Douglas Cunningham, Petra Hofstedt, Klaus Meer, Ingo Schmitt (Hrsg.): INFORMATIK 2015
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2015

Durchsuchbare Verschlüsselung in NoSQL Datenbanken

Tim Waage[1]

Abstract: Heutzutage werden immer häufiger sog. „Cloud"-Speicherdienste eingesetzt, sei es als Backup oder um die eigenen laufenden Kosten für IT-Infrastruktur zu senken. Leider sind diese aber nicht unbedingt vertrauenswürdig, wenn es um die Behandlung sensibler Informationen geht. Eine Verschlüsselung der Daten kann dem entgegenwirken, schränkt dann aber die Möglichkeiten ein, weiterhin wie gewohnt mit der Datenbank zu interagieren. Insbesondere wird das effiziente Durchsuchen erschwert. Eine Reihe von Schemata zur durchsuchbaren Verschlüsselung nehmen sich bereits dieses Problems an, jedoch mangelt es an praktischen Implementationen und der Erprobung mit existierenden Datenbanktechnologien. Die vorliegende Arbeit präsentiert eben solche Implementationen von zwei sehr unterschiedlichen Ansätzen ([SWP00, HK14]) zur durchsuchbaren Verschlüsselung in zwei populären NoSQL Datenbanken: Apache Cassandra und Apache HBase. Dabei liegt der Fokus auf den Problemen der Schemata im praktischen Einsatz, einem Performacevergleich und der Diskussion von Optimierungsmöglichkeiten.

Keywords: Durchsuchbare Verschlüsselung, verschlüsselte Datenspeicherung, NoSQL Datenbanken, Benchmarking, Apache Cassandra, Apache HBase

1 Einleitung

Cloud Computing als Alternative zur Datenverarbeitung auf eigener Hardware befreit den Nutzer von den damit verbundenen administrativen Aufgaben und verschafft ihm zusätzliche Hardwareressourcen. Die vorliegende Arbeit betrachtet insbesondere den Anwendungsfall der Speicherung von Datensätzen in Clouddatenbanken. Da traditionelle SQL Datenbanken die Anforderungen moderner Internetapplikationen oft nicht mehr erfüllen, werden hierfür mehr und mehr sog. „NoSQL"(Not only SQL) Datenbanken eingesetzt. Diesen fehlen in der Regel aber jegliche Sicherheitsmechanismen. So stellen potenziell sowohl mutwillige Einbrecher, aber auch die Administratoren oder sogar andere Nutzer der Datenbanken eine Gefahr für vertrauliche Daten dar.

Das Hauptziel dieser Arbeit ist es daher, die Sicherheit der extern gelagerten Daten gegen nicht legitimierten Zugriff zu schützen. Dafür werden alle Daten, die den Einflussbereich des Nutzers verlassen, symmetrisch verschlüsselt. Die Datenbank enthält somit zu jedem Zeitpunkt ausschließlich verschlüsselte Daten. Die entsprechend notwendigen Schlüssel werden auf der Nutzerseite verwaltet. Es wird angenommen, dass potenzielle Angreifer die Daten zwar versuchen zu lesen, sie aber sonst nicht manipulieren (was ohnehin nur dazu führen würde, dass eine Entschlüsselung höchstwahrscheinlich nicht mehr möglich wäre, der Angreifer somit keine Kenntnis der verschlüsselten Informationen erlangt).

[1] Georg August Universität Göttingen, Gruppe: Knowledge Engineering, Goldschmidtstrasse 7, 37077 Göttingen, tim.waage@informatik.uni-goettingen.de

Die vorliegende Arbeit leistet somit folgendes:

- Implementation zweier Schemata für durchsuchbare Verschlüsselung, beide grundverschieden in ihren Ansätzen (sowohl scan-, als auch indexbasiert), inklusive der notwendigen Interaktionen zu den NoSQL Datenbanken Apache Cassandra und Apache HBase (derzeit die populärsten Datenbanken ihrer Kategorie [So])
- Quantifizierung der Performance beider Schemata in Kombination mit beiden Datenbanken und Überblick der jeweiligen Stärken und Schwächen
- Einschätzung praktischen Umsetzbarkeit, Erörterung von Optimierungspotenzialen

2 Die Datenbanken

Wie eingangs erwähnt besitzen NoSQL Datenbanken in der Regel keinerlei Sicherheitsmechanismen, wie beispielsweise eine Nutzerverwaltung und das damit verbundene Zugriffsrechtemanagement. Es wird generell angenommen, dass ein entsprechendes Frontend diese Aufgaben übernimmt. Der Einbruch in selbiges würde dann die komplette Datenbank offenlegen. Eine Möglichkeit trotzdem noch den Schutz der Daten zu gewährleisten ist die Speicherung in verschlüsselter Form. NoSQL Datenbanken bieten aber auch dafür keinerlei native Mechanismen an.

Die vorliegende Arbeit beschäftigt sich daher näher mit Apache Cassandra [LM10, Tha] und Apache HBase [Bo11, Thb]. Sie gehören sowohl zur Gruppe der Key-Value Datenbanken, als auch zu den sog. Spaltenfamiliendatenbanken. Beide wurden in Java implementiert und bieten hohe Verfügbarkeit, da sie bereits nativ ihre Daten auf vielen Servern verteilen und replizieren. Cassandra setzt dabei auf das Peer-to-Peer Prinzip, benutzt das Gossip Protokoll zur Koordination der Knoten und ist somit nach dem CAP Theorem [Br00, Br10] besonders ausgelegt für Hochverfügbarkeit und Partitionstoleranz. HBase hingegen basiert auf einer Master-Slave-Struktur. Es wird nicht unterschieden zwischen verschiedenen Datentypen wie Zahlen und Text, stattdessen wird alles als Byte-Array behandelt. Ein Zookeeper System verwaltet dabei die verschiedenen Prozesse. Bezüglich des CAP Theorems ist HBase ausgelegt für Konsistenz und Hochverfügbarkeit.

3 Die Schemata zur durchsuchbaren Verschlüsselung

Für unsere Versuche nutzen wir zwei Schemata zur durchsuchbaren Verschlüsselung. Zum einen testen wir den Algorithmus von Song et. al [SWP00] (im folgenden „SWP Schema“, in der Literatur so benannt nach den Anfangsbuchstaben der Nachnamen der Autoren). Es basiert auf linearen Scans, also dem kompletten Durchlauf der verschlüsselten Daten. Es ist eines der ersten Schemata für durchsuchbare Verschlüsselung überhaupt und auch das einzige, das in einem praktisch relevanten Kontext überhaupt schon einmal implementiert wurde [Po12]. Trotz seines auf den ersten Blick ineffizienten Linear-Scan-Ansatzes besitzt es attraktive Eigenschaften in der Praxis, die eine genauere Untersuchung durchaus rechtfertigen. Zum anderen testen wir ein relativ neues Schema, vorgeschlagen von Hahn und

Kerschbaum [HK14] (im folgenden „SUISE Schema“, kurz für „securely updating index-based searchable encryption“), welches wiederum zwei Indizes nutzt, um eine effiziente Suche zu ermöglichen.

Die folgenden beiden Kapitel stellen die Funktionsweise beider Schemata kurz vor und präsentieren einen Überblick der jeweiligen Stärken und Schwächen.

3.1 Das SWP Schema [SWP00]

Das SWP Schema ist praktisch die einzige Wahl, wenn die Aufrechterhaltung eines Index aus praktischen Gründen vermieden werden soll [Bö14]. Es untergliedert sich in vier Subschemata, beginnend mit dem „Basic Scheme“. Die *Verschlüsselung* funktioniert wie folgt: Jedes Klartextwort W wird mittels Padding oder Splitting auf eine Länge von n Bytes gebracht. Für jedes dieser Worte wird mit einem Zufallsgenerator G ein Wert S der Länge $n-m$ Bytes (mit $m < n$) erzeugt. Aus S wird dann mittels eines Schlüssels k ein (Hash-)Wert $F_k(S)$ der Länge m errechnet. $F_k(S)$ wird an S angehangen (was wieder ein Wort der Länge n ergibt) und XOR-verknüpft mit W um den finalen Geheimtext C zu erzeugen.

Die *Suche* im Schlüsseltext ist dann simpel. Wird nach einem bestimmten Wort W gesucht, so wird für jedes Wort im Geheimtext überprüft, ob $W \oplus C$ der Form $S||F_k(S)$ genügt.

Alle anderen Schemata der Autoren sind Erweiterungen dieses Grundschemas, die zusätzliche Sicherheitseigenschaften bringen sollen. Das zweite Schema („controlled searching“) erlaubt die Suche ausschließlich nach dem gewünschten Suchwort, indem es die Erzeugung von k jeweils vom Klartextwort abhängig macht. Das dritte Schema („hidden searches“) führt eine Vorverschlüsselung $E(W)$ ein, was eine Offenlegung des gewünschten Suchwortes beim Suchvorgang vermeidet. Das zweite und dritte Schema in Kombination bringen die unerwünschte Eigenschaft mit, dass die Erzeugung von k nun an vorverschlüsselte Klartextworte gebunden ist, was im Zweifel dazu führen kann, das der Dateneigentümer nicht mehr in der Lage ist, seine eigenen Daten zu entschlüsseln. Das vierte Schema („Final Scheme“) behebt dieses Problem, indem das vorverschlüsselte Wort $E(W)$ in zwei Teile L mit Länge $n-m$ und R mit Länge m gespalten wird. L kann über S rekonstruiert werden, da der Dateneigentümer den Seed des Zufallsgenerators G kennt. Aus der Errechnung von $F_k(S)$ kann dann R erschlossen werden. Aus Platzgründen sei für eine ausführlichere Darstellung an dieser Stelle auf das Originalpaper [SWP00] verwiesen.

In der vorliegenden Arbeit ist stets das „Final Scheme“gemeint, wenn auf das SWP Schema referenziert wird, da dieses alle möglichen Sicherheitsvorteile beinhaltet. Aus praktischer Sicht hat das SWP Schema folgende Stärken (+) und Schwächen (–):

\+ Aufgrund des linear-scan-Konzepts ist es nicht notwendig einen Index oder andere Statusinformationen (außer den Schlüsseln selbst) zu speichern, zu erstellen oder zu verwalten.

\+ Verschlüsselung und Suche können instantan durchgeführt werden ohne Anfragen an eine Indexstruktur stellen zu müssen.

+ Im Gegensatz zu den meisten index-basierten Schemen wird nicht nur die Information geliefert, ob ein Suchwort in einem Dokument vorhanden ist oder nicht. Auch Anzahl der Treffer und exakte Positionen im Dokument können bestimmt werden

+ Der SWP Algorithmus ist naturbedingt leicht umsetzbar für die darunterliegenden Datenbanken, bzw. generell in Client-Server-Umgebungen.

– Für große Datenmengen kann die Suche lange dauern.

– Es kann nur nach kompletten Worten gesucht werden, nicht aber nach Substrings oder regulären Ausdrücken.

– Der Geheimtext kann deutlich größer werden als der Klartext, da der SWP Algorithmus alle Klartextworte mit einer Länge von (einem Vielfachen von) n Bytes verschlüsselt (siehe Abschnitt 5.4).

3.2 Das SUISE Schema [HK14]

Im Gegensatz zum SWP Schema arbeitet SUISE mit den beiden Indizes γ_f und γ_w. γ_f speichert die verschiedenen Worte pro Dokument [2] in verschlüsselter Form. γ_w ist ein sog. invertierter Index, der zu allen bereits durchgeführten Suchen die Identifikatoren (im Folgenden kurz „IDs“) der jeweils gefunden Dokumente speichert, was wiederholtes Suchen nach dem gleichen Wort extrem beschleunigt.

Während der *Verschlüsselung* wird eine Liste aller verschiedenen Worte $(w_1, ..., w_n)$ pro Dokument f kreiert. Ein Zufallsgenerator G erzeugt die gleiche Anzahl an Werten $(s_1, ..., s_n)$. Dann werden für jedes Wort w die folgenden Schritte ausgeführt: (1) mittels einer Zufallsfunktion F und eines Schlüssels k_1 wird ein Such-Token $r_w = F_{k_1}(w)$ erzeugt. (2) Wurde r_w schon einmal benutzt, wird es zur Liste x hinzugefügt. (3) Mittels eines Zufallsorakels H wird $c = H_{r_w}(s)||(s)$ erzeugt. Die Datenbank speichert dann alle c's des Dokuments f in γ_f. Zusätzlich wird die ID des Dokuments $ID(f)$ in γ_w abgelegt für alle Einträge in x

Für die *Suche* nach Dokumenten, die ein Wort w beinhalten, wird dann zuerst wieder das Such-Token $r_w = F_{k_1}(w)$ berechnet. Existiert bereits ein Eintrag für r_w in γ_w (d.h. es wurde schon einmal nach w gesucht), kann einfach die in γ_w gespeicherte Liste der entsprechenden IDs geliefert werden. Andernfalls wird diese Liste wie folgt erzeugt: jedes c in γ_f wird zerlegt in $l||r$. Falls $H_{r_w}(r) = l$ wird die entsprechende Dokument-ID zur Ergebnismenge aufgenommen und der entsprechende Eintrag in γ_w erzeugt. Für eine ausführlichere Darstellung sei erneut auf das Originalpaper verwiesen [HK14].

Wie für das SWP Schema folgt eine Auflistung der Stärken (+) und Schwächen (–):

+ Wiederholtes Suchen nach einem Wort w ist in konstanter Zeit möglich.

+ Die Verschlüsselung benötigt nur so viele Iterationen, wie es verschiedene Worte pro Dokument gibt. Im Gegensatz dazu muss SWP alle Verschlüsselungsschritte für alle Worte ausführen.

[2] gleiche Worte im selben Dokument werden also nur als ein Eintrag im Index abgelegt

– Trotz Indizes erfordert die Suche nach noch nicht zuvor gesuchten Worten linearen Aufwand.

– Der Index γ_f speichert $c = H_{r_w}(s)||(s)$ für alle verschiedenen Worte pro Dokument. c ist somit fast immer deutlich länger als das ursprüngliche Klartextwort, wodurch γ_f sehr viel Speicherplatz belegen kann (siehe Abschnitt 5.5).

– Es werden nur die verschiedenen Worte pro Dokument erfasst, d.h. SUISE kann nicht wie SWP die genaue Anzahl und Positionen des Suchworts im Dokument liefern.

– Auch SUISE beherrscht keine Substrings oder regulären Ausdrücke.

4 Implementation

Beide Schemata wurden in Java Version 8 implementiert. Die Java Cryptography API (JCA) ist die Basisarchitektur für kryptographische Algorithmen in Java. Sie spezifiziert Interfaces, die dann wiederum von sog Kryptographie Providern implementiert werden. In der vorliegenden Arbeit kommen zwei davon zum Einsatz: die Java Cryptography Extension (JCE, die built-in Lösung von Java), und das „Legion of the Bouncy Castle package“(BC) [Thc]. Die BC-Implementationen sind fast immer etwas schneller als die der JCE, jedoch in ihrer Funktionalität meist nicht ganz so umfangreich.

Falls beide Pakete die gewünschte Funktionalität zur Verfügung stellten, wurde für die vorliegende Arbeit nach einigen Tests immer die schnellere beibehalten. Die Tabellen 1 und 2 bieten einen Überblick über die jeweils benutzen Implementierungen.

Tab. 1: Kryptographische Verfahren im SWP Schema

Funktionalität	Provider
Suchworte vorverschlüsseln $E(W)$	Bouncy Castle (AES)
Schlüssel für „controlled searching“ berechnen	JCE (HMAC-MD5)
Padding der Klartextworte auf Länge n	Bouncy Castle (PKCS7 Padding)
Zufallsgenerator G	JCE (SecureRandom, SHA1PRNG)
(Hash-)Werte $F(S)$ berechnen	JCE (HMAC-MD5)

Tab. 2: Kryptographische Verfahren im SUISE Schema

Funktionalität	Provider
Dateien verschlüsseln	Bouncy Castle (AES)
Such-Token r_w erzeugen	JCE (HMAC-SHA1)
Zufallsorakel H	JCE (HMAC-SHA1)
Zufallsgenerator G	JCE (SecureRandom, SHA1PRNG)

Zur Anbindung an Cassandra kommt der Java Treiber 2.1 [Da] zum Einsatz, der die Nutzung der aktuellen Cassandra Query Language (CQL3) erlaubt. HBase bietet als schnellste Möglichkeit der Interaktion eine native Java API, die aber ohne die Highlevel-Optionen

einer Anfragesprache wie CQL auskommen muss. Hier werden alle Anfragen auf die drei Standardoperationen put, get und delete zurückgeführt, die mit sog. Filtern dann direkt an die Tabellen und Keyspaces repräsentierenden Objekte übergeben werden.

5 Benchmarks

Das folgende Kapitel quantifiziert die Performance beider Schemata in Kombination mit beiden Datenbanken. Darüber hinaus werden alle Tests auch ohne Datenbanken auf dem bloßen Dateisystem ausgeführt, um zu ermitteln, welchen Geschwindigkeitsvorteil der Einsatz einer Datenbank überhaupt bringt. Hardwarebasis ist ein Intel Core i7-4600 CPU (2,1 GHz), 8 GB RAM und einer Samsung PM851 SSD. Darauf läuft Ubuntu 14.04 LTS, Java 8 (64bit), Apache Cassandra in Version 2.1.2, der Cassandra Java Treiber in Version 2.1, CQL 3.1.7, sowie HBase in Version 0.94.20.

5.1 Der Datensatz

Der Testdatensatz ist eine Teilmenge des TREC 2005 Spam Track Public Corpus [CL05], einer Sammlung von annähernd 100.000 E-Mails, die in ihrer jeweiligen Größe von vorwiegend einigen Kilobytes bis hin zu mehreren Megabytes variieren. Um den Anwendungsfall einer Mailbox zu simulieren, beginnen wir unsere Tests jeweils mit den ersten 1.000 Mails des Corpus und erhöhen schrittweise bis auf 10.000 Mails. Dabei wächst die Größe des Klartextes von ca. 700.000 auf 7 Millionen Worte an. Zählt man gleiche Worte pro Dokument nur einmal, beträgt die Menge der Klartextworte jeweils nur immer ca. 40%, konkret 273.204 Worte bei 1.000 E-Mails und knapp 3 Millionen Worte bei 10.000 E-Mails. Dies ist wichtig für die Einschätzung der Ergebnisse des SUISE Schemas, da dieses nur eben diese 40% des Datensatzes berücksichtigen muss, während das SWP Schema eine komplette Verschlüsselung des Datensatzes vornimmt.

5.2 Test 1: Verschlüsseln

Der erste Test misst die zur Verschlüsselung benötigte Zeit, inklusive Speicherung der Daten in der jeweiligen Datenbank oder dem Dateisystem. Diese bestehen bei SWP nur aus den verschlüsselten Dateien selbst, während bei SUISE noch die Indizes γ_f und γ_w hinzukommen. Für SWP wird die Wortlänge auf $n = 8$ festgelegt (siehe Abschnitt 5.4).

Abbildung 1 zeigt die Ergebnisse. Da beide Algorithmen einmal über den kompletten Testdatensatz iterieren müssen, wächst der zeitliche Aufwand dafür linear. SUISE ist dabei unabhängig von der darunterliegenden Datenbank schneller als SWP, obwohl komplexere Schritte zur Verschlüsselung erforderlich sind. Wie oben erwähnt ist die Ursache hierfür in der Charakteristik des Testdatensatzes zu suchen, denn wie oben beschrieben muss SUISE nur 40% der Klartextworte tatsächlich verschlüsseln.

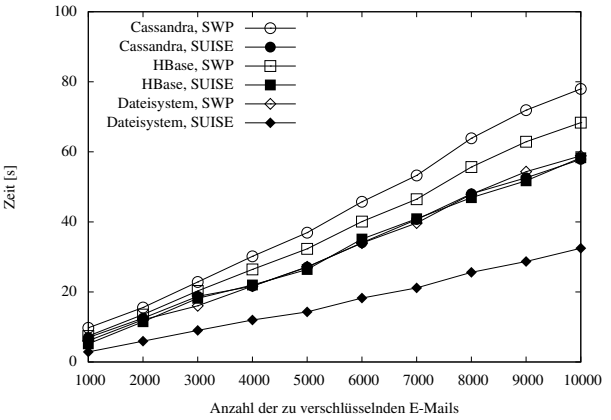

Abb. 1: Zur Verschlüsselung benötigte Zeit mit zunehmender Größe des Datensatzes

Beide Algorithmen sind schneller, wenn sie nur auf dem Dateisystem operieren, statt ihre Ergebnisse in einer Datenbank unterzubringen. Ursächlich kann angenommen werden, dass direktes Schreiben in Dateien zügiger ist, als im Falle von Cassandra erst entsprechende Insert/Update-Queries zu erstellen und auszuführen bzw. im Falle von HBase erst die entsprechenden Tabellen- und Filterobjekte zu erzeugen. Das Rennen unter den Datenbanken entscheidet HBase für sich, wenn auch nicht signifikant.

5.3 Test 2: Suchen

Der zweite Test misst die Zeit für den Suchprozess. Für einen fairen Vergleich beider Schemata wird hier aber der SWP Algorithmus insofern modifiziert, als dass er die Suche nach dem ersten Treffer in einem Dokument abbrechen darf. Damit liefert er die gleiche Information wie SUISE (nämlich ob ein Suchwort in einem Dokument vorkommt, oder nicht). Die Fähigkeit des SWP Algorithmus im Gegensatz zu SUISE eigentlich auch Anzahl und Position von Treffern pro Dokument zu liefern, würde ihm sonst zum Nachteil werden.

Abbildung 2 zeigt die Ergebnisse. Ausdrücklich angemerkt sei hier, dass im Falle von SUISE zum ersten mal nach dem Suchwort gesucht wird, also noch kein entsprechender Eintrag in γ_w vorliegt, mit dem das Suchergebnis sofort zurückgegeben werden könnte. Das resultiert ähnlich zu SWP in linear wachsender Zeit mit wachsender Datensatzgröße. Darüber hinaus macht es im Falle von SUISE in Kombination mit der Operation auf dem Dateisystem einen signifikanten Unterschied, ob die Indizes γ_f und γ_w bereits im Hauptspeicher sind (SUISE(2)), oder nicht (SUISE(1)). SUISE(2) kann somit auch als Simulation eines konstant laufenden Servers betrachtet werden, welcher die Indizes permanent im Hauptspeicher hält.

Im Gegensatz zu den Resultaten aus Test 1 ist SUISE nicht in jedem Fall besser. Der wie oben beschrieben modifizierte SWP-Algorithmus findet die gesuchten Dokumente schneller, es sei denn Cassandra wird als darunterliegende Technologie eingesetzt. Eine

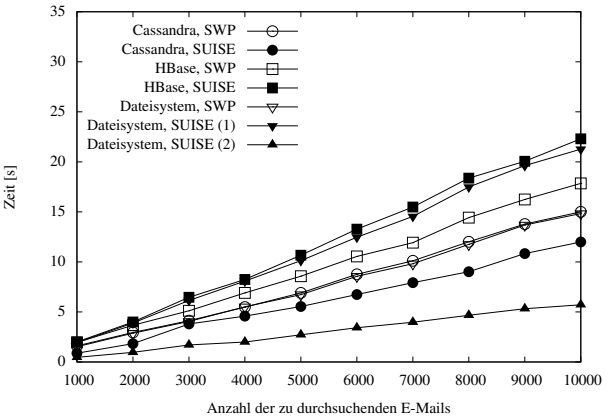

Abb. 2: Zur Suche benötigte Zeit mit zunehmender Größe des Datensatzes

Ausnahme ist die wiederholte Suche nach dem gleichen Suchwort. SUISE kann hier auf seinen Index γ_w zurückgreifen und somit in konstanter Zeit Ergebnisse liefern. Die hierfür benötigten Zeitspannen belaufen sich im Durchschnitt und unabhängig von der Datensatzgröße auf 22ms mit Cassandra und 10ms mit HBase. Bei der Operation auf dem Dateisystem und mit den Indizes im Hauptspeicher (also SUISE (2) entsprechend) ist sogar nur <1 ms notwendig. Muss erst der Index in den Hauptspeicher geladen werden (vgl. SUISE(1)) ist keine konstante Suchzeit mehr möglich. Die Suchdauer steigt dann von 1,56s für 1.000 Mails auf 17,32s für 10.000 Mails.

Was den Blick auf die Datenbanken anbelangt, so ist im Gegensatz zu den Ergebnissen beim Verschlüsselungsprozess Cassandra schneller, in Kombination mit SWP um ca. 10% und mit SUISE sogar fast doppelt so schnell verglichen mit HBase.

5.4 Test 3: Optimierung des SWP Schemas

Sowohl SWP als auch SUISE bieten Optimierungspotenzial. Bei SWP ist der wichtigste Parameter die eingesetzte Wortlänge n. Worte mit einer Länge kleiner n müssen via Padding künstlich vergrößert werden. Worte mit einer Länge größer als n müssen (mehrfach) gesplittet werden. Damit führt Padding zu mehr Bytes und Splitting zur mehr Worten im Geheimtext (und damit zu mehr benötigten Iterationen des Verschlüsselungs- und Suchalgorithmus). Das bedeutet in der Praxis: kleine n führen zu Performanceverlust, aber sparen Speicherplatz. Große n resultieren in mehr Speicherplatzverbrauch, aber können Verschlüsselung und Suche beschleunigen, da weniger Iterationen notwendig sind.

Abbildung 3 zeigt den Einfluss der wachsenden Wortlänge von $n = 4$ bis $n = 9$. Während der Speicherplatzbedarf wie erwartet mit wachsendem n linear zunimmt, stellt sich ab $n = 8$ keine signifikante Verbesserung der Performance mehr ein. $n = 8$ scheint somit der beste Kompromiss zu sein und damit geeignet als Parameter für die vorangegangenen Tests. Die optimale Wahl von n ist natürlich individuell vom konkreten Datensatz abhängig.

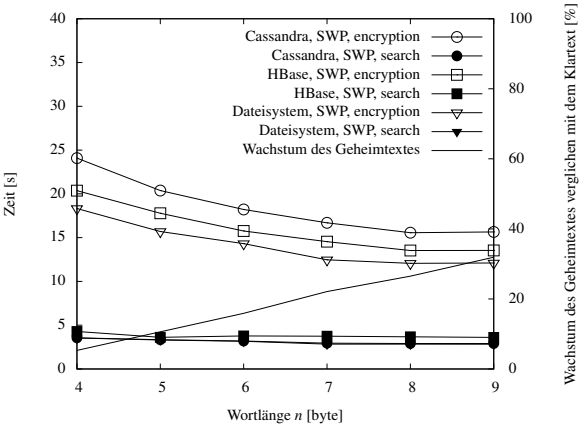

Abb. 3: Performance und Speicherplatzbedarf von SWP mit steigender Wortlänge

5.5 Test 4: Optimierung des SUISE Schemas

Die vorangegangenen Tests demonstrierten weitestgehend den Geschwindigkeitsvorteil des SUISE Schemas, der jedoch den Preis der Aufrechterhaltung zweier potenziell großer Indizes hat. Während über γ_w schwer allgemeine Aussagen getroffen werden können, da dessen Größe maßgeblich vom Suchmuster abhängt, wächst γ_f im Vergleich zum Klartext sehr schnell. Die Ursache dafür liegt in der Art und Weise, wie die Repräsentationen der Klartextworte zustande kommen, die in γ_f gespeichert werden, wie oben beschrieben nämlich $H_{r_w}(s)||(s)$. Wird wie von den Autoren vorgeschlagen [HK14] zur Umsetzung von H HMAC-SHA1 benutzt, schlägt der erste Teil bereits mit 20 Bytes zu Buche. Dazu kommt die Outputlänge des Zufallsgenerators G, der s generiert. Der Empfehlung der Autoren folgend fallen auch dafür 20 Bytes an. Das bedeutet in der Praxis: selbst wenn ein Klartextwort nur ein Byte lang ist, belegt es 40 Bytes in γ_f. Somit wird γ_f schnell sehr groß, obwohl gleiche Worte immer nur einmal pro Dokument dort abgelegt werden.

Abbildung 4 zeigt die tatsächliche Größe von γ_f abhängig von der darunterliegenden Datenbank oder dem Dateisystem. Die Größe des Testdatensatzes beträgt dabei konstant 2.000 E-Mails. γ_f wächst erwartungsgemäß linear mit der Outputlänge von G (also der Länge von s). Den empfohlenen Werten der Autoren folgend wird γ_f schon mehr als doppelt so groß als der komplette Klartext. In der Praxis kommen dazu noch die verschlüsselten Daten selbst. Verheerend ist die Situation, wenn γ_f in HBase abgelegt wird, wo die interne Ablage des Index besonders viel Platz benötigt, teilweise bis zum 5-fachen der Klartextgröße. Eine Verringerung der Outputlänge von G verringert den Platzbedarf, aber nicht signifikant. Ohne dafür gesondert eine Abbildung zu bemühen, sei des Weiteren angemerkt, dass verschiedene Outputlängen von G keinen Einfluss auf die Geschwindigkeit des SUISE Schemas haben.

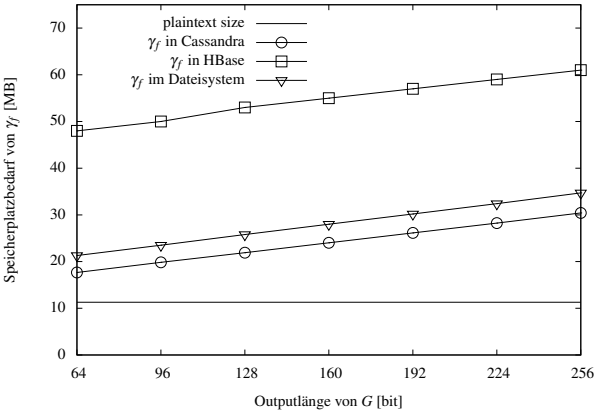

Abb. 4: Größe von γ_f mit wachsender Länge von s

5.6 Zusammenfassung

Die vorangegangenen Tests zeigen Stärken und Schwächen, sowohl der Schemata für durchsuchbare Verschlüsselung, als auch der darunterliegenden NoSQL-Datenbanken selbst. Während der *Verschlüsselung* ist der SUISE Algorithmus schneller, um den Faktor 1,2 bis 1,4 auf den Datenbanken, um den Faktor 1,8 auf dem Dateisystem. Ein ähnlich klares Ergebnis ist beim *Suchen* nicht festzustellen. Eine Ausnahme sind Anwendungsfälle, in denen mit hoher Wahrscheinlichkeit mehrfach nach denselben Worten gesucht wird. Hier ist SUISE dank seines Index γ_W zu bevorzugen.

Ansonsten kann in Anbetracht der Ergebnisse kaum eine allgemein gültige Empfehlung ausgesprochen werden. SUISE ist fast immer schneller, benötigt aber potenziell sehr große Indizes. Ist von kleineren Datensätzen auszugehen und der Speicherplatz eher knapp bemessen, so liefert auch das SWP Schema im Vergleich eine gute Performance. Dieses hat außerdem den Vorteil, dass es aufgrund seines simplen Designs einfach in Client-Server-Anwendungen umzusetzen ist.

Aus Sicht der Datenbanken ist festzuhalten, dass HBase schneller ist beim Verschlüsseln und Cassandra beim Suchen, letzteres besonders wenn der SUISE Algorithmus zum Einsatz kommt. Eine Ausnahme bildet die wiederholte Suche nach Worten mit SUISE, wo sich ein umgekehrtes Bild ergibt. Es kann außerdem beobachtet werden, dass sich zumindest für den Prozess der Verschlüsselung kein Geschwindigkeitsvorteil aus der Nutzung eines der beiden Datenbanksysteme ergibt.

6 Verwandte Arbeiten

Zu den praktischen Aspekten durchsuchbarer Verschlüsselung gibt es kaum verwandte Arbeiten. Eine Ausnahme ist [Po12], in der das SWP Schema für relationale Datenban-

ken aufgegriffen wird. Dabei gäbe es noch einige andere Schemata zu untersuchen. Eine exzellente Übersicht bietet [Bö14].

Bereits die Autoren des SWP Schemas schlagen die Nutzung eines Index für ihr Schema vor, gehen aber nicht im Detail darauf ein. Viele Nachfolgearbeiten basierten dann auf einem invertierten Index, beispielsweise [Cu06, CK10, Ca13]), da dieser auch noch effizient sein kann, wenn er nicht mehr in den Hauptspeicher passt [Ca14] (vgl. Abschnitt 5.5). Sind die zu speichernden Daten sehr dynamisch, ist dies eine besondere Herausforderung für den Index [KPR12, Ca14], während sequential-scan-Verfahren wie der SWP-Algorithmus damit naturbedingt keinerlei Probleme haben.

7 Einbettung in das übergeordnete Thema der Dissertation

Die vorliegende Arbeit zeigt, dass es mit jeweils vollkommen unterschiedlichen Ansätzen möglich ist, Suchvorgänge in verschlüsselten NoSQL-Datenbanken effizient durchzuführen. Beide Schemata erreichen bereits auf gewöhnlicher Notebookhardware ca. 100.000 Worte pro Sekunde beim Verschlüsseln und über 500.000 Worte pro Sekunde beim Durchsuchen.

Ziel der Dissertation ist es jedoch, nicht nur in verschlüsselten Daten in NoSQL-Datenbanken suchen zu können, sondern in Anlehnung an [Po12] auch andere Funktionalitäten auf ausschließlich verschlüsselten Daten bereitzustellen, welche die Query-Mechanismen dieser Datenbanken naturgemäß zu Verfügung stellen. Dazu gehört beispielsweise die Nutzung von deterministischer Verschlüsselung für Gleichheitsabfragen, Gruppierungen etc., sowie ordnungsbewahrender Verschlüsselung für sog. Range Scans, Sortierungen, dem Finden von Minima und Maxima etc. Letztere ist insbesondere für Rowkeys (zur Gewährleistung der Lokalität der Daten) und Timestamps (zur Versionierung) von Bedeutung.

Die größte Herausforderung dabei ist es, die Datenbanken an sich unverändert zu belassen. Interna wie Kompaktierungen, Bloomfilter, sog. Tombstones (Löschmarkierungen), sekundäre Indizes und andere Mechanismen sollen ungeachtet des verschlüssselten Inhalts weiterhin uneingeschränkt funktionieren.

Diese Arbeit wurde von der DFG gefördert. Förderkennzeichen: WI 4086/2-1.

Literaturverzeichnis

[Bo11] Borthakur, Dhruba; Gray, Jonathan; Sarma, Joydeep Sen; Muthukkaruppan, Kannan; Spiegelberg, Nicolas; Kuang, Hairong; Ranganathan, Karthik; Molkov, Dmytro; Menon, Aravind; Rash, Samuel et al.: Apache Hadoop goes realtime at Facebook. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, S. 1071–1080, 2011.

[Bö14] Bösch, Christoph; Hartel, Pieter; Jonker, Willem; Peter, Andreas: A survey of provably secure searchable encryption. ACM Computing Surveys (CSUR), 47(2):18, 2014.

[Br00] Brewer, Eric A: Towards robust distributed systems. In: PODC. S. 7, 2000.

[Br10] Brewer, Eric: A certain freedom: thoughts on the CAP theorem. In: Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing. ACM, S. 335–335, 2010.

[Ca13] Cash, David; Jarecki, Stanislaw; Jutla, Charanjit; Krawczyk, Hugo; Roşu, Marcel-Cătălin; Steiner, Michael: Highly-scalable searchable symmetric encryption with support for boolean queries. In: Advances in Cryptology–CRYPTO 2013, S. 353–373. Springer, 2013.

[Ca14] Cash, David; Jaeger, Joseph; Jarecki, Stanislaw; Jutla, Charanjit; Krawczyk, Hugo; Rosu, Marcel-Catalin; Steiner, Michael: Dynamic searchable encryption in very large databases: Data structures and implementation. In: Proceedings of the Network and Distributed System Security Symposium (NDSS). Jgg. 14, 2014.

[CK10] Chase, Melissa; Kamara, Seny: Structured encryption and controlled disclosure. In: Advances in Cryptology-ASIACRYPT 2010, S. 577–594. Springer, 2010.

[CL05] Cormack, Gordon V; Lynam, Thomas R: TREC 2005 Spam Track Overview. In: TREC. 2005.

[Cu06] Curtmola, Reza; Garay, Juan; Kamara, Seny; Ostrovsky, Rafail: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of the 13th ACM conference on Computer and communications security. ACM, S. 79–88, 2006.

[Da] Datastax: Java Driver 2.1 for Apache Cassandra. `http://www.datastax.com/documentation/developer/java-driver/2.1/common/drivers/introduction/introArchOverview.html`, Eingesehen am 04.05.2015.

[HK14] Hahn, Florian; Kerschbaum, Florian: Searchable Encryption with Secure and Efficient Updates. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, S. 310–320, 2014.

[KPR12] Kamara, Seny; Papamanthou, Charalampos; Roeder, Tom: Dynamic searchable symmetric encryption. In: Proceedings of the 2012 ACM conference on Computer and communications security. ACM, S. 965–976, 2012.

[LM10] Lakshman, Avinash; Malik, Prashant: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, 44(2):35–40, 2010.

[Po12] Popa, Raluca Ada; Redfield, Catherine; Zeldovich, Nickolai; Balakrishnan, Hari: CryptDB: Processing queries on an encrypted database. Communications of the ACM, 55(9):103–111, 2012.

[So] SolidIT: DB-Engines Ranking. `http://db-engines.com/en/ranking`, Eingesehen am 04.05.2015.

[SWP00] Song, Dawn Xiaoding; Wagner, David; Perrig, Adrian: Practical techniques for searches on encrypted data. In: Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on. IEEE, S. 44–55, 2000.

[Tha] The Apache Software Foundation: Apache Cassandra. `http://cassandra.apache.org`, Eingesehen am 04.05.2015.

[Thb] The Apache Software Foundation: Apache HBase. `http://hbase.apache.org`, Eingesehen am 04.05.2015.

[Thc] The Legion of the Bouncy Castle. `http://bouncycastle.org/`, Eingesehen am 04.05.2015.