70
AES-Enc	11354	0,03	0,03	0,74
AES-Dec	1086078	0,03	0,03	27,48
OPE-Enc	100	108,44	102,36	172,21
OPE-Dec	2119	108,28	101,70	264,86
OPE-Cache-Read	41436	0,00	0,00	0,16
Index-Scan	100	2.925,11	65,23	26.262,40
Index-ProcessScanResult	100	2.436,23	0,10	7.417,61
Index-ReadTable	100	798,76	0,01	2.239,54
Between				
V-Scan	100	6.318,64	856,42	28.744,562
V-BatchGetItem	100	1.133,23	169,83	1.478,22
AES-Enc	17214	0,03	0,03	6,67
AES-Dec	1498143	0,03	0,03	42,80
OPE-Enc	190	106,16	102,56	114,77
OPE-Dec	306	105,86	102,47	129,10
OPE-Cache-Read	28530	0,00	0,00	0,06
Index-Scan	100	4.512,47	73,25	26.540,78
Index-ProcessScanResult	100	411,60	113,99	1.009,32
Index-ReadTable	100	1.135,42	169,93	1.479,32

Tabelle 6.15: Die Bereichsabfragen, ausgeführt mithilfe der `Scan`-Operation basierend auf dem invertierten Index.

Diese Scan-Operation durchsucht die Tabellen solange, bis ein Limit von 1 MB oder 100 durchsuchten Datensätzen gefiltert wurde (vgl. Abschnitt 2.6.5). Dadurch, dass in den Index-Tabellen nur Referenzen auf die Originaldaten gespeichert werden, findet die Lösung mehr Ergebnisse in den Index-Tabellen (bis das Ergebnismengenlimit von 1 MB erreicht wird, siehe Abschnitt 2.6.5), was wiederum die hohe Zahl an Entschlüsselungen bei AES (siehe Tabelle 6.15) im Vergleich zum Bucketing-Index (vgl. Tabelle 6.11) erklärt. Die meiste Ausführungszeit wird von den Index-Operationen (**Scan** und **ReadTable**) benötigt. Die hohe Ausführungszeit der Index-Operationen **ReadTable** resultiert in den **V-BatchGetItem**-Aufrufen des Verschlüsselungsclients. Mithilfe dieser Operation werden die Daten der Referenztabelle geladen und entschlüsselt. Sie retourniert maximal 100-Datensätze oder 1 MB an Daten aus der Datenbank. Des Weiteren ist anhand der Tabelle 6.15 auch die Bedeutung des OPE-Caches deutlich erkennbar. Dieser wird sehr oft aufgerufen. Ohne ihn würde die Ausführungszeit von OPE drastisch steigen.

In Abbildung 6.13 werden die Ergebnisse dieses Testlaufs mit jenen des unverschlüsselten und des Bucketing-Index verglichen. Hier werden ebenfalls Durchschnittswerte in Millisekunden gegenübergestellt. Es ist deutlich erkennbar, dass der invertierte Index schneller ist als der Bucketing-Index.

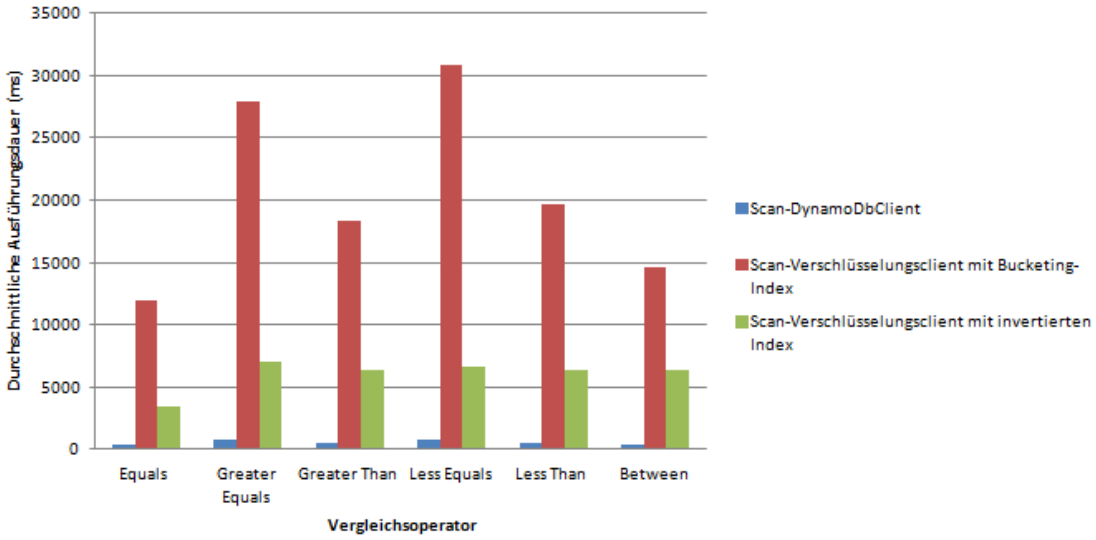

Abbildung 6.13: Vergleich der verschiedenen Index-Implementierungen je nach verwendetem Vergleichsoperator bei `Scan`-Operationen.

Der letzte Testlauf dieses Testfalls ist die Durchführung von Konjunktiv-Abfragen mithilfe der Scan-Operation. Hier wird der gleiche Ablauf wie bei Testfall zwei herangezogen. Die Ausführungszeiten sind in Tabelle 6.16 ersichtlich. In der Spalte **# num Werte** befindet sich die Anzahl an verknüpften Bedingungen pro Vergleichsoperation. Diese Bedingungen werden ebenfalls mit einem logischen `AND` verknüpft. Eine Zeile zeigt die gesamte Ausführungszeit der Operation mithilfe des Verschlüsselungsclients.

Scan-Operator	# num Werte	Ø(ms)	Min (ms)	Max (ms)
Equals	1	6.633,41	472,15	28.405,86
	2	6.407,41	832,73	29.746,97
	3	6.610,12	1.068,87	30.183,47
	4	6.584,48	2.912,07	30.937,83
	5	4.474,37	1.053,18	31.492,80
	6	6.362,73	1.464,94	29.689,13
	7	6.559,13	2.098,49	32.413,12
Between	1	3.876,93	819,51	28.673,14
	2	6.562,26	1.605,97	29.591,65
	3	5.662,74	810,28	29.449,61
	4	5.680,06	1.824,03	13.011,85
	5	5.926,46	2.009,55	20.268,94
	6	6.483,38	2.403,89	18.736,37
	7	6.892,72	2.502,73	11.610,72

Tabelle 6.16: Konjunktiv-Abfragen, durchgeführt mithilfe der `Scan`-Operation unter Einsatz des invertierten Index.

In den Abbildungen 6.14 und 6.15 ist deutlich erkennbar, dass die Ausführungszeit des invertierten Index im Vergleich zum Bucketing-Index bzw. zum unverschlüsselten Ansatz sehr hoch ist. Dies ist dadurch bedingt, dass die Indexlösung mehr Ergebnisse findet, da weniger große Datensätze in den Indextabellen durchsucht werden und somit mehr Entschlüsselungen durchgeführt werden müssen. Dies erhöht die Ausführungszeit.

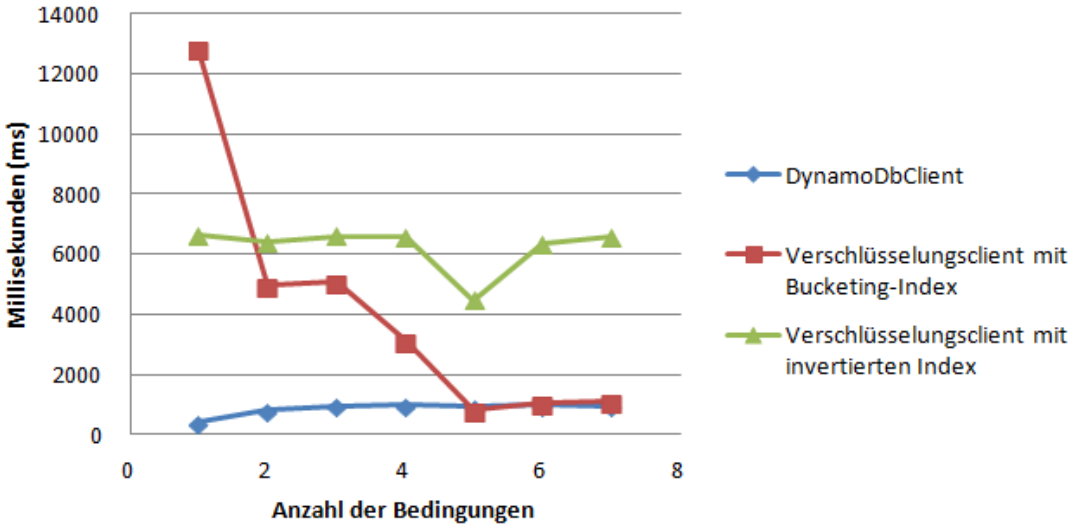

Abbildung 6.14: Das Diagramm zeigt einen Vergleich der Konjunktiv-Abfragen (`Equals`) durch die `Scan`-Operation invertierter Index (vgl. die Tabellen 6.12, 6.13 und 6.16).

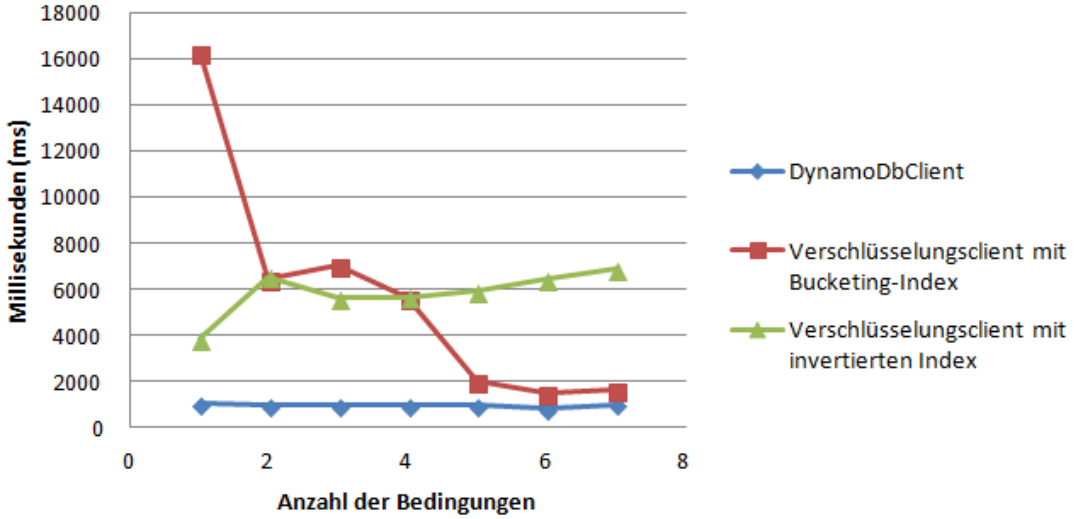

Abbildung 6.15: Das Diagramm zeigt einen Vergleich der Konjunktiv-Abfragen (`Between`) durch die `Scan`-Operation invertierter Index (vgl. die Tabellen 6.12, 6.13 und 6.16).

6.4.4 Fazit

Die Ergebnisse des ersten Testfalls zeigen, dass die zellenbasierte Verschlüsselung der Daten realisierbar ist, jedoch ist dies mit entsprechenden Kosten verbunden (Testfall 1: vgl. Tabellen 6.4, 6.6). Zusätzlich kann die Query-Operation ohne weitere Index-Strukturen mithilfe des Verschlüsselungsclients realisiert werden. Um diese Abfrageart realisieren zu können, müssen verschlüsselte Abfragen auf Schlüsselattribute unterstützt werden. Der Verschlüsselungsclient verwendet daher immer OPE für numerische Schlüsselattribute. Der Einsatz von OPE erhöht jedoch die Laufzeit. Dies wird im ersten Testlauf in den Laufzeitergebnissen der Datenbankoperationen in Tabelle 6.6 und den Durchführungszeiten der Query-Operation in Tabelle 6.8 ersichtlich. Die Ausführungszeit für den OPE-Algorithmus ist sehr hoch.

Das zur Verfügung stellen der `Scan`-Funktionalität ist im Vergleich zum HDSE-Ansatz noch kostenintensiver (Testfall zwei und drei). Die Testergebnisse des Verschlüsselungsclients unter Einsatz eines Bucketing-Index ergeben, dass die Kosten für Datenmanipulationen im Vergleich zur invertierten Index-Lösung geringer sind (vgl. Abbildung 6.12). Die Ausführungszeiten der `Scan`-Operation mit der invertierten Index-Lösung sind im Vergleich zum Bucketing-Ansatz besser (vgl. Abbildung 6.13).

Die Durchführung von Konjunktiv-Abfragen ist mithilfe des Verschlüsselungsclients möglich. Jedoch ist diese unverschlüsselt wie auch verschlüsselt sehr zeitaufwendig.

Zusammenfassend bedeutet dies:

1. Der Großteil der DynamoDbClient Funktionalität wird ohne die Notwendigkeit, Indizes anlegen und pflegen zu müssen, vom HDSE unterstützt, bedingt durch das

Datenmodell der Datenbank.

2. Die Verschlüsselung durch den HDSE erhöht die Ausführungszeiten von Datenbankoperationen.
3. Die `Scan`-Operation ist im Vergleich zu anderen Abfrageoperationen sehr zeitintensiv. Dies trifft auf alle getesteten Szenarien zu.
4. Der Bucketing-Ansatz ermöglicht effizienteres Schreiben, Aktualisieren und Löschen der Datensätze als der invertierte Index.
5. Der Bucketing-Ansatz weicht die Garantien des HDSE-Ansatzes etwas auf, indem er Informationen über den Dateninhalt preisgibt (spaltenbasierte Verschlüsselung innerhalb der Datensätze einer Tabelle).
6. Der Verwaltungsaufwand des invertierten Index ist sehr hoch. Jede Schreiboperation muss propagiert zu den Index-Tabellen propagiert werden.
7. Der invertierte Index ermöglicht schnellere Leseoperationen als der Bucketing-Ansatz. Abfragen wie `Scan, GetItem` und `Query` sind im Vergleich zum Bucketing-Ansatz sehr schnell. Die Operationen `GetItem` und `Query` können, ohne Index, direkt auf die Primärdaten durchgeführt werden.
8. Konjunktiv-Abfragen sind möglich, sind jedoch sehr zeitintensiv.
9. Der OPE-Algorithmus ist sehr zeitaufwendig. Der durchgehende und bewusste Einsatz des Caches und dieser Verschlüsselung (durch selektive Auswahl der abzufragenden Attribute) ist zu empfehlen.

Kapitel 7

Zusammenfassung

In dieser Arbeit wurde ein Verschlüsselungsclient für die cloud-basierte NoSQL-Datenbank namens Amazon DynamoDB entwickelt. Die Aufgabe des Clients ist es, die Daten verschlüsselt auf den Server zu übertragen bzw. Abfragen so umzuformulieren (Query-Rewriting), dass die bestehende Datenbankfunktionalität verwendet werden kann. Beim implementierten zellenbasierten clientseitigen Verschlüsselungsansatz handelt es sich um die *Hierarchically Derived Symmetric Encryption* (HDSE). Dieser Ansatz erstellt, basierend auf dem gegebenen Datenmodell, die Verschlüsselungsschlüssel abhängig von der Verschachtelungstiefe der Daten. Wichtige Bestandteile des resultierenden Verschlüsselungsschlüssels sind ein clientseitig geheimer Initialisierungsvektor und der Primärschlüssel. Dadurch ermöglicht dieser Ansatz unterschiedliche Verschlüsselungsschlüssel pro Datenwert. Der Nachteil der zellenbasierten Verschlüsselung ist die Einschränkung der Abfragefunktionalität.

Es wurden in dieser Arbeit zudem diverse Verschlüsselungsansätze zur Unterstützung von Datenbankabfragen untersucht. Hierbei handelt es sich großteils um verschiedene verschlüsselte Indizes und den Einsatz von homomorphen Verschlüsselungsalgorithmen, wodurch Bereichsabfragen realisiert werden können.

Durch den Verschlüsselungsansatz HDSE und das Datenmodell der DynamoDB kann der Großteil der Abfragefunktionalitäten unterstützt werden.

Zur Unterstützung von serverseitigen Bereichsabfragen wird auf numerische Attribute eine Order-Presever-Encryption [23] Verschlüsselung angewendet. Dies erlaubt unter Einsatz desgleichen Verschlüsselungsschlüssels Vergleiche auf numerische Attribute.

Die Scan-Operation und die Unterstützung von Bereichsabfragen sind nicht durch den HDSE unterstützt und benötigen daher eine zusätzliche Implementierung wie einen verschlüsselten Index. Die Scan-Operation erlaubt Abfragen auf alle Attribute einer Tabelle sowie die Kombination von beliebig vielen Attributen (Konjunktiv-Abfrage) der DynamoDB. Um diese auch weiterhin zu unterstützen, wurden in dieser Arbeit zwei Index-Implementierungen (Bucketing-Index [36] und invertierter Index [26]) umgesetzt.

Beim Bucketing-Index wird ein grob verschlüsselter Index innerhalb einer Datentabelle von Dynamo erstellt. Damit die Werte des Bucketing-Index vergleichbar sind, wird hier eine spaltenbasierte Verschlüsselung auf die Index-Werte eingesetzt.

Die zweite Index-Implementierung erstellt pro indiziertem Attribut eine eigene Index-Tabelle. Dies wiederum zieht zusätzlichen Mehraufwand bei der Selbstverwaltung der Index-Tabellen nach sich und macht eine Schreibeoperation der Daten sehr kostspielig.

Um die Scan-Funktionalität der Konjunktiv-Abfragen ebenfalls zu unterstützen, werden die Ergebnismengen clientseitig gefiltert. Der Client lädt die entsprechenden Ergebnisse (durch serverseitige Filterung) und sortiert diese auf Basis der Nutzeranfrage.

Im letzten Abschnitt dieser Arbeit werden Laufzeittests durchgeführt, die die Laufzeit des Verschlüsselungsclients und die Kosten der Verschlüsselung zeigen. Diese Tests zeigen, dass die Verschlüsselung der Daten mithilfe des HDSE realistisch anwendbar ist. Auf der Amazon DynamoDB-Datenbank kann der Großteil der Datenbankfunktionalitäten auch weiterhin trotz der zellenbasierten Verschlüsselung unterstützt werden.

Jedoch erhöht die mehrheitliche Unterstützung der Datenbankfunktionalität mithilfe der Index-Strukturen die Ausführungszeiten drastisch. Daher sollte der Einsatz dieser Strukturen mit Bedacht gewählt werden. Beide Index-Strukturen haben unterschiedliche Vor- und Nachteile. Beim Bucketing-Index sind Schreiboperationen kostengünstiger als beim invertierten Index. Jedoch sind beim invertierten Index Leseoperationen entscheidend günstiger.

Mögliche zukünftige Schritte wären die Unterstützung von Sekundärindizes der Amazon DynamoDB in der Beispielimplementierung, die Unterstützung von Operationen auf verschlüsselte Zeichenketten (z. B. `CONTAINS` auf Zeichenketten) sowie die Verbesserung der Ausführungszeit der in dieser Arbeit beschriebenen Beispielimplementierung.

Zusammenfassend bedeutet dies, dass mithilfe des HDSE-Verschlüsselungsansatzes der Großteil der Datenbankfunktionalitäten trotz Datenbankverschlüsselung auch weiterhin unterstützt werden kann. Die Ausführungszeiten der Datenbankoperationen und Abfragen sind jedoch stark abhängig von den verwendeten Funktionalitäten (Index-Strukturen).

Quellenverzeichnis

Literatur

[1] Martin L. Abbott und Michael T. Fisher. *Scalability rules: 50 principles for scaling Web sites*. Upper Saddle River, NJ: Addison-Wesley, 2011 (siehe S. 6–9).

[2] Rakesh Agrawal u. a. „Order preserving encryption for numeric data“. In: *the 2004 ACM SIGMOD international conference*. Hrsg. von Patrick Valduriez u. a., S. 563 (siehe S. 37).

[3] J. Chris Anderson, Jan Lehnardt und Noah Slater. *CouchDB: The definitive guide*. 1st ed. Beijing und Cambridge [Mass.]: O'Reilly, ©2010 (siehe S. 11).

[4] AWS. *Amazon DynamoDB: BatchGetItem*. 19.06.2014. URL: http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_BatchGetItem.html (besucht am 25.07.2014) (siehe S. 27).

[5] AWS. *Amazon DynamoDB: BatchWriteItem*. 7.08.2014. URL: http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_BatchWriteItem.html (besucht am 16.08.2014) (siehe S. 24, 25).

[6] AWS. *Amazon DynamoDB: Condition*. 19.06.2014. URL: http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Condition.html (besucht am 26.07.2014) (siehe S. 28).

[7] AWS. *Amazon DynamoDB: DeleteItem*. 7.08.2014. URL: http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_DeleteItem.html (besucht am 13.08.2014) (siehe S. 26).

[8] AWS. *Amazon DynamoDB: DynamoDB Data Model: Developer Guide (API Version 2012-08-10)*. 2012. URL: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DataModel.html (besucht am 14.07.2014) (siehe S. 17–19, 22).

[9] AWS. *Amazon DynamoDB: GetItem*. 19.06.2014. URL: http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_GetItem.html (besucht am 25.07.2014) (siehe S. 27).

[10] AWS. *Amazon DynamoDB: Global Secondary Indexes*. 11.09.2014. URL: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html (besucht am 18.09.2014) (siehe S. 22).

[11] AWS. *Amazon DynamoDB: Limits in DynamoDB*. 19.06.2014. URL: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Limits.html (besucht am 24. 07. 2014) (siehe S. 23).

[12] AWS. *Amazon DynamoDB: Local Secondary Indexes*. 11.09.2014. URL: ttp://docs.aws.amazon.com/amazondynamodb/latest/developerguide/LSI.html (besucht am 18. 09. 2014) (siehe S. 22).

[13] AWS. *Amazon DynamoDB: Making HTTP Requests to DynamoDB*. 19.06.2014. URL: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/MakingHTTPRequests.html (besucht am 24. 07. 2014) (siehe S. 19).

[14] AWS. *Amazon DynamoDB: Provisioned Throughput in Amazon DynamoDB: Developer Guide (API Version 2012-08-10)*. 2012. URL: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ProvisionedThroughputIntro.html (besucht am 14. 07. 2014) (siehe S. 17, 33, 96).

[15] AWS. *Amazon DynamoDB: PutItem*. 7.08.2014. URL: http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API\textunderscore PutItem.html (besucht am 13. 08. 2014) (siehe S. 23).

[16] AWS. *Amazon DynamoDB: Query*. 19.06.2014. URL: http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API\textunderscore Query.html (besucht am 25. 07. 2014) (siehe S. 27, 29, 30).

[17] AWS. *Amazon DynamoDB: Query and Scan Operations*. 19.06.2014. URL: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/QueryAndScan.html (besucht am 25. 07. 2014) (siehe S. 27, 28).

[18] AWS. *Amazon DynamoDB: Scan*. 19.06.2014. URL: http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API\textunderscore Scan.html (besucht am 26. 07. 2014) (siehe S. 27, 31).

[19] AWS. *Amazon DynamoDB: UpdateItem*. 7.08.2014. URL: http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API\textunderscore UpdateItem.html (besucht am 16. 08. 2014) (siehe S. 25, 26).

[20] AWS. *Amazon DynamoDB: What is Amazon DynamoDB?* 7.08.2014. URL: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html (besucht am 08. 08. 2014) (siehe S. 17).

[21] Christian Baun u. a. *Cloud computing: Web-based dynamic IT services*. Berlin und Heidelberg: Springer, 2011 (siehe S. 37).

[22] Alex Biryukov u. a. „Database Encryption“. In: *Encyclopedia of Cryptography and Security*. Hrsg. von van Tilborg,Henk C. A und Sushil Jajodia. Boston, MA: Springer US, 2011, S. 307–312 (siehe S. 38–43).

[23] Alexandra Boldyreva u. a. „Order-Preserving Symmetric Encryption“. In: *Advances in Cryptology - EUROCRYPT 2009*. Hrsg. von David Hutchison u. a. Bd. 5479. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, S. 224–241 (siehe S. 2, 37, 74, 122).

[24] Eric Brewer. „CAP twelve years later: How the "rules" have changed“. In: *Computer* 45.2 (2012), S. 23–29 (siehe S. 6–8).

[25] Eric. Brewer. *Principles of distributed computing: Proceedings*. New York: ACM Press, 2000 (siehe S. 7).

[26] David Cash u. a. „Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries“. In: *Advances in Cryptology – CRYPTO 2013*. Hrsg. von David Hutchison u. a. Bd. 8042. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 353–373 (siehe S. 52, 53, 122).

[27] Rick Cattell. „Scalable SQL and NoSQL data stores“. In: *ACM SIGMOD Record* 39.4 (2011), S. 12 (siehe S. 9, 10).

[28] Michael Coles und Rodney Landrum. *Expert SQL server 2008 encryption*. New York, NY: Apress, 2009 (siehe S. 40, 43).

[29] Craig Gentry. „A fully homomorphic encryption scheme“. Diss. Stanford University, 2009 (siehe S. 36).

[30] Giuseppe DeCandia u. a. „Dynamo: Amazon's highly available key-value store“. In: *twenty-first ACM SIGOPS symposium*. Hrsg. von Thomas C. Bressoud und M. Frans Kaashoek, S. 205 (siehe S. 12, 13, 15).

[31] Stefan Edlich. *NoSQL Databases: Your Ultimate Guide to the Non - Relational Universe!* 2014. URL: http://nosql-database.org/ (siehe S. 3–6, 9).

[32] Stefan Edlich u. a. *NoSQL: Einstieg in die Welt nichtrelationaler Web-2.0-Datenbanken*. München: Hanser, 2010 (siehe S. 3, 4, 6–12, 15).

[33] Yuval Elovici u. a. „A Structure Preserving Database Encryption Scheme“. In: *Secure Data Management*. Hrsg. von David Hutchison u. a. Bd. 3178. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, S. 28–40 (siehe S. 40, 47–50).

[34] Youssef Gahi, Mouhcine Guennoun und Kalil El-Katib. „A Secure Database System using Homomorphic Encryption Schemes“. In: *DBKDA 2011 : The Third International Conference on Advances in Databases, Knowledge, and Data Applications*, S. 54–58 (siehe S. 36).

[35] Oliver Haase und Margit Roth. *Kommunikation in verteilten anwendungen: Einführung in Sockets, Java RMI, CORBA und Jini*. 2., überarbeitete und erweiterte Auflage. Munich, Germany: Oldenbourg Verlag, 2008 (siehe S. 6).

[36] Hakan Hacigümüş u. a. „Executing SQL over encrypted data in the database-service-provider model“. In: *the 2002 ACM SIGMOD international conference*. Hrsg. von David DeWitt, Michael Franklin und Bongki Moon, S. 216 (siehe S. 2, 46, 47, 122).

[37] James Heather, Steve Schneider und Vanessa Teague. „Cryptographic protocols with everyday objects“. In: *Formal Aspects of Computing* 26.1 (2014), S. 37–62 (siehe S. 38, 56).

[38] B. Kaliski. *PKCS #5: Password-Based Cryptography Specification Version 2.0*. 107, rfcmarkup version 1., 8.06.2014. URL: http://tools.ietf.org/html/rfc2898 (besucht am 29. 07. 2014) (siehe S. 36).

[39] Lianzhong Liu und Jingfen Gai. „A new lightweight database encryption scheme transparent to applications“. In: *2008 6th IEEE International Conference on Industrial Informatics (INDIN)*, S. 135–140 (siehe S. 47).

[40] Dan McCreary und Ann Kelly. *Making sense of NoSQL: A guide for managers and the rest of us*. Shelter Island, NY: Manning, 2014 (siehe S. 3, 4, 9–12).

[41] Joe McKendrick. *Cloud IT Spending Surges, And May Be More Pervasive Than The Numbers Show - Forbes*. 2015. URL: http://www.forbes.com/sites/joemckendrick/2015/07/10/cloud-it-spending-surges-and-may-be-more-pervasive-than-the-numbers-show/ (siehe S. 1).

[42] Microsoft SQLServer 2008 R2. *Understanding Pages and Extents*. 17.07.2014. URL: http://technet.microsoft.com/en-us/library/ms190969(v=sql.105).aspx (besucht am 17. 07. 2014) (siehe S. 43).

[43] Microsoft SQLServer 2014. *Transparent Data Encryption (TDE)*. 1.08.2014. URL: http://msdn.microsoft.com/en-us/library/bb934049.aspx (besucht am 01. 08. 2014) (siehe S. 2, 43).

[44] Oracle. *Java - Cryptography Architecture: Standard Algorithm Name Documentation*. 8.05.2014. URL: http://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html\#pkcs5Pad (besucht am 29. 07. 2014) (siehe S. 36).

[45] Oracle. *Tablespaces, Datafiles, and Control Files*. 11.11.2014. URL: https://docs.oracle.com/cd/B19306_01/server.102/b14220/physical.htm#i2009 (besucht am 28. 11. 2014) (siehe S. 40, 41).

[46] Oracle. *Transparent Data Encryption (TDE)*. URL: http://www.oracle.com/technetwork/database/options/advanced-security/index-099011.html (besucht am 28. 11. 2014) (siehe S. 2).

[47] Oracle 10g Release 2. *DBMS_CRYPTO*. 2005. URL: http://docs.oracle.com/cd/B19306\textunderscore 01/appdev.102/b14258/d\textunderscore crypto.htm (besucht am 01. 08. 2014) (siehe S. 41, 42).

[48] Oracle 11g. *Developing Applications Using the Data Encryption API*. 22.11.2014. URL: http://docs.oracle.com/cd/B28359_01/network.111/b28531/data_encryption.htm#DBSEG80084 (siehe S. 42).

[49] Oracle 11g. *Securing Stored Data Using Transparent Data Encryption*. 11.12.2014. URL: http://docs.oracle.com/cd/E11882_01/network.112/e40393/asotrans.htm (siehe S. 41, 42).

[50] Raluca Ada Popa u. a. „CryptDB: Protecting Confidentiality with Encrypted Query Processing“. In: *the Twenty-Third ACM Symposium*. Hrsg. von Ted Wobber und Peter Druschel, S. 85 (siehe S. 2, 37, 44, 45).

[51] Arjol Qeleshi. „EncryptedCassandra - Client-Side Encryption through Query-Rewriting of CQL“. Magisterarb. Linz: Johannes Kepler Universtät Linz, 2015. URL: http://www.dke.jku.at/research/publications/details.xq?type=masterthesis&code=MT1508 (siehe S. 2).

[52] Bruce Schneier. *Applied cryptography: Protocols, algorithms, and source code in C*. 2nd ed. New York: Wiley, 1996 (siehe S. 34, 36, 40).

[53] *Securing Data: Database 2 Day Security Guide*. 11.12.2014. URL: http://docs.oracle.com/cd/E11882_01/server.112/e10575/tdpsg_securing_data.htm (siehe S. 41, 42).

[54] William Stallings. *Cryptography and network security: Principles and practice*. 5th ed. Boston: Prentice Hall, 2011 (siehe S. 35).

[55] Douglas R. Stinson. *Cryptography: Theory and practice*. 3rd ed. CRC Press series on discrete mathematics and its applications. Boca Raton: Chapman & Hall/CRC, 2006 (siehe S. 34–37).

[56] Carlo Strozzi. *NoSQL: A Relational Database Management System*. 1998,2010. URL: http://www.strozzi.it/cgi-bin/CSA/tw7/I/en\textunderscore US/nosql/Home\%20Page (besucht am 07. 07. 2014) (siehe S. 3).

[57] Andrew S. Tanenbaum und Maarten van Steen. *Distributed systems: Principles and paradigms*. 2nd ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2007 (siehe S. 4, 6, 8, 15).

[58] Thierry Bertin-Mahieux u. a. „The Million Song Dataset“. In: *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*. 2011 (siehe S. 94).

[59] Tony Bako. *What's the difference between sharding and partition?* Quora. URL: https://www.quora.com/Whats-the-difference-between-sharding-and-partition (siehe S. 4).

[60] Werner Vogels. *Amazon DynamoDB: A Fast and Scalable NoSQL Database Service Designed for Internet Scale Applications*. 2012. URL: http://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html (besucht am 14. 07. 2014) (siehe S. 16).

[61] Werner Vogels. „Eventually Consistent“. In: *ACM Queue* 6.6 (2008), S. 14 (siehe S. 8, 9, 17).

Anhang A

Appendix

A.1 Entwicklung

Der Verschlüsselungsclient wurde mithilfe der AWS Java SDK in der Version 1.8.5[1] implementiert. Er basiert auf Java 7. Die Entwicklung wurde mithilfe von Eclipse durchgeführt. Amazon bietet hier ein Plugin[2] für Eclipse an, welches eine einfach Einbindung der AWS-Dienste ermöglicht.

Für die Nutzung der AWS-Dienste muss man sich bei AWS registrieren. Man kann anschließend die Zugangsdaten für die Amazon DynamoDB anlegen. Hier werden diverse Methoden wie Access Keys oder X.509 Certificates angeboten. Diese Daten werden vom SDK benötigt, um eine gesicherte Verbindung herstellen zu können. Für die lokale Entwicklung wird die DynamoDB-Local[3] angeboten. Diese simuliert das gleiche Verhalten wie die cloud-basierte DynamoDB.

Der CryptDB OPE C++ Code wurde mit Microsoft Visual C++ kompiliert, weswegen das Microsoft Visual C++ 2012 Redistributable (x86) – 11.0.61030 benötigt wird, um die generierten Binaries ausführen zu können, falls OPE verwendet werden soll.

Bei der Entwicklung eines Projektes, welche die DynamoDB-API und den neuen Verschlüsselungsclient verwenden möchte, muss zuerst ein Java Projekt erstellt werden, welches die AWS-Bibliothek und den Verschlüsselungsclient einbindet. In diesem Java-Projekt muss nun eine CryptoConfig.xml und ein DataDictionary.xml angelegt werden. Diese beiden Daten müssen sich auf der gleichen Ebene befinden wie der src-Ordner. Desweiteren muss sich die Crypto_Ope.dll im src-Ordner befinden. Eine beispielhafte Konfiguration ist in Abbildung A.1 ersichtlich.

A.2 Installation

Die Einbindung des Verschlüsselungsclients in ein existierendes Projekt benötigt die Konfiguration und das Setup, welches in Abschnitt A.1 beschrieben ist. Es werden vor

[1] AWS SDK für Java - http://aws.amazon.com/de/sdk-for-java/
[2] AWS Toolkit for Eclipse Core http://aws.amazon.com/de/eclipse/
[3] DynamoDB Local https://aws.amazon.com/de/blogs/aws/dynamodb-local-for-desktop-development/

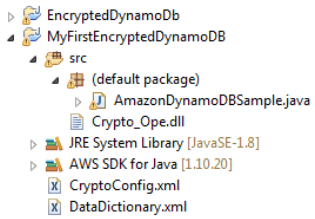

Abbildung A.1: Konfiguration eines Projektes, das den Verschlüsselungsclient verwendet.

allem die DLL (Dynamic Link Libary) OPE.dll und die Konfigurationsdateien vom Client benötigt. Verwendet das existierende Projekt bereits den AmazonDynamoDbClient, so muss der Code nur minimal angepasst werden, um den Verschlüsselungsclient zu verwenden. Eine Änderung ist nur bei der Initialisierung des Client-Objekts zum Laden der CryptoConfig und des Data-Dictionary nötig. Ein Auszug ist in Beispiel A.2 ersichtlich.

```
1  AWSCredentials credentials = new ProfileCredentialsProvider("default").getCredentials();
2  //init EncryptedAmazonDynamoDbClient
3  CryptoConfigFile.readCryptoFile("CryptoConfig.xml");
4   Map<String, HashMap<String, DataDictionaryAttributeDefinition>> datadic =
      DataDictionaryFile.readDataDictionaryFile("DataDictionary.xml");
5
6  AmazonDynamoDBClient dynamoDB = new EncryptedAmazonDynamoDbClient(credentials);//former: new
     AmazonDynamoDBClient(credentials);
7  Region usWest2 = Region.getRegion(Regions.US_WEST_2);
8  dynamoDB.setRegion(usWest2);
9  dynamoDB.setEndpoint("http://localhost:8050"); //to use DynamoDB-Local
```

Abbildung A.2: Beispielhafte Initialisierung des Verschlüsselungsclients.

A.3 Codestruktur

Die Codestruktur des Verschlüsselungsclient basiert auf der präsentierten Architektur in Abschnitt 4.2 und der dort gezeigten Abbildung 4.10.

Der Verschlüsselungsclient verwendet als Kommunikationsschnittstelle zur Datenbank den AmazonDynamoDBClient, indem er dessen Implementierung erweitert. Ein Überblick über die Paketstruktur des EncryptedDynamoDbClient ist in Abbildung A.3 ersichtlich. Es werden folgende Pakete gezeigt:

com.amazonaws.services.dynamodbv2 – Dieses Paket beinhaltet die Schnittstelle zur Amazon DynamoDB-Datenbank. Dessen Implementierung wird vom Verschlüsselungsclient erweitert.

at.jku.dke.encryptedDynamoDb.api – Dieses Paket beinhaltet die gesamte Verschlüsselungsclient-Implementierung.

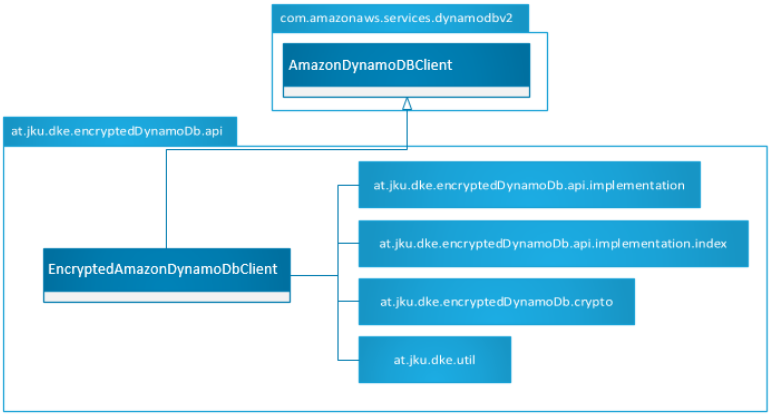

Abbildung A.3: Übersicht über die Paketstruktur des Verschlüsselungsclient.

at.jku.dke.encryptedDynamoDb.api.implementation – In diesem Paket befindet sich die Logik der Verschlüsselung und deren Anwendung auf das Datenmodell der DynamoDB. Es handelt sich hierbei um das zuvor beschriebene Modul *Encrypted-Client-Core*.

at.jku.dke.encryptedDynamoDb.api.implementation.index - Dieses Paket implementiert die in der vorliegenden Arbeit beschriebenen Indizes, Bucketing und den invertierten verschlüsselten Index. Im Abschnitt Architektur 4.2 wird dieses Paket als Index-Modul bezeichnet.

at.jku.dke.encryptedDynamoDb.api.crypto - Dieses Paket (Crypto-Modul) beinhaltet die Verschlüsselungsalgorithmen und Hash-Funktionen.

at.jku.dke.util – Das Util-Paket beinhaltet die Logik des Data-Dictionarys, der Crypto-Config und des Loggings.

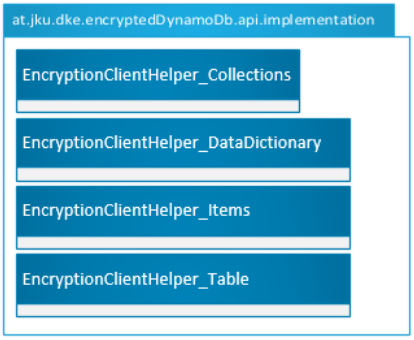

Abbildung A.4: Inhalt des Encrypted-Client-Core Modul.

Die Implementierung des Moduls *Encrypted-Client-Core* befindet sich, wie bereits beschrieben, im Paket `at.jku.dke.encryptedDynamoDb.api.implementation` (siehe Abbildung A.4). Es besteht aus den nachfolgenden vier Klassen:

EncryptionClientHelper_Collections – Diese Klasse beinhaltet häufig verwendete Operationen, welche auf Java-Mengen (Collections) durchgeführt werden. Eine Beispiel-Operation ist: An welcher Index-Position dieses Arrays ist der Datensatz zu finden, der den übergebenen Namen besitzt?

EncryptionClientHelper_DataDictionary – Diese Klasse kapselt die Zugriffe auf das Data-Dictionary. Eine beispielhafte Abfrage ist: Welchen Datentyp hat dieses Attribut?

EncryptionClientHelper_Items – Diese Klasse implementiert die Verschlüsselungs- und Entschlüsselungsabläufe nach dem HDSE-Verschlüsselungschema. Eine beispielhafte Operation ist die Verschlüsselung eines einfachen Attributwerts oder eines Schlüsselattributs.

EncryptionClientHelper_Tables – In der letzten Klasse dieses Pakets wird die Ver- und Entschlüsselung des Tabellennamens und dessen Tabelleninformationen durchgeführt. Diese Daten müssen, wie bereits in den Abschnitten 2.6.3 und 5.3 erörtert, aufgrund der Einschränkung des Zeichensatzes von der DynamoDB anders verschlüsselt werden.

Das Paket `at.jku.dke.encryptedDynamoDb.api.implementation.index` beinhaltet die beschriebene Logik des Index-Moduls. In Abbildung A.6 werden die einzelnen Klassen des Moduls gezeigt.

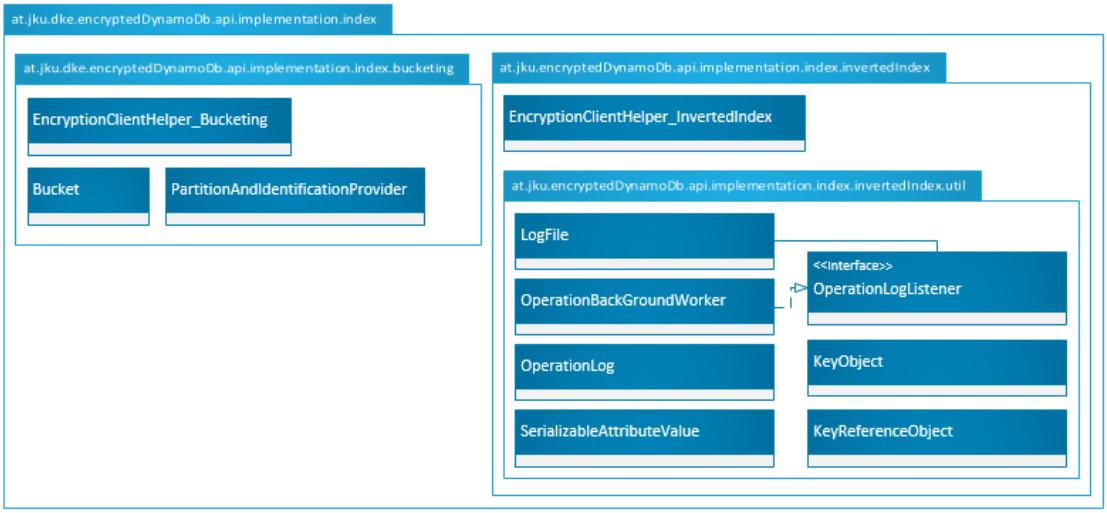

Abbildung A.5: Inhalt des Encrypted-Client-Core-Modul.

Die Kernbestandteile des Bucketing-Index sind die Klassen EncyptionClientHelper _Bucketing, PartionAndIdentificationFunctionProvider und Bucket. Die erste Klasse

realisiert die Postfilterung der Scan-Implementierung mithilfe des Bucketing-Index und die Erstellung des Bucketing-Index. Die zweite Klasse implementiert die Erstellung des Buckets und des Identifikationswerts.

Die zweite Index-Implementierung, der invertierte Index, ist im Vergleich zur Bucketing-Implementierung aufwändiger implementiert. Er besteht aus einer EncyptionClientHelper _InvertedIndex-Klasse, die alle nötigen Operationen für die Verwaltung der Index-Tabellen beinhaltet. Des Weiteren beinhaltet diese Index-Implementierung ein weiteres Paket namens `at.jku.dke.encryptedDynamoDb.api.implementation.index. invertedIndex.util`. In diesem Paket befindet sich die komplette Logik des Backgroundworkers (siehe Abschnitt 5.5.2), der hierzu geschriebenen Log-Dateien sowie die Datenstruktur (KeyObject und KeyReferenceObject), die die Referenz auf den indizierten Datensatz speichert.

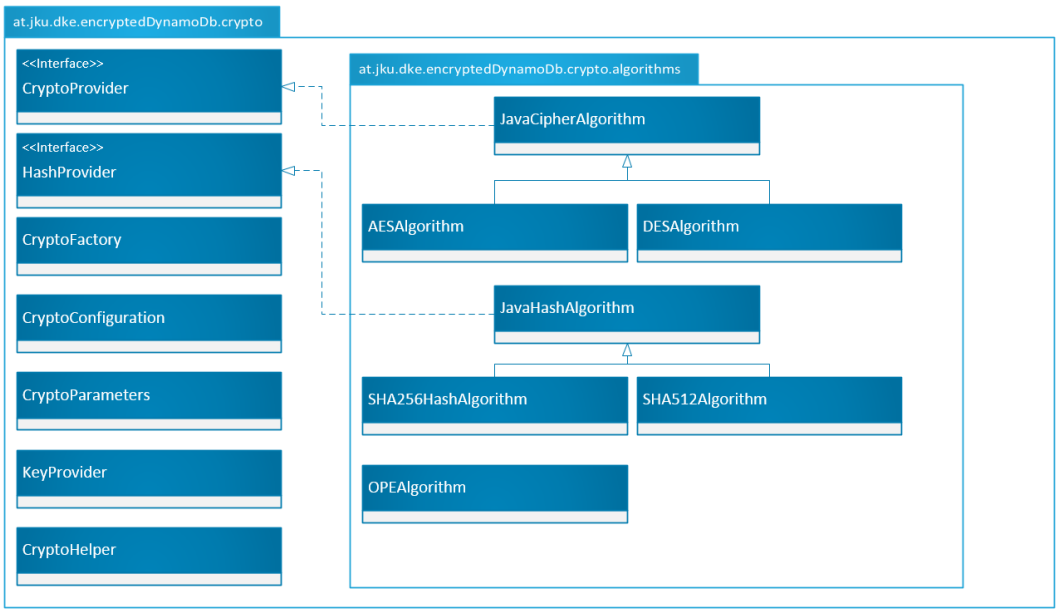

Abbildung A.6: Inhalt des Crypto-Moduls. Neben den Standardalgorithmen, Hashfunktionen und dem OPE befindet sich hier auch die Implementierung der Teilschlüsselverkettung und die Schnittstelle zu den geladenen Konfigurations-Parametern.

Das vorletzte Paket, welches der Verschlüsselungsclient verwendet, ist `at.jku.dke. encryptedDynamoDb.api.crypto`, welches auch als Crypto-Modul bezeichnet wird, und beinhaltet die Standardverschlüsselungsalgorithmen wie AES und DES sowie Hashfunktionen wie SHA256 und SHA512. In diesem Paket befindet sich auch die JNI Schnittstelle zur OPE-Bibliothek. Des Weiteren befindet sich hier mit der Klasse *CryptoConfiguration* eine zentrale Schnittstelle zur vom Nutzer definierten Crypto-Konfiguration. Darüber hinaus wird durch die Klasse *KeyProvider* die Verkettung der Teilschlüssel des HDSE realisiert.

Das letzte Paket `at.jku.dke.util` (siehe Abbildung A.7) beinhaltet die Lesefunk-

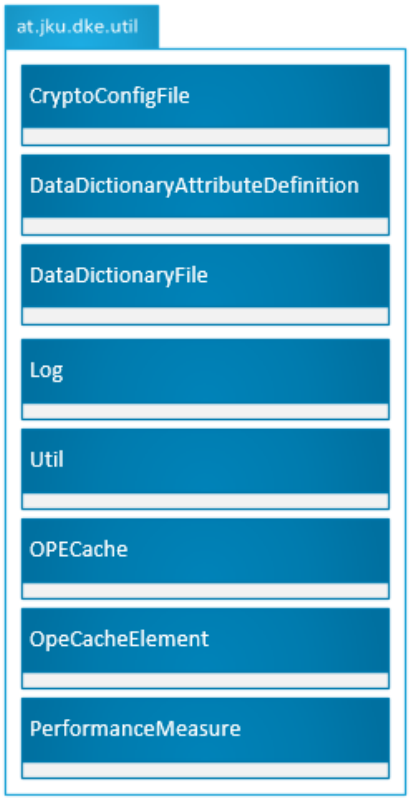

Abbildung A.7: Dieses Paket beinhaltet die Realisierungen der Data-Dictionary-Datei, der Crypto-Config-Datei und weiteren Hilfsklassen.

tionalität der Crypto-Konfigurationsdatei und der Data-Dictionary-Datei sowie die Implementierung des OpeOPE-Cache. Die Klasse-Util besteht aus Hilfsmethoden zur Verschlüsselung von hexadezimalen und Base64-Kodierungen sowie Serialisierungsmethoden. Die Log-Implementierung ist eine Schnittstelle zur verwendeten Log4J-Bibliothek[4].

[4]Log4J - http://logging.apache.org/log4j/2.x/