Indexe auf verschlüsselten Daten: Konzepte, Performanz, Sicherheitsaspekte

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Jan Lehnhardt

aus

Überlingen am Bodensee

Bonn, 2017

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn.

Erstgutachter:	Prof. Dr. Armin B. Cremers, Bonn
Zweitgutachter:	Prof. Dr. Rainer Manthey, Bonn
Tag der Promotion:	27.03.2019
Erscheinungsjahr:	2019

Überblick

Diese Dissertation befasst sich mit Informationssystemen mit nicht vertrauenswürdiger Serverseite. Für solche Systeme bietet sich eine clientseitige Verschlüsselung ihrer Daten an, um hohe Anforderungen an deren Vertraulichkeit erfüllen zu können. Gleichzeitig soll trotz der Verschlüsselung hohe Performanz bei Zugriff und Verarbeitung der Daten gewährleistet werden. Bezüglich dieser Problemstellung leistet die Dissertation den Beitrag der ausführlichen Beschreibung, Ausarbeitung und Bewertung verschiedener neuartiger, ebenfalls verschlüsselter Indexierungstechniken und Indexstrukturen, die die häufigsten Zugriffstypen in Informationssystemen bedienen. Diese Strukturen ermöglichen einen sinnvollen Kompromiss zwischen Vertraulichkeit und Performanz, nicht zuletzt vor dem Hintergrund der in der Bundesrepublik Deutschland geltenden Datenschutzgesetze. Zunächst werden Indexstrukturen für die Gleichheitssuche auf verschlüsselten Daten behandelt. Bezüglich der im SQL-Standard definierten Datentypen herrschen bezüglich des Gleichheitsbegriffs mitunter unterschiedliche Definitionen, weshalb auch unterschiedliche Ausprägungen der Indexstrukturen präsentiert werden. Weiterhin wird mit dem *cTree* eine verschlüsselte Datenstruktur eingeführt, die Ungleichheits- und Intervallsuchen auf linear geordneten Daten ermöglicht. Auf lexikografisch geordneten Daten ermöglicht der cTree weiterhin Präfixsuchen, die sich zu flexibel parametrisierbaren Infixsuchen erweitern lassen. Verschiedene Modifikationen der cTree-Datenstruktur, insbesondere client- und serverseitiges Caching sowie parallele Ausführung und ein leichtgewichtiges transaktionales Konzept, ermöglichen signifikante Steigerungen ihrer Performanz und Praxistauglichkeit. Auf der anderen Seite exponieren die Indexstrukturen Teile des Klartextgehaltes der indexierten Daten und können so ungewollten Datenverlust verursachen. Modifikationen wie die Vermeidung deterministischer Verschlüsselung oder die Verschlüsselung von Fremdschlüsselbeziehungen schaffen hier Abhilfe, indem sie unter Inkaufnahme von Performanzeinbußen den Informationsverlust vermeiden bzw. signifikant abschwächen. Die Performanz der Indexstrukturen in ihrer Grundform sowie ihrer Hochsicherheitsvariante wird in einer Testreihe auf einer praktischen Implementierung ermittelt und mit konventionellen Datenbankindexen verglichen, die unverschlüsselte Datenbankattribute indexieren. Die Ergebnisse zeigen, dass die Indexstrukturen beim Entwurf eines Informationssystems flexibel an dessen Bedürfnisse bezüglich Vertraulichkeit und Performanz angepasst werden können und belegen somit den praktischen Nutzen des wissenschaftlichen Beitrags dieser Arbeit.

Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung[1]

Informationssysteme, bei denen der Speicherort und die Verarbeitungskapazität (oder zumindest große Teile derselben) in entfernten, über das Internet erreichbaren Rechenzentren angesiedelt sind, haben stark an Bedeutung gewonnen. Der Begriff „Cloud Computing" hat sich etabliert und nimmt in vielen Produktstrategien eine prominente Rolle ein. Im Folgenden wird bezüglich solcher Informationssysteme von *Cloud-Informationssystemen* (*CIS*) gesprochen.

1.1 „…as a service"

Im Bereich des kommerziellen Cloud-Computings werden heutzutage neben der Speicherung viele der Datenverarbeitungsoperationen eines Informationssystems als Dienstleistung angeboten, so dass sich Produktkategorien wie „Software as a service", „Platform as a service" und „Infrastructure as a service" etabliert haben. Allen ist gemein, dass sie Speicher-, Verfügbarkeits- und Verarbeitungskapazität in entfernten Informationssystemen bereitstellen, ohne signifikante Performanzverluste gegenüber lokalen Systemen in Kauf nehmen zu müssen. So bieten viele Hersteller Cloud-Dienste an wie etwa Google Drive[2], Microsoft OneDrive[3], Apple iCloud[4], ownCloud[5], Seafile[6] und Dropbox[7].

Im Widerspruch zur hohen Verbreitung von CIS steht jedoch ein gravierender Nachteil dieser Systeme in Bezug auf deren Sicherheit, insbesondere der Vertraulichkeit der gespeicherten Daten, der im Folgenden näher erläutert wird. Der wissenschaftliche Beitrag dieser Dissertation befasst sich mit der Behebung bzw. Eindämmung dieses Nachteils.

In der üblichen CIS-Architektur handelt es sich beim Client um eine relativ wenig umfangreiche, beispielsweise in HTML/Javascript entwickelte Softwarekomponente, die in einem Webbrowser ausgeführt und dargestellt wird. Die Serverseite (beispielsweise bestehend aus einem Application Server und einem Datenbankmanagementsystem (*DBMS*)), auf der der größere Teil der Systemfunktionalität angesiedelt ist, speichert und verarbeitet die Daten an einem Ort, der sich der faktischen Kontrolle der CIS-Anwender entzieht. Oft hat der CIS-Anwender nicht einmal die Kontrolle über die Auswahl des Ortes, an dem seine Daten verarbeitet werden (etwa in welchem Land sich das betreffende Rechenzentrum befindet und welcher Rechtsprechung die verarbeiteten Daten damit unterliegen). Stattdessen wird dies durch den Hersteller des CIS diktiert, oder durch den Anbieter, der das Rechenzentrum betreibt, in dem der Server gehostet wird.

[1] Die vorliegende Dissertation wurde nach den Regeln der deutschen Rechtschreibung verfasst.
[2] siehe [URL-GoogleDrive].
[3] siehe [URL-OneDrive].
[4] siehe [URL-ICloud].
[5] siehe [URL-OwnCloud].
[6] siehe [URL-Seafile].
[7] siehe [URL-Dropbox].

1.2 Vertraulichkeit

In Bezug auf den Schutz der Vertraulichkeit[8] der in einem CIS gespeicherten Daten ist deren Sicherheit während ihrer Darstellung im Client stark abhängig von der Absicherung des Client-Betriebssystems gegen Schadsoftware. Dieser Teilaspekt der Sicherheit von Informationssystemen liegt jedoch nicht im Fokus dieser Dissertation und wird im Folgenden als gegeben angenommen. Auch Techniken zur Absicherung der Client-Server-Kommunikation stellen kein großes Problem dar: Mit performanten Algorithmen verschlüsselte Verbindungen, die ad hoc ausgehandelte Sitzungsschlüssel verwenden, wie dies etwa im TLS-Protokoll der Fall ist, sind etablierter Standard der Netzwerksicherheit; sie liegen ebenfalls außerhalb des Fokus dieser Dissertation.

Bei der Verarbeitung und Speicherung der Daten auf der Serverseite ist es dagegen gängige Praxis, dass in den CIS-Betrieb involvierten Personen weitreichender Zugriff auf die unverschlüsselten Daten eingeräumt wird, obwohl dafür keine unmittelbare, anwendungsbezogene Notwendigkeit besteht. Ein im Rechenzentrum angestellter System- oder Datenbankadministrator hat etwa oft direkten, weitreichenden Zugriff auf die im CIS gespeicherten Daten, obwohl ein explizites *need-to-know* (siehe Kapitel 3.2.2) für diese Art des Zugriffs angesichts seines Aufgabenprofils bezüglich des CIS fehlt.

Offensichtlich besteht hier die Gefahr von *social engineering*-Angriffen, bei denen versucht wird, solche mit weitreichenden Privilegien ausgestatteten Personen zu korrumpieren, beispielsweise über Bestechung oder Erpressung. Diese Gefahr mag abgemildert werden können, indem der Rechenzentrumsbetreiber seinen Mitarbeitern strenge administrative Regeln für den Umgang mit den anvertrauten Kundendaten auferlegt und juristische Konsequenzen für Privilegienmissbrauch androht; grundsätzlich eliminiert wird sie dadurch jedoch nicht und dürfte bei hohem Sensitivitätsgrad der Daten entsprechend höher ausfallen.

1.3 Rechtlicher Rahmen bei personenbezogenen Daten

Werden personenbezogene Daten in einem CIS verarbeitet, kann bereits allein die Delegierung der Datenverarbeitung, mit oder ohne Schadensfall, rechtliche Konsequenzen haben, wie im Folgenden erläutert wird.

1.3.1 § 11 BDSG

Die Datenverarbeitungsdienstleistung durch einen Auftragnehmer, wie sie durch ein CIS erbracht wird, wird im Kontext des Datenschutzgesetzes der Bundesrepublik Deutschland (*Bundesdatenschutzgesetz*, *BDSG*) als *Auftragsdatenverarbeitung* bezeichnet. § 11(1) BDSG besagt hierzu: „*Werden personenbezogene Daten im Auftrag durch andere Stellen erhoben, verarbeitet oder genutzt, ist der Auftraggeber für die Einhaltung der Vorschriften dieses Gesetzes und anderer*

[8] Die beiden anderen Hauptziele der Informationssicherheit, Verfügbarkeit und Integrität, stehen im Vergleich zur Vertraulichkeit viel weniger bzw. gar nicht im Fokus dieser Dissertation. Methoden zu ihrer jeweiligen Sicherstellung sind gut erforscht, und die entsprechenden Verfahren sind auch im CIS-Umfeld anwendbar, so dass hier kaum Forschungsbedarf besteht. So kann Verfügbarkeit auf der Serverseite durch etablierte Verfahren sichergestellt werden, wie etwa Clustering, Caching, Load Balancing, etc. Einen wirksamen Schutz vor einer Verletzung der Integrität der im CIS gespeicherten Daten kann es dagegen zwar kaum geben, aber für die zuverlässige Feststellung einer Verletzung der Integrität stehen ebenfalls Verfahren wie etwa (hashbasierte) Message Authentication Codes (*MAC*, *HMAC*) oder digitale Signaturen zur Verfügung.

Vorschriften über den Datenschutz verantwortlich“[9]. Weiterhin heißt es in § 11(2), dass „*der Auftragnehmer (…) unter besonderer Berücksichtigung der Eignung der von ihm getroffenen technischen und organisatorischen Maßnahmen sorgfältig auszuwählen*“ ist, und außerdem: „*Der Auftraggeber hat sich vor Beginn der Datenverarbeitung und sodann regelmäßig von der Einhaltung der beim Auftragnehmer getroffenen technischen und organisatorischen Maßnahmen zu überzeugen*.“[10]

Dem Auftraggeber der Datenverarbeitung werden also Pflichten zum Schutz der verarbeiteten personenbezogenen Daten auferlegt. Dies bewirkt, dass er für einen Schadensfall mit Vertraulichkeitsverlust haftbar gemacht werden kann, selbst wenn der Schaden nicht von ihm verursacht wurde.

Darüber hinaus besagt § 4 BDSG: „*Die Erhebung, Verarbeitung und Nutzung personenbezogener Daten sind nur zulässig, soweit dieses Gesetz oder eine andere Rechtsvorschrift dies erlaubt oder anordnet oder der Betroffene eingewilligt hat.*“[11] Dies bedeutet, dass etwa ein Arzt, der seine Arbeitsprozesse in einem cloudbasierten System abbildet, nur dann im Sinne des BDSG handelt, wenn er von jedem Patienten eine Einverständniserklärung verlangt, bevor er ihn behandelt. Verweigerte ein Patient diese Einverständniserklärung, könnte der Arzt ihn nicht behandeln, ein Umstand, der Letzteren in der Gestaltung seiner Geschäftsprozesse einengt.

Andererseits führt das BDSG an mehreren Stellen aus, dass die Maßnahmen zum Schutz von Daten und deren Schutzbedürfnis in einem angemessenen Verhältnis stehen sollten. So heißt es etwa in § 9 S. 2 BDSG (Technische und organisatorische Maßnahmen): „*Erforderlich sind Maßnahmen nur, wenn ihr Aufwand in einem angemessenen Verhältnis zu dem angestrebten Schutzzweck steht.*“ und in § 4f Abs. 2 BDSG (Beauftragter für den Datenschutz): „*Zum Beauftragten für den Datenschutz darf nur bestellt werden, wer die zur Erfüllung seiner Aufgaben erforderliche Fachkunde und Zuverlässigkeit besitzt. Das Maß der erforderlichen Fachkunde bestimmt sich insbesondere nach dem Umfang der Datenverarbeitung der verantwortlichen Stelle und dem Schutzbedarf der personenbezogenen Daten, die die verantwortliche Stelle erhebt oder verwendet.*“

Diese Aussage hat eine hohe Bedeutung für die Bewertung der Sicherheit der in dieser Arbeit vorgestellten Indexstrukturen (siehe Kapitel 10).

1.3.2 § 203 StGB

Unter Umständen können gesetzliche Bestimmungen die Speicherung von Daten auf entfernten Servern komplett untersagen. Das *Strafgesetzbuch der Bundesrepublik Deutschland* (*StGB*) nennt in § 203 eine Reihe von Berufsgruppen wie beispielsweise Ärzte, Steuerberater oder Rechtsanwälte und klassifiziert deren Angehörige als *Geheimnisträger*. Weiterhin werden „*Amtsträger, für den öffentlichen Dienst besonders Verpflichtete, Personen, die Aufgaben oder Befugnisse nach dem Personalvertretungsrecht wahrnehmen*“ und weitere Personengruppen als Geheimnisträger genannt. Wenn ein solcher Geheimnisträger ein ihm im Rahmen seiner Tätigkeit anvertrautes Geheimnis unbefugt offenbart, kann er laut § 203 StGB mit einer Freiheitsstrafe von bis zu einem Jahr oder mit einer Geldstrafe bestraft werden[12].

[9] siehe [URL-BDSG-11].
[10] siehe [URL-BDSG-11].
[11] siehe [URL-BDSG-4].
[12] siehe [URL-StGB-203].

§ 203 StGB folgend kommt beispielsweise bei einem AIS die Speicherung von ungeschützten Patientendaten durch einen Arzt auf einem entfernten Server in einem Rechenzentrum einer Öffentlichmachung gleich, da im Rechenzentrum Personal beschäftigt ist, das Zugriff auf die Daten nehmen kann. Somit wäre der Arzt haftbar und könnte bestraft werden, unabhängig davon, ob ein Privilegienmissbrauch durch das Rechenzentrumspersonal stattgefunden hat oder nicht. Der Betrieb eines cloudbasierten Informationssystems ist hier also nicht möglich.

1.3.3 Rechtliche Behelfskonstrukte

Die in Kapitel 1.3.1 bereits genannten Einverständniserklärungen sind ein zulässiges, oft genutztes Mittel, um die oben genannte Problematik für CIS-Kunden zu umgehen. Eine Einverständniserklärung wird von den Besitzern der im CIS gespeicherten personenbezogenen Daten, im Falle eines AIS den Patienten, eingeholt, was dem CIS-Kunden (also dem Arzt) erlaubt, die Daten Dritten zugänglich zu machen, wenn auch in eingeschränktem Maße. Der Patient erklärt sich somit einverstanden, dass die Schweigepflicht des Arztes teilweise außer Kraft gesetzt wird. Man könnte also anstelle einer Einverständniserklärung auch von einer Schutzverzichtserklärung des Patienten sprechen.

Diese Vorgehensweise mag rechtlich zulässig sein, bewirkt aber, dass die Datenbesitzer sich dem erhöhten Risiko des Missbrauchs ihrer Daten aussetzen und sich mit dieser Situation dauerhaft abfinden müssen.

Bezüglich § 203 StGB ist derzeit eine Neuregelung dieses Paragrafen angedacht, in der das Verbot der Offenbarung von Privatgeheimnissen durch Geheimnisträger abschwächt ist. So heißt es in einer Stellungnahme des Bundesministeriums für Justiz und für Verbraucherschutz vom 15.02.2017: *„Der Entwurf sieht daher eine Einschränkung der Strafbarkeit nach § 203 StGB vor. Ausdrücklich nicht der Strafbarkeit unterfallen soll zukünftig das Offenbaren von geschützten Geheimnissen gegenüber Personen, die an der beruflichen oder dienstlichen Tätigkeit des Berufsgeheimnisträgers mitwirken, soweit dies für die ordnungsgemäße Durchführung der Tätigkeit der mitwirkenden Personen erforderlich ist. Im Gegenzug sollen diese mitwirkendenden Personen in die Strafbarkeit nach § 203 StGB einbezogen werden. Darüber hinaus werden für Berufsgeheimnisträger strafbewehrte Sorgfaltspflichten normiert, die bei der Einbeziehung dritter Personen in die Berufsausübung zu beachten sind.“*[13]

Zum jetzigen Zeitpunkt ist jedoch unklar, ob und wenn ja, wann § 203 StGB neu geregelt wird. Es ist jedoch fraglich, ob die Neuregelung einen besseren Schutz von Privatgeheimnissen ermöglichen kann.

1.4 Vertraulichkeit durch Verschlüsselung

Diese Dissertation hat das Ziel der Entwicklung von Konzepten für ein zum BDSG und zu § 203 StGB konformes CIS, das ohne die in Kapitel 1.3.3 genannten rechtlichen Hilfskonstrukte auskommt und das unabhängig von einer in der Zukunft liegenden Neuregelung von § 203 StGB betrieben werden kann. Für diesen alternativen Ansatz (im Folgenden: „Vertraulichkeit durch Verschlüsselung“- oder *VdV-Ansatz* genannt) gilt das Paradigma, dass der Schutz der Vertraulichkeit der im CIS gespeicherten Daten nicht bzw. nur ergänzend durch administrative Maßnahmen erreicht wird. Insbesondere soll beim VdV-Ansatz keine sogenannte *vertrauenswürdige dritte Stelle* (*trusted third party*; *TTP*) eine zentrale Rolle in der Si-

[13] siehe [URL-StGB-203-2].

cherheitsarchitektur spielen, indem sie zentrale Aufgaben wie das Berechtigungs- oder Schlüsselmanagement übernimmt. Benutzer des Systems wären gezwungen, stets auf die korrekte Funktionsweise einer solchen TTP zu vertrauen, während sie gleichzeitig serverseitig als Schwachstelle des Systems exponiert würde.

Stattdessen wird im VdV-Ansatz die Vertraulichkeit der Daten durch deren Verschlüsselung sichergestellt. Der Zugriff auf die zur Entschlüsselung notwendigen Schlüssel wird unter strikter Berücksichtigung des jeweiligen need-to-know der am CIS beteiligten Personen geregelt. Anhand der beiden folgenden Abbildungen wird dies näher erläutert.

1.4.1 Konventionelle Architektur

Abb. 1 zeigt eine konventionelle Client-Server-Architektur eines Informationssystems, bei dem die Vertraulichkeit der Daten auf der Serverseite durch administrative Maßnahmen geschützt werden, wie etwa strenge Zugangskontrollen und -beschränkungen sowie Verhaltensregeln für das Personal des Rechenzentrums. Auch Verschlüsselung kann zum Einsatz kommen, etwa in Form eines TLS-Kanals für den Transport der Daten zum Client über das Internet, oder durch die verschlüsselte Persistierung der Daten auf den physikalischen Datenträgern, bzw. als transparente Verschlüsselung (*Transparent Data Encryption*, *TDE*) im DBMS. Da jedoch in allen genannten Fällen die Schlüssel zur Entschlüsselung der Daten innerhalb des Hoheitsbereichs des Servers liegen, mögen diese Formen der Verschlüsselung die Vertraulichkeit der Daten zwar vor externen Angriffen schützen, nicht aber vor den zuvor in Kapitel 1.2 beschriebenen Angriffen korrupter Insider.

Stattdessen steht dem Server die Semantik der Daten offen, um dort komplexe Datenverarbeitungsaufgaben durchführen zu können. Daten und Verarbeitungsergebnisse werden über eine gegebenenfalls verschlüsselte Netzwerkverbindung an den Client übertragen, der die empfangenen Daten weiterverarbeitet bzw. zur Anzeige bringt.

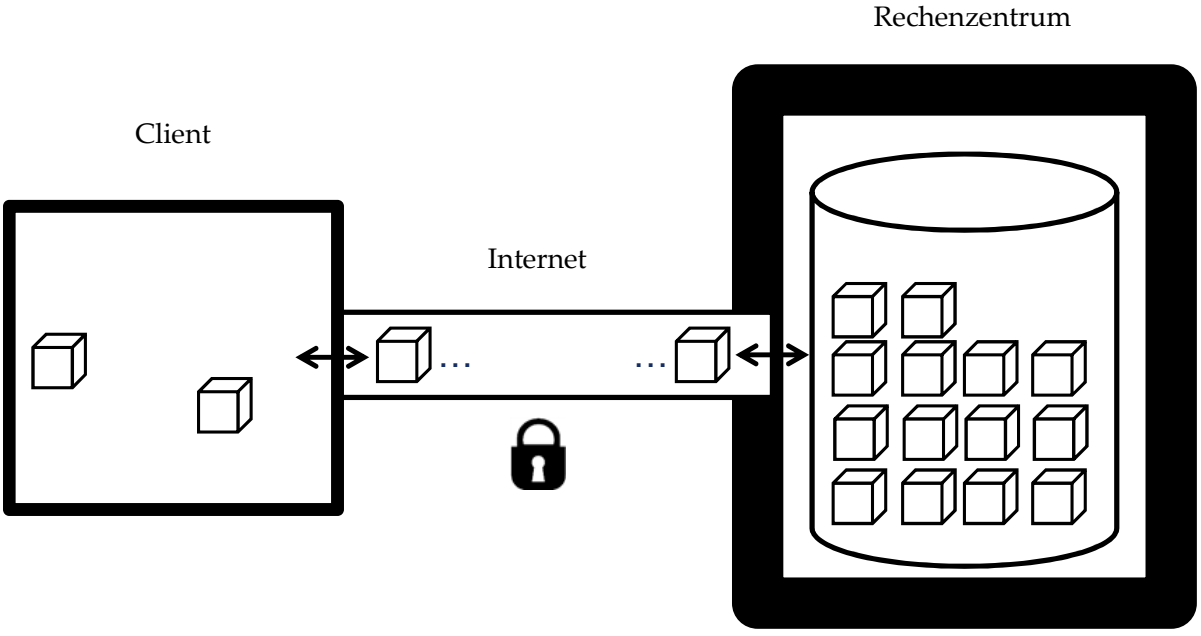

Abb. 1: Herkömmliche Client-Server-Architektur für ein Informationssystem.

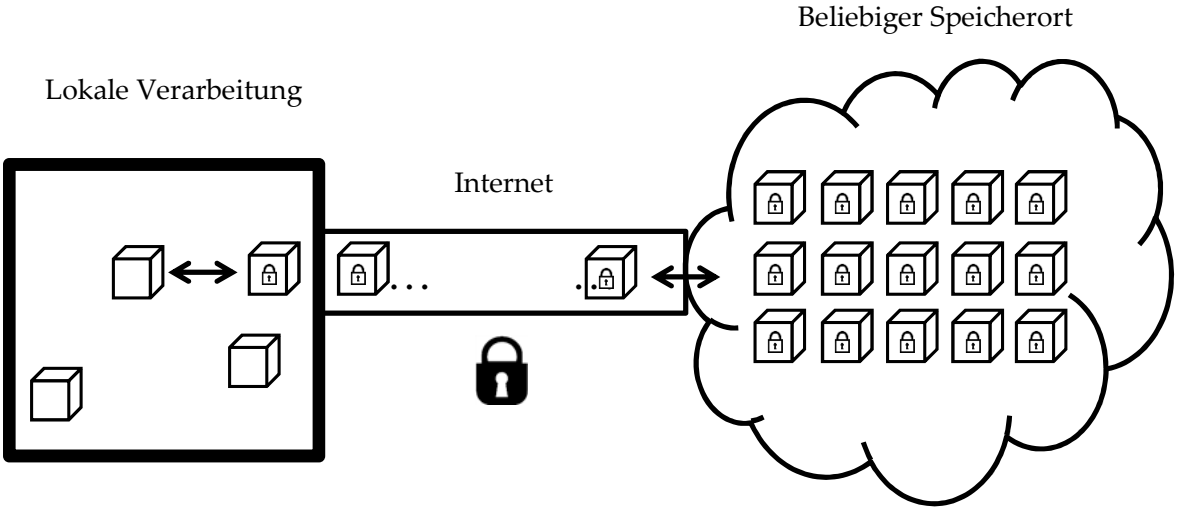

Abb. 2: Client-Server-Architektur mit CTPZ und vollverschlüsseltem Backend.

1.4.2 VdV-Ansatz

In Abb. 2 wird nun die Architektur dargestellt, die dem zuvor eingeführten VdV-Ansatz entspricht. Darin stellt der Client die sogenannte *Klartextverarbeitungszone* (*Cleartext Processing Zone*, *CTPZ*) dar. Sie ist der einzige Ort der gesamten Architektur, an dem die im Informationssystem gespeicherten Daten im Klartext vorliegen: Bevor Daten die CTPZ verlassen, um zum Server transferiert zu werden, werden sie verschlüsselt. Analog dazu werden Daten, nachdem sie vom Server abgerufen und zum Client übertragen worden sind, im Client entschlüsselt und verarbeitet.[14] Weiterhin gilt, dass alle Schlüssel, die zur Ver- und Entschlüsselung von Daten verwendet werden, die CTPZ ebenfalls nicht bzw. nicht im Klartext in Richtung Server verlassen. Stattdessen werden diese Schlüssel entweder direkt durch die CIS-Benutzer ins (Client-)System eingegeben, oder sie werden verschlüsselt vom Server abgerufen und durch weitere Schlüssel im Client entschlüsselt.

Da die Daten außerhalb der CTPZ verschlüsselt sind, ist die Gefährdung der Daten durch korrupte Insider wie beispielsweise Rechenzentrumsadministratoren beseitigt, bzw. erheblich abgeschwächt. Bezüglich der in Kapitel 1.2 genannten Überprüfbarkeit der Integrität der Daten über digitale Signaturen sei festgelegt, dass die Signaturen ausschließlich im Client mit ausschließlich dort im Klartext vorliegenden Schlüsseln auf dem Klartext der Daten berechnet werden. Dies verringert die Gefahr von (unentdeckter) Integritätsverletzung durch Manipulation. Der auf dem Client ausgeführte Code, welcher in einer Webapplikation ebenfalls vom Server geladen wird, sollte ebenfalls bezüglich seiner Integrität überprüft werden. Es soll sichergestellt werden, dass der Code keine unerwünschten Operationen ausführt. Eine gängige Methode hierfür ist die Erstellung einer digitalen Signatur über den Quellcode auf dem Server und deren Verifikation auf dem Client vor dessen Ausführung.

Maßnahmen zur Sicherstellung der Verfügbarkeit der Daten hingegen können aus der konventionellen Architektur übernommen werden.

[14] Damit kann die in Kapitel 1.4.1 genannte Transportverschlüsselung als redundant angesehen werden, da sie den bereits verschlüsselten Daten keine zusätzliche Sicherheit verleiht.

1.4.3 Verschlüsselungsverfahren

Bezüglich der Wahl des im VdV-Ansatz verwendeten Verschlüsselungsverfahrens gelte, dass prinzipiell jedes Verfahren eingesetzt werden kann, so lange es die nötige Resistenz gegenüber gegenwärtigen Methoden der Kryptoanalyse aufweist. Dennoch wird im weiteren Verlauf dieser Dissertation ohne Beschränkung der Allgemeinheit ein konkretes Verfahren genannt, mit dem alle genannten Ver- und Entschlüsselungsoperationen implementiert seien. Aufgrund weiter Verbreitung, hoher Akzeptanz und guter Performanz sei dies das symmetrische Kryptosystem AES-256 im CBC-Modus.

1.4.4 Clientseitige Verarbeitung und Performanz

Eine zentrale Eigenschaft des VdV-Ansatzes ist, dass durch den Einsatz von clientseitig verschlüsselten Daten ein Großteil der Möglichkeiten zu deren serverseitiger Verarbeitung wegfällt: Der Server hat durch die Verschlüsselung keine Kenntnis mehr von Semantik und Struktur der Daten. Stattdessen findet die Verarbeitung zu einem großen Teil im Client statt, was wiederum viele Cloud Computing-Anwendungsformen ausschließt bzw. ihre Flexibilität verringert.[15]

Dennoch gilt auch für den VdV-Ansatz mit clientseitiger Datenverarbeitung das Ziel, die Fähigkeiten der Serverseite, besonders des DBMS, optimal auszunutzen. Vor allem soll eine Standardoperation gängiger Informationssysteme möglich sein: das schnelle Durchsuchen großer Datenbestände nach verschiedenen Kriterien. Der Fokus liege zunächst auf den gängigsten Suchverfahren in kommerziellen, kundenorientierten Informationssystemen, also Gleichheits-, Ungleichheits-, Intervall- und Präfixsuche. Diese Suchtypen sollen durch weitere Suchtypen erweitert und ergänzt werden, etwa Infix- und Volltextsuche.

Ein naheliegendes Beispiel für eine gängige Suchanfrage ist eine solche auf den Patientenstammdaten eines AIS nach Personen, deren Nachname einer vom Benutzer eingegebenen Zeichenkette gleicht oder damit beginnt. Die dazu korrespondierende Treffermenge aus dem gegenwärtigen Zustand der Datenbankextension soll in möglichst kurzer Zeit zum Client transferiert und dem Benutzer präsentiert werden.

Die angestrebte maximale Dauer der gesamten Operation vom Absenden der Anfrage bis zur Präsentation der Ergebnisse sollte idealerweise so kurz sein, dass sie die Arbeitsabläufe des Benutzers nicht signifikant beeinträchtigt. Dabei muss nicht zwingend eine reine Echtzeitdatenverarbeitung beansprucht werden (hierfür wird in [Nielsen1994] eine Obergrenze von 100 ms angegeben: „*0.1 second is about the limit for having the user feel that the system is reacting instantaneously, meaning that no special feedback is necessary except to display the result.*“[16]). Stattdessen ist auch eine höhere Obergrenze für die Antwortzeit vertretbar: [Nielsen1994] beschreibt etwa eine Obergrenze von 1000 ms als „*about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay*“.[17] In dieser Dissertation wird mit

---

[15] Bei der clientseitigen Verarbeitung der Daten im Web-Umfeld wird der erforderliche Programmcode wie die Daten auch von einem entfernten Server geladen. Analog zur Integritätsprüfung der Daten ist auch die Integrität des Programmcodes zu prüfen, etwa über die Verifikation einer mit einem akkreditierten Zertifikat erstellten digitalen Signatur, die über die Gesamtheit des Codes erstellt wurde.

[16] siehe [Nielsen1994], S.135.

[17] siehe [Nielsen1994], S.135.

300 ms ein Schwellwert für die Antwortzeit angesetzt, bei dem der Benutzer die Verzögerung zwar möglicherweise registriert, in seinen Arbeitsabläufen jedoch nicht beeinträchtigt wird, so dass er die Reaktion des Systems als „Quasi-Echtzeit“ wahrnimmt. Auf keinen Fall überschritten werden sollten jedoch die genannten 1000 ms.

In einem auf Klartextdaten operierenden DBMS existieren für eine solche Anforderung an die Reaktionszeit etablierte Lösungen und Hilfsmittel, wobei insbesondere Datenbankindexe zu nennen sind. Sie ermöglichen es, große Datenbankextensionen schnell zu durchsuchen und Ergebnismengen zusammenzustellen.

Auf verschlüsselten Daten sind diese Datenbankindexe jedoch meist nicht einsetzbar, da die inhärente Semantik der zu indexierenden Daten, wie etwa deren Ordnung, für das DBMS nicht mehr erkennbar ist und dementsprechend nicht mehr ausgenutzt werden kann. In einem naiven Ansatz zur Implementierung einer solchen Suchoperation schickte das DBMS die gesamte zu durchsuchende verschlüsselte Extension an den Client, welcher die erhaltenen Tupel entschlüsselt und durchsucht und so die Treffermenge zusammenstellt. Es ist offensichtlich, dass bei diesem Vorgehen eine niedrige Performanz zu erwarten ist.

1.5 Indexe auf verschlüsselten Daten

Diese Dissertation befasst sich mit Lösungsansätzen für Suchoperationen auf verschlüsselten Daten, die ein Ergebnis in (nahezu) Echtzeit und ohne Verletzung der Vertraulichkeit liefern. Verschiedene Indexierungsverfahren wurden erarbeitet, bei denen die Daten mit zusätzlichen, ebenfalls verschlüsselten Strukturdaten versehen auf der Serverseite abgelegt werden. Dabei offenbaren die Strukturdaten dem Server eine kleine Menge an Klartextinformation, die jedoch für die Vertraulichkeit der Daten im jeweiligen zugrundeliegenden Anwendungsszenario unkritisch ist.

Unter Verwendung der Strukturdaten wird dem Server ermöglicht, die Suche auf verschlüsselten Daten zu beschleunigen, wobei auch die Stärken der oben genannten Datenbankindexe genutzt werden. Letztere sollen in Abgrenzung zu den Strukturdaten im Folgenden *konventionelle* oder *herkömmliche Datenbankindexe* genannt werden, während die Strukturdaten als *verschlüsselte Datenbankindexe* bezeichnet werden.

1.5.1 Verschlüsselte Indexe in den folgenden Kapiteln

Analog zu konventionellen Datenbankindexen, bei denen Indeximplementierungen Stärken und Schwächen gegenüber unterschiedlichen Anwendungsszenarien haben[18], wurden auch für den in Kapitel 1.4 genannten VdV-Ansatz unterschiedliche Typen von verschlüsselten Datenbankindexen entwickelt, um unterschiedliche Typen von Suchanfragen optimiert bedienen zu können.

Im Folgenden wird erläutert, welche Kapitel dieser Dissertation sich welchen Themenbereichen verschlüsselter Indexe widmen.

1.5.1.1 Vorbereitende Arbeiten

Die folgenden Kapitel 2, 3 und 4 beinhalten vorbereitende Arbeiten: Kapitel 2 behandelt verwandte Veröffentlichungen, Kapitel 3 macht einige Angaben zu wichtigen Fachbegriffen,

[18] So eignen sich beispielsweise Hash-Indexe nur schlecht für Intervallsuchen, während sie bei punktuellen (Gleichheits-)Suchen Stärken haben.

die in der Dissertation verwendet werden. Kapitel 4 führt unterstützende Datenstrukturen ein, insbesondere aus dem Datenbankkontext, um die in den folgenden Kapiteln beschriebenen Algorithmen und Datenstrukturen einheitlich und übersichtlich behandeln zu können.

1.5.1.2 Indexe für die Suche auf Gleichheit

Der naheliegendste Suchtyp in einem DBMS ist die Suche nach allen Tupeln t aus der Extension $X(r)$ einer Datenbankrelation r, für die der Wert eines bestimmten Attributs a einem Suchkriterium s gleicht (in relationaler Algebra ausgedrückt als, $\sigma_{r.a=s}(r)$ bzw. $\sigma_{r.a=s}(X(r))$). Dabei kann das Konzept von Gleichheit bei genauerer Betrachtung unterschiedliche Ausprägungen haben. Verschlüsselte Indexe für die Suche auf Gleichheit auf verschlüsselten Daten werden in Kapitel 5 behandelt. Dort wird zunächst das abstrakte Konzept der *semantischen Gleichheit* eingeführt, bevor für eine Vielzahl von in einem DBMS vorkommenden Datentypen eine Implementierung dieses Konzepts angeboten wird.

1.5.1.3 Ungleichheitssuche / Intervallsuche / Präfixsuche

Im folgenden Kapitel 6 werden Datentypen betrachtet, auf denen eine lineare Ordnung und die entsprechende asymmetrische Relation „<“ definiert sind. Aus der Menge von Suchtypen, die auf „<“ operieren, wurde die Intervallsuche als allgemeinster Suchtyp ausgewählt, wie beispielsweise die Suche nach allen Tupeln aus der Extension von r, deren Attributwert für a größer als s_1 und kleiner als s_2 sind: $\sigma_{s_1<r.a \wedge r.a<s_2}(r)$. Sie deckt die Ungleichheitssuche mit ab, wie etwa die Suche nach allen Tupeln der Extension von r, die größer als s sind ($\sigma_{r.a>s}(r)$). Mit Hilfe der Intervallsuche lässt sich auch die Präfixsuche auf lexikografisch geordneten Daten implementieren, also eine Suchoperation, die dem regulären Ausdruck $w\Sigma^*$ auf einem Alphabet $\Sigma = \{\sigma_1, \dots, \sigma_n\}$ mit $w \in \Sigma^*$ entspricht. Dies ist möglich, da sich ein solcher regulärer Ausdruck leicht in eine Intervallsuche umformulieren lässt.

Die Intervallsuche auf verschlüsselten Daten kann beschleunigt werden, indem die lineare Ordnung der Daten dem Server offenbart wird und er sich diese Information zunutze macht. Dies kann beispielsweise durch die Verwendung eines speziellen Verschlüsselungsverfahrens erreicht werden, bei dem die möglichen Kryptotexte der gleichen linearen Ordnung unterliegen wie ihre Klartextpendants. Solche *order-preserving encryption*-Verfahren (*OPE*-Verfahren) sind in zahlreichen Veröffentlichen wie [Agrawal2004], [Boldyreva2009], [Seungmin2009] und [Kerschbaum2014] vorgestellt worden. Jedoch leiden viele dieser Verfahren unter Informationsverlust[19] und haben Performanznachteile gegenüber Standardverfahren wie AES; daher sind sie in der kommerziellen Nutzung nicht weit verbreitet. Zudem stehen sie im Widerspruch zum Paradigma der freien Wahl des Verschlüsselungsverfahrens im VdV-Ansatz.

Aus diesem Grund wird in Kapitel 6 mit dem *cTree* eine Datenstruktur eingeführt, die die lineare Ordnung der Indexdaten in zwei verschiedenen Repräsentationen im Klartext offenbart.[20] Der cTree verwendet konventionelle Datenbankindexe und wird in einem mehrstufigen, iterativen Prozess verwendet, um die Intervallsuche auf verschlüsselten Daten zu beschleunigen. In Kapitel 6 wird nicht nur die Intervallsuche unter verschiedenen Datentypen betrachtet, sondern darüber hinaus auch Einsatzmöglichkeit des cTree für die Gleichheitssu-

[19] siehe [Popa2013], S. 1.

[20] Der mit der Offenlegung der linearen Ordnung einhergehende Informationsverlust wird in Kapitel 10 diskutiert.

che, Auswirkungen von Einfüge-, Aktualisierungs- und Löschoperationen auf dem Index, seine Verwaltung und Spezialfälle bei seiner Verwendung.

1.5.1.4 Infixsuche

Nachdem die cTree-Datenstruktur in Kapitel 6 ausgiebig behandelt wurde, wird in Kapitel 7 mit dem *shrinking-window*-Ansatz eine Erweiterung davon vorgestellt, die die Menge der auf der cTree-Datenstruktur durchführbaren Suchen auf die Menge der Infixsuchen erweitert, auch wenn dies nur unter Inkaufnahme der Offenlegung von weiterer Klartextinformation über die verschlüsselten, indexierten Daten möglich ist.

Ein kleiner Ausblick beschreibt in Kapitel 7 darüber hinaus eine weitere Anwendungsmöglichkeit des cTree-Indexes, mit der sich Suchen nach komplexeren regulären Ausdrükken auf verschlüsselten Daten implementieren lassen.

1.5.1.5 Volltext-Index

Mit einer Kombination der in den Kapiteln 5, 6 und 7 vorgestellten Techniken kann eine Annäherung an einen Volltext-Index auf verschlüsselten Daten implementiert werden, die in Kapitel 8 vorgestellt wird. Hierzu werden zunächst unterschiedliche Tokenisierungsansätze vorgestellt und anschließend deren Kombinierbarkeit mit Gleichheits- und Ungleichheitssuche betrachtet.

1.5.2 Performanz des cTree

Der in Kapitel 6 behandelte cTree-Index stellt die wichtigste der in dieser Dissertation vorgestellten Indexierungstechniken für verschlüsselte Daten dar. Für ihn werden in Kapitel 9 mehrere Techniken zur Optimierung seiner Performanz präsentiert.

Neben trivialen Maßnahmen wie dem gezielten Einsatz von herkömmlichen Datenbankindexen oder Redundanz werden Optimierungsmöglichkeiten für Betriebsmodi, Datenstrukturen für die Implementierung des cTree-Index und der Einsatz definiter Verschlüsselung diskutiert. Weiterhin werden die Auswirkungen von Pipelining und Caching zur Performanzsteigerung beim schreibenden und lesenden Zugriff untersucht.

1.5.3 Sicherheitsaspekte

Allen in den Kapiteln 5 bis 8 vorgestellten verschlüsselten Datenbankindexen ist gemein, dass sie dem Server Teile der Klartextinformation der Indexdaten offenbaren, um dessen Möglichkeiten der Datenverarbeitung zu nutzen. Offensichtlich bedeutet dies eine Beeinträchtigung der Vertraulichkeit der Daten in einem Informationssystem, das diese Indexe nutzt. Kapitel 10 betrachtet diese Problematik eingehend. Das Kapitel beginnt mit einer Aufstellung der von den Indexen exponierten Klartextinformation; Beispiele dafür sind etwa der Einsatz definiter Verschlüsselung bei Indexen für die Suche auf Gleichheit und die Exponierung der linearen Ordnung von Daten in der cTree-Datenstruktur.

Daraus werden verschiedene Szenarien für Angriffe auf die Vertraulichkeit der verschlüsselten Daten abgeleitet, welche wiederum in die Kategorien von statischen und dynamischen Angriffen unterteilt werden. Dabei liegt der Fokus deutlich auf statischen Angriffen.

Für die genannten Angriffsszenarien werden im weiteren Verlauf von Kapitel 10 verschiedene Modifikationen der verschlüsselten Indexstrukturen vorgestellt, die die Gefährdungen eliminieren oder zumindest signifikant abschwächen. Dazu gehören die Umsetzung der

Gleichheitssuche ohne Verwendung definiter Verschlüsselung, Verschlüsselung von Fremdschlüsselbeziehungen, redundante Speicherung randomisiert verschlüsselter Indexelemente, zyklische Verschiebung des Attributwertebereichs der indexierten Attribute, gezielte Verschleierung von präzisen Wortlängen bei Indexen für die Infixsuche sowie etablierte Verfahren wie *k*-anonymity, Padding, Verschleierung zeitlicher Kohärenz und Onion Routing.

1.5.4 Performanzmessungen

Performanzmessungen auf konkreten Implementierungen der verschlüsselten Indexstrukturen, deren Ergebnisse in Kapitel 11 präsentiert werden, runden die Dissertation ab. Die Testreihen sind dabei in den folgenden Dimensionen variiert:

- Messungen für
    - konventionelle Datenbankindexe auf Klartextdaten,
    - verschlüsselte Indexe in ihrer Basisversion und
    - eine Variante derselben mit gemäß Kapitel 10 erhöhter Sicherheit.
- Messungen für Gleichheits- und Intervallsuche.
- Messungen für schreibenden und lesenden Zugriff.
- Variation des Umfangs der Extension der indizierten Attribute.
- Variation der Beschaffenheit der zu indizierenden Attributextension.
- Variation der vorherrschenden Latenz zwischen Client und Server.

Darüber hinaus wird eine weitere Testreihe für die Intervallsuche über eine Variation einer bestimmten cTree-Eigenschaft (der Teilbaumhöhe) präsentiert sowie Testreihen über die Auswirkungen der in Kapitel 9 vorgestellten Maßnahmen zur Performanzsteigerung.

1.5.5 Zusammenfassung, Ausblick und Bibliografie

Kapitel 12 beinhaltet die Zusammenfassung der vorangegangenen Kapitel sowie das Fazit und den Ausblick auf zukünftige Erweiterungen. Die Bibliografie in Kapitel 13 schließt die Dissertation ab.

2 Verwandte Arbeiten

Auf dem Gebiet von Methoden und Techniken zur Suche auf verschlüsselten Daten existieren zahlreiche Publikationen, auch wenn die Mehrheit davon andere Schwerpunkte hat und andere Strategien verfolgt, als es in dieser Dissertation der Fall ist. Beispielsweise liegt der Fokus vieler früherer Arbeiten auf verschlüsselten Dokumenten, die nach bestimmten, fest vordefinierten Schlüsselwörtern durchsucht werden können (*keyword search*), anstatt auf Ansätzen, die die gesamte Extension eines verschlüsselten Attributs einer Datenbankrelation nach beliebigen Suchbegriffen effizient zu durchsuchen. Somit weichen diese Arbeiten von der eigentlichen Zielsetzung dieser Dissertation ab; jedoch gibt es Anwendungsszenarien, bei denen ihre Ergebnisse dennoch mit den in dieser Dissertation gewonnenen Erkenntnissen vergleichbar sind.

Beispielsweise präsentieren die Autoren von [Song2000] ein Verfahren, das für die Verschlüsselung vertraulicher Dokumente geeignet sei und das probabilistische Suchen auf den verschlüsselten Dokumenten ermögliche (was bedeutet, dass false positives möglich sind). Dabei beweisen die Autoren formal die Sicherheit des Verfahrens. Bei dem Verfahren werden alle Wörter des indizierten Dokuments durch Aufspaltung bzw. Padding auf eine fixe Länge gebracht. Wenn nun nach einem bestimmten Schlüsselwort gesucht wird, werde die verschlüsselte Extension nach passenden Kryptotexten durchsucht.

In [Boneh2004] präsentieren die Autoren ein kryptografisches Verschlüsselungsverfahren, das sie „*Public-key encryption with keyword search*“ (*PEKS*) nennen. Das Verfahren ermögliche Suchen auf verschlüsselten Dokumenten, indem verschlüsselte Dokumente, die an entfernten und möglicherweise nicht vertrauenswürdigen Orten gespeichert sind (z. B. E-Mails), mit einer Anzahl von Schlüsselwörtern markiert werden. Bei einer Suche auf diesen Dokumenten sende der Client eine Menge von Schlüsselwörtern zum Server und lasse diesen prüfen, ob eine E-Mail mit einem oder mehreren der Schlüsselwörter markiert ist.

Dieses Konzept wird in [Abdalla2005] erweitert. Hier wird neben einem weiteren Verschlüsselungsverfahren ein Transformationsverfahren vorgestellt, das ein anonymes identitätsbasiertes Verschlüsselungsverfahren (*IBE*) in ein PEKS überführt, sowie einige weitere Erweiterungen des ursprünglichen Basisschemas.

Weiterhin ist [Boyen2007] zu nennen, die ebenfalls einem PEKS-ähnlichen Ansatz folgt, obwohl sich die Publikation eigentlich nur am Rande mit PEKS beschäftigt.

Die Publikation [Li2010] erweitert den keyword search-Ansatz vom bisher in diesem Forschungszweig verfolgten *exact match*- zu einem *fuzzy match*-Ansatz: Beispielhaft wird von einer Menge von 10.000 keywords ausgegangen, mit denen Dokumente, also Volltext-Instanzen, indexiert werden können, sowie von einer maximalen *edit distance* von 2. Letzteres bedeutet, dass an jedem keyword bis zu zwei Veränderungen zulässig sind, also Hinzufügungen, Löschungen oder Veränderungen von bis zu zwei Zeichen. Davon ausgehend generiert ein straight-forward-Ansatz laut den Autoren für den Index Variationen mit einem Datenvolumen von 30 GB.

Zur Reduktion dieser Datenmenge bieten die Autoren zwei Verfahren an: Das erste, „*wildcard-based fuzzy set construction*“, bietet anstelle der expliziten Ausformulierung der Variationen eine wildcard-basierte Repräsentation an, durch die das Datenvolumen des Index auf

40MB reduziert werden könne. Das zweite Verfahren, „*gram-based fuzzy set construction*“, nutzt Teilstring-Charakteristika aus, mit deren Hilfe das Indexdatenvolumen auf 10MB weiter reduziert werden kann. Die generierten fuzzy sets werden in einer n-ären Baumstruktur organisiert, die bei einer Suche mit einem neu entwickelten Algorithmus traversiert wird. Zu einem Eingabewort werden diejenigen keywords (und damit verknüpfte Dokumenten) als Treffer zurückgegeben, die einen exakten Treffer produziert haben. Falls kein solcher Treffer stattgefunden hat, werden diejenigen keywords zurückgegeben, die die geringste Distanz zu den Eingabewörtern haben. Eine Sicherheitsanalyse belegt, dass die vorgestellten Verfahren keine Information über die indizierten keywords preisgeben.

In [Wang2014] wird der Ansatz aus [Li2010] unter Zuhilfenahme von Techniken wie *locality-based hashing* und dem *Bloom-Filter* erweitert, so dass auch fuzzy-match-basierte Suchen anhand mehrerer Eingabewörter effizient durchgeführt werden können.

In [Wang2010] und [Cao2014] wird das „multi-keyword search“-Konzept aus [Wang2014] um den Aspekt der Gewichtung der Suchergebnisse erweitert, so dass diesen jeweils ein Rang zugewiesen wird – diesen Ansatz bezeichnen die Autoren als „multi-keyword ranked search over encrypted cloud data (MRSE)“. Die Gewichtung der Suchergebnisse wird auf Basis der Verfahren des coordinate matchings und der inner product similarity ermittelt.

In den Arbeiten [Boneh2007] und [Golle2004] wird ein Verschlüsselungsverfahren vorgestellt, das mehrere Verschlüsselungsprimitive enthält, die Vergleiche und Teilmengenbildungen auf den verschlüsselten Daten erlauben. Dabei betonen die Autoren der Arbeiten die Fähigkeit des Schemas, konjunktive Anfragen auf verschlüsseltem Kryptotext ohne Informationsverlust durchführen zu können bei gleichzeitiger relativ geringer Größe des Kryptotexts im Vergleich zu anderen Algorithmen. Dieser Ansatz liefert gute Ergebnisse, wenn auch nur für eine kleine Menge von Suchtypen. Vor allem aber geht er dem Paradigma dieser Dissertation entgegen, beliebig verschlüsselte Daten für unterschiedliche Suchtypen zu indizieren.

Die Autoren von [Bellare2007] präsentieren ein kryptografisches Primitiv zur Suche auf Gleichheit auf verschlüsselten Daten, das sie „*efficiently-searchable encryption*“ (*ESE*) nennen und das auf dem Gebiet der asymmetrischen Kryptografie angesiedelt ist. Es fällt auf, dass die Autoren die Möglichkeiten definiter Verschlüsselung ausnutzen, so wie dies auch bei der Suche auf Gleichheit in dieser Dissertation getan wird, auch wenn dies im Gegensatz zu den hier vorgestellten Verfahren unter Verwendung asymmetrischer Kryptografie stattfindet.

Die Autoren geben an, ESE-verschlüsselte Datenbankextensionen könnten genau so effizient indexiert, aktualisiert und durchsucht werden wie entsprechende unverschlüsselte Datenbankextensionen und führen eine ausführliche Betrachtung der Sicherheit des vorgestellten ESE-Systems durch.

Eine wichtige Veröffentlichung ist [Popa2012], in welcher Anregungen für den Entwurf sicherer Informationssysteme und auch für diese Dissertation gefunden werden können. Ähnlich zu dem bei dieser Dissertation gewählten Ansatz liegt der Fokus von [Popa2012] auf dem Entwurf eines Informationssystems mit verschlüsseltem Datenbank-Backend. Die Autoren präsentieren ein entsprechendes System mit dem Namen „CryptDB“. Darin werden Benutzer-Credentials wie beispielsweise Passwörter mit kryptografischen Schlüsseln assoziiert, so dass verschlüsselte Daten nur von denjenigen Benutzern entschlüsselt werden könnten, die Zugriff auf die entsprechenden Credentials haben.

Weiterhin präsentieren die Autoren von [Popa2012] den Ansatz, einen Satz von „SQL-kompatiblen“ (*SQL-compatible*), bzw. „SQL-bewussten“ (*SQL-aware*) Verschlüsselungsverfahren zusammen- und bereitzustellen, mit denen jeweils native SQL-Anfragen auf verschlüsselten Daten mit unterschiedlichen Sicherheitsniveaus, aber auch unterschiedlichen Möglichkeiten hinsichtlich Durchsuchbarkeit ausgeführt werden könnten. Die genannten Verschlüsselungsverfahren werden auf die folgenden drei Klassen verteilt: Randomisierte (*RND*), deterministische (*DET*) und homomorphe Verfahren (*HOM*).

Es wird ein Verfahren namens „*Onions of Encryption*“ vorgestellt, bei dem Daten mehrfach mit Algorithmen aus unterschiedlichen Klassen verschlüsselt würden und das serverseitig flexibel auf sich ändernde Anforderungen bezüglich der gewünschten Durchsuchbarkeit der Daten angepasst werden könne. Bezüglich deterministischer Verfahren führen die Autoren aus, dass diese insbesondere für Suchen auf Gleichheit geeignet seien und damit auch für die Verwirklichung von JOIN-Operationen.

Jedoch belassen es die Autoren von [Popa2012] bezüglich der Implementierung der genannten Verfahren für die Durchsuchbarkeit von verschlüsselten Daten bei der Nennung der verschiedenen Klassen von Verschlüsselungsverfahren wie RND, HOM oder DET, während in dieser Dissertation mehr Wert auf die Ausarbeitung der verschiedenen Klassen von Verschlüsselungsverfahren, Suchtypen darauf und deren Beschleunigung der Fall ist.

So werden in [Popa2012] keine detaillierte Implementierungen der Gleichheitssuche unter Verwendung definiter Verschlüsselung in Abhängigkeit des darunter liegenden Datentyps beschrieben, wie es in Kapitel 5 dieser Dissertation der Fall ist. Es wird keine ordnungserhaltende Datenstruktur wie der cTree vorgestellt, die Ungleichheitssuchen auf verschlüsselten Daten ermöglicht und die durch Modifikationen auch für die Durchführbarkeit von Präfix-, Infix- und Postfixsuchen auf verschlüsselten Daten erweitert werden kann. Die Anwendung von Gleichheitssuche und cTree-Datenstruktur zur Implementierung eines verschlüsselten Volltext-Indexes fehlt dementsprechend ebenfalls.

Auch die Performanz erweiternde Maßnahmen wie Caching-Strategien und Pipeline-basierte Index-Updates werden in [Popa2012] nicht behandelt, so dass sich die in dieser Dissertation präsentierten Indexierungstechniken vom CryptDB-Ansatz deutlich abgrenzen.

Die Autoren von [Hacigümüş2002-1] und [Hacigümüş2002-2] behandeln in ihren beiden recht frühen Publikationen ein Anwendungsgebiet, das für die vorliegende Dissertation ebenfalls von Belang ist: Informationssysteme, bei denen die Serverseite in einer nicht vertrauenswürdigen Umgebung angesiedelt ist. In [Hacigümüş2002-1] wird ein webbasierter Ansatz präsentiert, in dem „*database as a service*“ in einer Serverarchitektur bestehend aus Application Server und Datenbankserver bereitgestellt wird. Der Application Server enthält eine Vielzahl von Java-Servlets, die auf den Datenbankserver zugreifen. Das System wird komplettiert durch einen Webserver, auf den über eine verschlüsselte TLS-Verbindung von einem gängigen Webbrowser zugegriffen wird.

Auch wenn es sich hierbei um kein neuartiges Szenario handelt, ist es interessant zu sehen, dass die Autoren von der gleichen Rahmenbedingung ausgehen, die auch der in Kapitel 0 dieser Dissertation beschriebenen Motivation zugrunde liegt: Der Betrieb der Serverkomponente in einer Umgebung, von deren Vertrauenswürdigkeit nicht ausgegangen werden kann. Das probate Mittel dagegen sei die Sicherstellung der Vertraulichkeit der im Informa-

tionssystem gespeicherten Daten durch deren Verschlüsselung. Darüber hinaus entwerfen die Autoren ihr System unter dem selben Paradigma „Do as much work on the server as possible“, das auch für die im Rahmen dieser Dissertation entwickelten Indexstrukturen gilt.

Die Autoren von [Hacigümüş2002-1] diskutieren die Vor- und Nachteile verschiedener Verschlüsselungsverfahren sowie unterschiedlicher Grade von Verschlüsselungsgranularität und testen ihr System unter Verwendung des TPC-H-Benchmarks[21]. Dennoch weist das vorgestellte Informationssystem den entscheidenden Nachteil auf, dass die Verschlüsselung der Daten auf dem Server geschieht und sie dort zumindest zeitweise im Klartext vorliegen.

In [Hacigümüş2002-2] nehmen sich die Autoren der Behebung dieses Nachteils an. In dieser Arbeit liegt, wie in der vorliegenden Dissertation, der Fokus auf der Erstellung von Indexstrukturen, die einen beschleunigten Zugriff auf verschlüsselte Daten ermöglichen. Diese Indexstrukturen sollen ebenfalls in einer Umgebung mit nicht vertrauenswürdigen Serverkomponenten betrieben werden können. Die Autoren führen an, aus diesem Grund müssten alle Verschlüsselungs- und Entschlüsselungsoperationen ausschließlich auf der Clientseite ausgeführt werden. Dennoch gelte für das dargestellte Informationssystem wie in dieser Dissertation auch das Paradigma, dass so viel Arbeit wie möglich auf der Serverseite durchgeführt werden solle.

Es wird die Erstellung eines „groben“ Indexes („coarse index“) für ein Attribut a_i vorgeschlagen, das in einem herkömmlichen DBMS Teil einer Klartextrelation r ist, neben den weiteren Attributen $a_1, \dots a_{i-1}, a_{i+1}, \dots, a_n$. Anstatt jedoch r wie oben beschrieben zu speichern, wird hier so vorgegangen, dass zunächst der Wertebereich von a_i in eine Menge von disjunkten Intervallen $i_1, \dots i_k$ partitioniert wird, die durch die Labels $l_1, \dots, l_k$ repräsentiert werden. Wenn ein Tupel $t = (t_1, \dots, t_n)$ in r eingefügt werden soll, wird zunächst die Partition i_j mit dem dazugehörigen l_j ermittelt, in der $t_i = \pi_{a_i}(t)$ liegt. Anschließend wird t als Ganzes zu $t' = Enc(t)$ verschlüsselt und zusammen mit l_j gespeichert.

Wenn eine Selektionsoperation $\sigma_{a_i=x}(r)$ nach allen Tupeln von r ausgeführt werden soll, die für a_i den Wert x aufweisen, wird zunächst die Partition i_m ermittelt, in der x liegt. Anschließend werden alle Tupel von der Datenbank abgerufen, die mit dem entsprechenden Label l_m versehen sind, und zum Client übertragen. Dort werden sie von der Clientanwendung entschlüsselt und nach denjenigen Instanzen gefiltert, die tatsächlich für a_i als Attributwert den Suchwert x haben. Die Autoren führen aus, dass dadurch, dass ausschließlich verschlüsselte Daten, zusammen mit den Labels l_i der Wertebereichpartitionen, zum potenziell nicht vertrauenswürdigen Server transferiert würden, die Vertraulichkeit der in r gespeicherten Daten gesichert sei.

Die Autoren von [Hacigümüş2002-2] verfolgen weiterhin das Ziel, die oben beschriebene Kernidee in jeden Aspekt des SQL-Standards einfließen zu lassen, oder zumindest in die gängigsten Teile davon (auch wenn dies nicht explizit genannt wird, kann davon ausgegangen werden, dass es sich dabei um den SQL-92-Standard[22] handelt). So könne die Bedingung einer Selektionsoperation die folgenden Kriterien beinhalten: Gleichheit und Ungleichheit[23]

[21] siehe [URL-TPC].

[22] siehe [URL-SQL-92].

[23] Hiermit ist die tatsächliche Ungleichheit gemeint, die mit dem Operator „≠“ ausgedrückt wird und keine „>“- bzw. „<“-Relation.

mit Konstanten, Gleichheit und Ungleichheit mit anderen Attributen, sowie Konjunktionen und Disjunktionen. Negationen werden ausgelassen. Weitere abgedeckte Aspekte des SQL-Standards umfassten JOIN-Operationen, Gruppierungsoperationen, Aggregatfunktionen, Sortierung, Duplikateliminierung (wie beispielsweise in einer SELECT DISTINCT-Anweisung), Mengendifferenz, Vereinigung und Projektion.

Darüber hinaus wird ein „algebraisches Framework zur Aufspaltung von Anfragen" vorgestellt („*algebraic framework for query splitting*"), das als eine Art zusätzlicher query optimizer einsetzbar sei. Es übersetze auf der verschlüsselten Datenbank abgesetzte Anfragen in eine Form, die von der oben beschriebenen Funktionsweise von Selektionsoperationen schnell ausgeführt werden könne, wieder unter der Maßgabe, so viel von der für die Abarbeitung der Anfrage anfallenden Arbeit wie möglich dem Server zu überlassen. Die Autoren legen Wert auf die Feststellung, sie stellten kein vollständiges, optimale Anfragen produzierendes Framework vor, sondern lediglich eine begrenzte Menge von Heuristiken, die abgesetzte Anfragen übersetzten.

Die Autoren setzen sich mit einem Problem auseinander, das auch dieser Dissertation zugrunde liegt: In einem Informationssystem ist jede Entität, die Zugriffsrechte auf Daten, aber kein explizites „need-to-know" dafür hat, eine potenzielle Bedrohung für die Vertraulichkeit der Daten (siehe Kapitel 1.2). Daher setzen die Autoren sich das ambitionierte Ziel, ein Informationssystem mit entfernten Serverkomponenten zu entwickeln, das ohne die Notwendigkeit einer vertrauenswürdigen dritten Stelle auskommt, und streben eine vollständige Übersetzung des SQL-Standards in eine Variante an, in der Vertraulichkeit wirksam über Verschlüsselung geschützt wird.

Es ist jedoch fraglich, ob der in [Hacigümüş2002-2] vorgestellte Ansatz der Erzeugung von „groben Indexen" die Möglichkeiten bietet, den Anforderungen sowohl nach angemessener Performanz für gängige Datenbankoperationen als auch nach angemessenem Schutz der Vertraulichkeit sensitiver Daten gerecht zu werden. Bezüglich ersterem fällt die massive Abrufung, Übertragung und clientseitige Entschlüsselung von verschlüsselten Tupeln t auf, die mit einem a_i-Partitionslabel l_j versehen sind, sich nach der Entschlüsselung jedoch als keine Treffer herausstellen. Diese false positives werden bei simplen Selektionsoperationen auf einzelnen Relationen bereits zahlreich auftreten; bei Selektionen über mehrere Relationen, die per JOIN-Operation kombiniert worden sind, multipliziert sich ihre Anzahl und wächst somit sehr stark an. Bezüglich des Schutzes der Vertraulichkeit ist es als problematisch anzusehen, dass die Partitionslabels an den Tupeln zwar nicht den konkreten Attributwert verraten, dafür aber das Intervall, in dem der Wert liegt. Dies stellt einen nicht unerheblichen Informationsverlust dar.

Eine naheliegende Möglichkeit, die hohe Anzahl von false positives für die Treffermenge der Selektion abzusenken, ist, die Größe der eingesetzten Partitionen zu verringern, d. h. ihre Anzahl zu erhöhen. Dadurch wird die Wahrscheinlichkeit (und dadurch auch die Anzahl) von false positive-Tupeln abgesenkt; gleichzeitig bewirkt dieser Schritt jedoch einen höheren Informationsverlust, da die Partitionslabels ein kleineres Intervall kodieren. Umgekehrt verringern größere Partitionsintervalle den Informationsverlust, erhöhen jedoch die Anzahl von false positives.

Das sich hier manifestierende Trade-off „Vertraulichkeit vs. Performanz" ist nicht ungewöhnlich im Bereich verschlüsselter Informationssysteme. Jedoch sind in diesem Fall

schwerwiegende Effekte zulasten von sowohl Vertraulichkeit als auch Performanz beobachtbar, so dass ein akzeptabler Kompromiss schwer erreichbar erscheint. Dieser Eindruck wird durch die Aussage der Autoren verstärkt, dass gute Performanz mit den vorgestellten Indexen erreichbar sei, wenn die Wertebereichspartitionen nur klein genug gemacht würden.

Neben den bisher genannten existieren noch weitere Publikationen, die interessante Strategien verfolgen, jedoch weicht deren Betätigungsfeld meist weit von dem dieser Dissertation ab. So versucht [Ge2007] etwa, Indexe auf verschlüsselten Daten zu erzeugen, indem ein homomorphes Verschlüsselungsverfahren angewendet wird, um Aggregatfunktionen auf den verschlüsselten Daten zu berechnen. Ein anderes Beispiel ist [Pappas2011] Der Fokus dieser Publikation liegt darauf, bestehende, schlüsselwortbasierte Suchsysteme auf verschlüsselten Daten mit Intervallsuchen anzureichern.

Auf dem Gebiet der homomorphen bzw. ordnungserhaltenden Verschlüsselung sind weitere Publikationen veröffentlicht worden, wie z. B. [Gentry2009], [Agrawal2004], [Boldyreva2009] und [Liu2013]. Jedoch weisen sowohl homomorphe als auch ordnungserhaltende Verfahren nach wie vor einen (insbesondere im homomorphen Fall erheblichen) Performanznachteil auf: So sprechen etwa die Autoren von [Gentry2009] von einer recht hohen, aber immer noch polynomiellen Laufzeit[24]. Im Hinblick auf ungewollte Offenlegung von Information ist darüber hinaus in [Popa2013] gezeigt worden, dass bei sehr vielen OPE-Verfahren die Kryptotexte eine erhebliche Anzahl von Bits der jeweiligen Klartexte verraten[25]. Ein weiterer Nachteil dieser beiden Klassen von Verschlüsselungsverfahren ist deren mangelnde Akzeptanz bzgl. praktischer Anwendung in konkreten Softwareprodukten.

Mit dem Einsatz spezialisierter Hardware verfolgt eine vorrangig aus dem Microsoft-Umfeld stammende Gruppe von Publikationen einen weiteren Ansatz. Zwei Beispiele dafür sind [Arasu2013] und [Bajaj2014]. Deren jeweilige grundlegende Idee basiert darauf, dass die Serverkomponente eines Informationssystems eine herkömmliche CPU zur Verarbeitung aller nichtsensitiven Daten verwendet, während alle sensitiven Daten in eine vertrauenswürdige Hardwarekomponente transferiert werden, welche verarbeitete Daten in Richtung DBMS nur in verschlüsselter Form ausgibt und in Empfang nimmt.

Die Autoren nehmen für ihren vorgestellten Ansatz in Anspruch, dass er zwar weniger performant sei als Klartextsysteme, aber um Größenordnungen schneller als rein softwarebasierte Ansätze wie homomorphe oder OPE-basierte Verfahren. Dennoch weist der Ansatz den Nachteil auf, dass solche auf dem Einsatz spezialisierter Hardware basierenden Systeme teuer in der Anschaffung und schwierig in der Administration sind, ganz abgesehen davon, dass bei diesen Systemen die Entschlüsselungsschlüssel auf der Serverseite residieren, wenn auch innerhalb von gesicherten Hardware-Modulen.

In [Tu2013] präsentieren die Autoren das System MONOMI, das laut den Autoren darauf optimiert sei, analytische Anfragen auf verschlüsselten Daten zu bearbeiten. MONOMI beinhaltet unter anderem die zentralen Komponenten *designer* und *planner*, wovon erstere für ein optimiertes physikalisches Layout der Daten zur Designzeit verantwortlich sei und letztere für eine optimierte Spezifikation der abgesetzten Anfragen zur Laufzeit. MONOMI sei mit guten Ergebnissen auf den TPC-H-Benchmarks angewendet worden, wobei betont wer-

[24] siehe [Gentry2009], S. 177.

[25] siehe [Popa2013], S. 1.

den soll, dass es laut den Autoren auf lesende Anfragen (also SELECT-Statements) optimiert sei. Performanz-Optimierungen, wie sie in Kapitel 8 dieser Dissertation vorgestellt werden, sind in [Tu2013] nicht berücksichtigt. Darüber hinaus verlässt sich MONOMI im Falle von Intervallsuchen auf OPE-Verfahren, was in dem in dieser Dissertation vorgestellten Ansatz vermieden wird.

In [Popa2013] zeigen die Autoren, wie bereits erwähnt, dass gegenwärtige OPE-Verfahren neben der unverschlüsselt preisgegebenen linearen Ordnung der verschlüsselten Elemente auch weitere Teile der verschlüsselten Information preisgeben[26]. Sie schlagen daher ein alternatives System vor, das der für diese Dissertation entwickelten cTree-Datenstruktur ähnelt. Popa et. al. nennen ihren Ansatz „*order-preserving encoding scheme*" und haben ihn so entworfen, dass dort ebenfalls ein beliebiges Verschlüsselungsverfahren eingesetzt und die lineare Ordnung der Elemente in einer binären Baumstruktur nachgebildet wird. Die zweite Nachbildung der linearen Ordnung in Form einer Ordnungszahl ist in [Popa2013] ebenfalls vorhanden, ist dort jedoch aufwändiger zu berechnen und erheblich teurer bei Aktualisierungen der Struktur (also im realen Einsatz).

Trotz einer Ähnlichkeit der Grundidee des oder-preserving encoding scheme und der cTree-Datenstruktur unterscheiden sich die beiden Ansätze in ihrer Ausrichtung: Ersterer orientiert sich am theoretischen Beweis, dass das Verfahren im Gegensatz zu anderen OPE-Verfahren abgesehen von der exponierten linearen Ordnung nicht mit Informationsverlust behaftet ist (ein Beweis, der sich aufgrund der Ähnlichkeit der Datenstrukturen auf den cTree übertragen lässt und somit zeigt, dass auch diese unter keinem inhärenten Informationsverlust leidet). Bei letzterer liegt der Fokus auf einem flexiblen, performanzoptimierten Einsatz in kommerziellen Software-Produkten.

In der Sicherheitsbetrachtung berufen sich Popa et. al. auf *IND-OCPA*, eine Definition von Sicherheit, die in [Boldyreva2009] eingeführt wurde. Es ist eine Abwandlung von IND-CPA, der von Goldwasser und Micali in [Goldwasser1984] eingeführten Sicherheitsdefinition einer Nichtunterscheidbarkeit von verschlüsselten, beliebig gewählten Klartexten[27]; Boldyreva et. al. schwächen diese für OPE-Verfahren nicht erfüllbare Definition zu *indistinguishability under ordered chosen-plaintext attack* (*IND-OCPA*) ab, so dass ein Angreifer nicht zwischen zwei Sequenzen von verschlüsselten Werten unterscheiden kann, so lange die beiden Sequenzen der selben Ordnungsrelation folgen.[28] Popa et. al. beweisen, dass beliebig verschlüsselte[29] Daten, die in einer binären Baumstruktur organisiert sind, welche die lineare Ordnung der Daten ausdrückt, keine Information über den verschlüsselten Klartext preisgeben und somit, im Gegensatz zu den meisten OPE-Verfahren, IND-OCPA erfüllen[30]. Dies ist eine wichtige Eigenschaft, die sich auch auf die in Kapitel 6 eingeführte cTree-Indexdatenstruktur übertragen lässt, welche einen wichtigen Platz in dieser Dissertation einnimmt.

In [Kamara2015] setzen sich die Autoren mit Angriffen auf Systeme auseinander, die kryptografische Verfahren wie definite und ordnungserhaltende Verschlüsselung einsetzen, wel-

[26] siehe [Popa2013], S. 1.

[27] *Indistinguishability under chosen plaintext attacks*; siehe [Goldwasser1984].

[28] siehe [Boldyreva2009], S. 6.

[29] Mit „beliebiger Verschlüsselung" ist gemeint, dass das bestmögliche Verschlüsselungsverfahren gewählt werden kann, ohne ausschließlich auf OPE-Verfahren eingeschränkt zu sein.

[30] siehe [Popa2013], S.16.

che auch in den in dieser Dissertation vorgestellten Indexierungsverfahren zum Einsatz kommen. Insbesondere beziehen die Autoren sich auf das in [Popa2012] vorgestellte System *CryptDB* sowie weitere kommerzielle verschlüsselte Informationssysteme. Es werden Angriffe durchgeführt, die sich in die folgenden vier Klassen einteilen lassen: *frequency analysis* (Häufigkeitsanalyse unter Zuhilfenahme von öffentlich verfügbarer Zusatzinformation), ℓ_p-*optimization* (eine neuartige Angriffsart auf definit verschlüsselte Daten, die auf kombinatorischen Optimierungen basiert), *sorting attack* (eine simple Angriffsart auf OPE-verschlüsselte Daten, die auf der Annahme beruht, dass der gesamte Wertebereich in der Extension eines verschlüsselten Attributs vorhanden ist) und *cumulative attack* (ein neuartiger Angriff auf OPE-verschlüsselte Daten, der ebenfalls kombinatorische Optimierung verwendet und ohne die starke Annahme von *sorting attack* auskommt).

Die Autoren führen an, dass bei Verwendung der oben genannten Angriffsarten je nach Anwendungsszenario ein substanzieller Klartextverlust bis hin zu 99% der verschlüsselten Extension erreicht werden könne. Somit kann [Kamara2015] als weitere Motivation gewertet werden, die Indexierungstechniken dieser Dissertation nicht nur in einer funktionstüchtigen und auf Performanz optimierten Basisversion bereitzustellen, sondern auch mit Abwehrstrategien gegen Angriffe wie die in [Kamara2015] dargestellten zu kombinieren. Dies wird in Kapitel 10 getan, so dass den Indexierungstechniken in ihrer Basisversion auch Varianten mit einer größeren Resilienz gegen Angriffe zur Seite stehen.

3 Grundlagen

In dieser Dissertation werden Kenntnisse der meisten verwendeten Begriffen der Informatik vorausgesetzt. Dennoch macht dieses Kapitel einige Angaben zu zentralen Begriffen aus den Bereichen Datenbanken, Sicherheit von Informationssystemen und Kryptografie.

3.1 Informationssysteme

Alle Angaben zu Informationssystemen, insbesondere relationalen Datenbanksystemen, die dem allgemeinen Stand der Technik entsprechen, wurden [Kemper2011] entnommen, dem Standardwerk auf diesem Gebiet im deutschsprachigen Raum. In diesem Zusammenhang werden hier folgende Anmerkungen gemacht:

3.1.1 Relationale Algebra

Angesichts des anwendungsorientierten Charakters dieser Dissertation und des Fokus auf DBMS wird zahlreich Gebrauch von relationalen Ausdrücken gemacht. Abgesehen von Programmfragmenten, die in der Datenbankanfragesprache SQL (siehe Kapitel 3.1.2) angegeben werden, werden viele Ausdrücke der relationalen Algebra[31] angegeben, wobei diese zumeist mit den Hauptoperatoren der Selektion (σ), der Projektion (π) und des Verbundes ($\bowtie$) auskommen (siehe auch Kapitel 3.5).

3.1.2 SQL

Alle in dieser Arbeit angeführten DDL- und DML-Anweisungen, die in einem DBMS in Form einer Anweisung in der Datenbankanfragesprache SQL[32] abgesetzt werden können, orientieren sich am internationalen Standard SQL-92[33]. Dies gilt ebenso für die verschiedenen Datentypen von Attributen einer Datenbankrelation: So steht beispielsweise der SQL-Datentyp $integer$ stets für ganze 32-Bit-Zahlen mit Vorzeichen, $varchar(n)$ für ein auf dem ASCII-Zeichensatz basierendes Zeichenkettenattribut mit der maximalen Länge n[34] und $varbinary(n)$ für binäres Attribut mit der maximalen Bytelänge n[35].

3.1.3 Konventionelle Datenbankindexe

Alle in dieser Dissertation vorgestellten Indexierungstechniken machen sich die Stärken konventioneller Datenbankindexe zunutze. Deren verschiedene Ausprägungen und Implementierungsformen sind gut erforscht und dokumentiert[36], weshalb in dieser Arbeit nicht im Detail darauf eingegangen wird. Ein konventioneller Datenbankindex wird als eine Zusatzstruktur angesehen, die unter Inkaufnahme von zusätzlichem Speicherplatz auf einem oder einer Menge von Attributen einer Datenbankrelation definiert werden und den Zugriff auf die Extensionselemente der Attributmenge erheblich beschleunigen kann.

Bezüglich der Implementierung konventioneller Datenbankindexe gibt es zwei wesentliche Ausprägungsformen: B- bzw. B⁺-Baum-Indexe und Hash-Indexe. Sie haben unterschiedliche

---

[31] siehe [Kemper2011], S. 85-98.
[32] siehe [Kemper2011], S. 111-158.
[33] siehe [URL-SQL-92].
[34] Oft wird jedoch von einer Spezifizierung der Länge abgesehen, so dass $varchar$ ohne Beschränkung der Allgemeinheit einen Zeichenketten-Datentyp beliebiger Länge beschreibt.
[35] Auch hier gilt, das zumeist nur vom Datentyp $varbinary$ die Rede ist.
[36] siehe [Kemper2011], S. 213-230.

Eignungsprofile für unterschiedliche Datenzugriffsoperationen, weshalb es ratsam ist, zur Designzeit eines Informationssystems nicht nur die bestmöglichen Attribute für eine Indexierung auszuwählen, sondern auch die bestmögliche Implementierungsform der Indexe. Eine Analogie zu dieser Eigenschaft weisen auch die in dieser Dissertation dargestellten Indexierungsformen verschlüsselter Daten auf.

3.1.3.1 B- / B⁺-Baum-Indexbäume[37]

Bei der Implementierung eines Index als B- oder B⁺-Baum werden dessen Elemente in einer Baumstruktur organisiert. Anstelle eines binären Baumes wird ein Baum mit höherem Verzweigungsgrad bevorzugt, um das Datenvolumen der Baumknoten auf die Größe der Speicherseiten des Betriebssystems zu abzustimmen und so die Anzahl der Speicherseiten-Zugriffe zu minimieren. B-/B⁺-Bäume sind für die meisten Suchtypen gut geeignet und haben insbesondere bei Intervallsuchen einen Performanzvorteil gegenüber Hash-Indexen.

3.1.3.2 Hash-Indexe[38]

Bei Hash-Indexen wird eine Hashfunktion eingesetzt, um Schlüsselwerte, also etwa die Werteinstanzen eines hash-indizierten Attributs, auf einen Speicherbereich (*bucket*) abzubilden, der das dem Schlüsselwert zugeordnete Datum (beispielsweise ein Tupel einer Relation) enthält. Hash-Indexe eignen sich insbesondere für Gleichheitssuchen (*punktuelle Suchen*), die sie sehr schnell bedienen können. Für andere Suchtypen wie etwa Intervallsuchen sind sie dagegen ungeeignet und tragen nicht wesentlich zur Beschleunigung einer Anfrage bei.

3.2 Sicherheit von Informationssystemen

Aus Platzgründen kann keine vollständige Einführung in das Thema der Sicherheit von Informationssystemen gegeben werden. Es sei stattdessen auf das Buch [Eckert2013] verwiesen, das einen guten Überblick über die Thematik bietet. Im Folgenden werden lediglich einige Begriffe definiert, denen in dieser Dissertation besondere Bedeutung beigemessen wird.

3.2.1 Vertrauenswürdige dritte Stelle

Eine vertrauenswürdige dritte Stelle, laut [VanTilborg2014] „*an entity in a domain that is trusted to perform a specific service*“[39], ist ein gängiger Bestandteil von Informationssystemarchitekturen. Üblicherweise ist sie auf der Serverseite angesiedelt, also an einem Ort, der sich der Kontrolle der Benutzer entzieht. Im Kontext dieser Dissertation wird die Existenz bzw. Notwendigkeit einer vertrauenswürdigen dritten Stelle als eine Schwachstelle der Architektur angesehen, da ihre postulierte Vertrauenswürdigkeit nicht mit der angenommenen Nicht-Vertrauenswürdigkeit der Serverseite vereinbar ist. Die vertrauenswürdige dritte Stelle würde als ein zentrales Angriffsziel exponiert, sowohl für Angreifer von außen, vor allem aber auch für korrupte Insider, die die ihnen zugeteilten Privilegien für Manipulationen der vertrauenswürdigen dritten Stelle missbrauchen könnten.

3.2.2 Need-to-know-Prinzip

Der Begriff des „need-to-know“ wird in [Furnell2008] folgendermaßen definiert: „*The need-to-know model is based on the principle that a subject can accesss only those objects that are necessary for fulfilling his work duties (i. e., he is required to have a ‚need-to-know‘ on the content of the object).*

[37] siehe [Kemper2011], S. 216-222.
[38] siehe [Kemper2011], S. 222-226.
[39] siehe [VanTilborg2014], S. 1335.

As the access permissions of subjects are assigned according to their competence and work area, the model is sometimes also called policy of confidence.“[40]

Im Kontext dieser Dissertation beschreibt es eine Eigenschaft der Serverseite des angestrebten Informationssystems. Danach stehe allen (potenziell korrumpierten) Personen auf der Serverseite des Informationssystems nur genau so viel Information zur Verfügung, wie sie zur Durchführung ihrer Aufgaben benötigen, und ausdrücklich nicht der Klartext der im Informationssystem gespeicherten Daten.

3.3 Kryptografie

Ähnlich zum Themenbereich „Sicherheit von Informationssystemen“ (siehe Kapitel 3.2) wird in dieser Dissertation keine umfassende Einführung in das Thema „Kryptografie“ gegeben. Bezüglich aller Fragen hierzu sei auf [Schneier2007] verwiesen, eines der Standardwerke auf diesem Gebiet. Tatsächlich herrscht in dieser Dissertation mit wenigen Ausnahmen ein abstrakter Blick auf Kryptografie vor. Danach wird diese in den meisten Fällen lediglich durch ein symmetrisches Verschlüsselungsverfahren in Form der beiden abstrakt gehaltenen Funktionen Enc und Dec repräsentiert[41]. Enc überführe dabei einen beliebigen Klartextwert x in einen für einen Angreifer unverständlichen Kryptotext $Enc(x) = x'$. Dec überführe einen Kryptotextwert zurück in einen Klartextwert und es gelte

$$Dec\big(Enc(x)\big) = x. \qquad \textbf{(F1)}$$

Auch von der Verwendung eines symmetrischen Schlüssels s bei Ver- und Entschlüsselungsoperationen, durch die (F1) zu

$$Dec(Enc(x, s), s) = x \qquad \textbf{(F2)}$$

abgewandelt werden müsste, wird in dieser abstrakten Sichtweise auf symmetrische Verschlüsselung abgesehen. Ohne Beschränkung der Allgemeinheit wird von einem beliebigen, aber konstanten s ausgegangen.

Dem Initialisierungsvektor (*IV*), einem besonderen Eingabeparameter beim Betrieb des Kryptosystems im CBC-Modus, kommt hingegen im weiteren Verlauf der Dissertation eine wichtigere Rolle zu: Stete Variation des IV sorgt üblicherweise dafür, dass bei jeder Verschlüsselung eines Klartextwertes x ein anderer Kryptotext x' berechnet wird, der aber dennoch stets zum gleichen Klartextwert x entschlüsselt werden kann:

Sei $\{iv_1, \dots, iv_n\}$ eine Menge von IV mit $i, j \in \{1, \dots, n\}: i \neq j \Leftrightarrow iv_i \neq iv_j$. Dann gelte für alle Klartext-Eingabewerte x für die symmetrische Verschlüsselung im CBC-Modus:

$$\forall i, j \in \{1, \dots, n\}: i \neq j \Leftrightarrow iv_i \neq iv_j \Leftrightarrow Enc(x, iv_i) = x_i' \neq x_j' = Enc\big(x, iv_j\big) \qquad \textbf{(F3)}$$

und

$$\forall i, j \in \{1, \dots, n\}: Dec(x_i', iv_i) = Dec\big(x_j', iv_j\big) = x. \qquad \textbf{(F4)}$$

---

[40] siehe [Furnell2008], S. 64.

[41] In den praktischen Anwendungen wurde dieses Kryptosystem durch den symmetrischen Blockchiffre AES-256 CBC (*Advanced Encryption Algorithm* im *Cipher Block Chaining*-Modus mit einer Schlüssellänge von 256 Bit) implementiert.

Weiterhin gelte für alle (x_i', x_j'), die auf die in (F3) beschriebene Art berechnet wurden, dass $Dec(x_i', iv_i) = Dec(x_j', iv_j)$ ohne Kenntnis von s nicht mit vertretbarem Aufwand ermittelbar sei. Entsprechend (F3) gelte weiterhin:

$$\forall i, j \in \{1, \dots, n\}: iv_i = iv_j \Leftrightarrow Enc(x, iv_i) = Enc(x, iv_j). \quad \textbf{(F5)}$$

Die oben beschriebenen Anwendungsmöglichkeiten des IV finden in den folgenden beiden Betriebsmodi der symmetrischen CBC-Verschlüsselung Verwendung:

3.3.1 Definite Verschlüsselung

Definite Verschlüsselung bezeichne einen Operationsmodus, bei dem bei jeder Ver- und Entschlüsselungsoperation der IV beliebig, aber konstant gehalten werde. Dadurch bilde die Verschlüsselungs entsprechend (F5) gleiche Klartextwerte stets auf gleiche Kryptotexte ab. Definite Verschlüsselung sei durch die Schreibweisen Enc_{det} und Dec_{det} gekennzeichnet.

3.3.2 Randomisierte Verschlüsselung

Dieser Begriff beschreibe die Verwendung von variierenden IV gemäß (F3) und sei durch die Schreibweisen Enc_{rand} und Dec_{rand} gekennzeichnet.

3.3.3 Beliebige Verschlüsselung

Sollte die Wahl des obigen Betriebsmodus unerheblich sein, wird von beliebiger Verschlüsselung gesprochen, die durch Enc und Dec gekennzeichnet sei.

3.4 AVL-Baum

Der Datenstruktur des AVL-Baums[42] kommt in Kapitel 6 eine besondere Bedeutung zu, weshalb sie in diesem Unterkapitel näher betrachtet wird. Es handelt sich im Wesentlichen um einen binären Baum mit der Bedingung für jeden seiner Knoten, dass die Höhe seiner beiden Teilbäume sich um maximal 1 unterscheiden darf. Dadurch wird eine suboptimale Auslastung des Baumes bewusst in Kauf genommen, jedoch lässt sich zeigen, dass ein AVL-Baum auch im ungünstigsten Fall maximal 44% höher ist als ein perfekt balancierter Baum.

Die Höhe jedes Teilbaums kann rekursiv, ausgehend von seiner jeweiligen Wurzel, ermittelt werden; effizienter ist jedoch, diese Höhe redundant in jedem Knoten zu speichern und bei Reorganisationsoperationen des AVL-Baumes entsprechend anzupassen.

Die oben genannte Bedingung kann beim Einfügen (neue Knoten werden ausschließlich auf der Blattebene eingefügt) und Löschen von Elementen verletzt werden, so dass ein inkonsistent gewordener AVL-Baum rebalanciert werden muss. Hierfür kommen in Abhängigkeit der Operation, die zur Dysbalance geführt hat, unterschiedliche Vorgehensweisen in Frage:

3.4.1 Rebalancierung nach einer Einfügung

Zur Erkennung einer Dysbalance aufgrund eines auf der Blattebene des Baumes neu eingefügten Knotens geht der Rebalancierungsalgorithmus so vor, dass er vom Elternknoten des neu eingefügten Knotens zurück zur Wurzel des Baumes navigiert und dabei auf jedem Knoten entlang dieses Pfades die oben genannte Bedingung prüft. Bei Nichterfüllung, einer lokalen Dysbalance also, wird selbige durch *Rotationsoperationen*, nämlich *Links-*, *Rechts-*,

[42] siehe [Pfaff1998], S. 107-137.

*Links-Doppel- u*nd *Rechts-Doppelrotationen* wieder hergestellt. Es lässt sich zeigen, dass nach einer Einfügung in einen AVL-Baum maximal eine solche Rotation nötig ist, so dass der AVL-Baum danach wieder balanciert ist und die Rücktraversierung beendet werden kann.

3.4.2 Rebalancierung nach einer Löschung

Die Löschung kann im Gegensatz zur Einfügung an jedem beliebigen Knoten des Baumes stattfinden. Bei Löschungen auf der Blattebene wird der Knoten einfach entfernt; im Falle der Löschung eines inneren Knotens des Baumes tritt dagegen einer seiner Nachfolgerknoten an seine Stelle. Eine daraufhin mögliche Dysbalance wird durch Rotationen behoben. Im Gegensatz zur Einfügung kann der Rebalancierungsalgorithmus anschließend nicht terminieren, sondern muss rekursiv weiter aufgerufen werden, bis sichergestellt ist, dass der ganze Baum, d. h. die Wurzel als Endpunkt der Rücktraversierung, balanciert ist.

3.5 Nomenklatur und Notation

In diesem Unterkapitel werden Konventionen für die in der Dissertation verwendete Nomenklatur und Notation von häufigen Begriffen festgelegt.

C	Clientrechner
ρ	Prozess auf einem Clientrechner
DB	Datenbank(server), bzw. DBMS
S	Datenbankschema
r	Datenbankrelation
a	Klartextattribut einer Datenbankrelation
a'	Verschlüsseltes Attribut einer Datenbankrelation
t	Tupel aus der Extension einer Datenbankrelation
σ	Selektion
π	Projektion
$\bowtie$	Verbund (Join)
X	Extension einer Menge; $X(r) = \{t_1, \dots, t_n\}$: Extension der Relation r; $X(a) = X(\pi_a(r)) = \{\pi_a(t_1), \dots, \pi_a(t_n)\}$: Extension des Attributs a
$ix_{r.a'}$	Datenbankrelation, die die Elemente eines Index enthält, der auf dem (verschlüsselten) Attribut a' einer Relation r definiert ist
$ix_{r.a'}^{equal}$	Datenbankrelation für die Elemente eines Index für die Gleichheitssuche auf dem (verschlüsselten) Attribut $r.a'$
$ix_{r.a'}^{cTree}$	Datenbankrelation für die Elemente eines cTree-Indexes für die Intervall- oder Präfixsuche auf dem (verschlüsselten) Attribut $r.a'$
T	AVL-Baum
v	Knoten eines AVL-Baums T; v_i: Knoten in T mit Nummer i
τ	Teilbaum eines AVL-Baums; τ_i: Teilbaum mit Wurzelknoten v_i
H	Höhe eines AVL-Baums T
h	Höhe eines Teilbaums τ von T
χ_i	Cacheeintrag, der den Teilbaum τ_i enthält
Enc	(Beliebige) Verschlüsselungsfunktion
Enc_{det}	Definite Verschlüsselungsfunktion
Enc_{rand}	Randomisierte Verschlüsselungsfunktion
Dec	Entschlüsselungsfunktion

4 Verwaltungsstrukturen

Diese Dissertation zeichnet sich durch hohen Praxisbezug aus. In den Kapiteln 5, 6, 7 und 8, die einen wesentlichen Bestandteil dieser Dissertation bilden, werden Indexierungsmethoden vorgestellt, die verschlüsselte Indexstrukturen generieren. Letztere beschleunigen den Zugriff auf die Extensionen von Attributen bzw. Kombinationen von Attributen in Datenbankrelationen (*Datenrelationen*). Die Indexdaten werden jeweils als Extension einer eigens dafür angelegten Datenbankrelation (*Indexrelation*) abgelegt.

Ohne Beschränkung der Allgemeinheit wird im Folgenden bei Beispielen zumeist Bezug auf ein einzelnes, verschlüsseltes Attribut a' einer Datenrelation r genommen, das mit einem verschlüsselten Index ix versehen sei bzw. wird. Dabei sind die Indexierungsmethoden auch auf Klartextattribute und auf beliebige Kombinationen von verschlüsselten und Klartextattributen anwendbar, auch wenn verschlüsselte Indexe darauf wenig Sinn ergeben.

Vor der Einführung der Indexierungstechniken ab Kapitel 5 behandelt dieses Kapitel Vorarbeiten und Verwaltungsstrukturen, die für Indexerstellung, -benutzung und -wartung notwendig bzw. hilfreich sind. So führt das Kapitel 4.1 einen formalistischen Rahmen für anwendungsspezifische Vorarbeiten beim Entwurf des Systems ein. Diese beinhalten den Entscheidungsprozess, welche Attribute in den Datenrelationen des Schemas zu verschlüsseln und darüber hinaus, welche Attribute bzw. Attributkombinationen mit einem Index zu versehen sind. Das Ergebnis der in Kapitel 4.1 beschriebenen Arbeitsschritte ist damit das Datenbankschema des Informationssystems, vorerst bestehend aus Datenrelationen, sowie eine präzise Menge von Angaben über die zu erstellenden verschlüsselten Indexe. Anschließend befassen sich Kapitel 4.2 und 4.3 mit der Vereinheitlichung der erzeugten Indexstrukturen, ihrer Verbindung zu ihren Datenrelationen und ihres Einsatzes.

4.1 Anwendungsspezifische Vorarbeiten

Beim Entwurf eines Informationssystems definiert dessen Designer entsprechend spezifischer Anforderungen an das zu erstellende System ein Datenmodell mit entsprechendem Datenbankschema S. Er folgt dabei den üblichen Vorgehensweisen der relationalen Modellierung[43]. Bei der Spezifizierung der in den Relationen $r_i \in S$ enthaltenen Attribute wird darüber hinaus analysiert, welche dieser Attribute in ihren Extensionen Informationen speichern, die nicht der Serverseite offenbart werden dürfen und daher durch Verschlüsselung geschützt werden müssen.

Ergebnis dieses initialen Schritts ist ein Datenbankschema, in dem eine Teilmenge der Attribute als zu verschlüsselnd identifiziert worden ist, während die Elemente ihres Komplements im Klartext verbleiben können. Ein naiver Ansatz, bei dem einfach alle Attribute des Schemas als zu verschlüsselnd gekennzeichnet werden, ist nicht empfehlenswert, da dies die Performanz unnötig beeinträchtigen würde.[44]

---

[43] siehe [Kemper2011], S. 31-70.

[44] Die Entscheidung, ein Attribut zu verschlüsseln, das sensitive Information enthält, kann auch solche Attribute treffen, die eine Fremdschlüsselbeziehung konstituieren. Hier wird besonders deutlich, dass die Entscheidung zur Verschlüsselung mit Sorgfalt zu treffen ist, denn ein solcher Schritt hat erhebliche Konsequenzen für den Zugriff auf das Informationssystem. Eine JOIN-Operation über zwei Relationen, die mit einer verschlüsselten Fremdschlüsselbeziehung verbunden sind, kann nicht mehr auf

Es wird also ein Datenbankschema S definiert, das eine Menge von Attributen

$$A = \{a_{1,1}, \dots, a_{1,m_1}, \dots, a_{n,1}, \dots, a_{n,m_n}\} \quad \textbf{(F6)}$$

beinhaltet, sowie eine Menge von Relationen

$$R = \{r_1, \dots, r_n\}, \quad \textbf{(F7)}$$

wobei jedes r_i als Tupel mit einer Teilmenge der Attribute aus A darstellbar ist, so dass R beschreibbar ist als:

$$R = \{\langle a_{1,1}, \dots, a_{1,m_1}\rangle, \dots, \langle a_{n,1}, \dots, a_{n,m_n}\rangle\}. \quad \textbf{(F8)}$$

Weiterhin wird bei diesem initialen Schritt der Schemadefinition applikationsspezifisch eine Teilmenge von Attributen $A' \subseteq A$ ermittelt, deren Extensionen Informationen mit einem erhöhten Schutzniveau beinhalten, so dass sie auf dem Server nicht im Klartext vorliegen dürfen und deshalb verschlüsselt gespeichert werden müssen. Im Folgenden wird davon ausgegangen, dass dieser Teilaspekt der Schemadefinition (menschliches) Domänenwissen erfordert und daher nicht automatisiert durchgeführt werden kann.

A' ist disjunkt in Teilmengen zerleg- und auf die $r_i \in R$ verteilbar:

$$A' = \bigcup_{i=1}^{n} A'_i, \text{ mit } \forall a' \in A'_i: a' \in r_i \text{ für } 1 \leq i \leq n, \quad \textbf{(F9)}$$

Aus den Potenzmengen aller A'_i sind nun, wieder abgestimmt auf die spezifischen Erfordernisse des entwickelten Informationssystems, diejenigen Teilmengen A_i^{ix} von Attributmengen zu bestimmen, von denen erwartet wird, dass ihre Elemente während des normalen Betriebs des Informationssystems Teil eines Zugriffspfades sind und deshalb mit einem verschlüsselten Index zu belegen sind:

$$A^{ix} = \bigcup_{i=1}^{n} A_i^{ix}, \text{ mit } \forall i \in \{1, \dots, n\}: A_i^{ix} \subseteq \wp(A'_i) \quad \textbf{(F10)}$$

Schließlich ist noch festzulegen, mit welchen Arten von verschlüsselten Indexen die Elemente von A^{ix} belegt werden, d. h. welche Suchtypen darauf möglich gemacht werden sollen. Hierfür ist eine feste Menge von Symbolen $searchTypes$ definiert, in der ein jedes Element für einen spezifischen Suchtyp und damit auch Indextyp steht:

$$searchTypes = \{equality, inequality, prefix, infix, postfix, fulltext\} \quad \textbf{(F11)}$$

Damit sind in $searchTypes$ genau diejenigen Suchtypen festgelegt, die durch die in den Kapiteln 5 bis 8 beschriebenen Indexierungsmethoden ermöglicht werden:

- $equality$: Suche auf Gleichheit über Attributen beliebigen Datentyps (siehe Kapitel 5).
- $inequality$: Suche auf Ungleichheit über Attributen beliebigen Datentyps, auf denen eine lineare Ordnung definiert ist (siehe Kapitel 6).
- $prefix$, $infix$, $postfix$: Präfixsuche, Infixsuche und Postfixsuche auf Attributen, die lexikografisch geordnete Daten beinhalten (siehe Kapitel 6 und 7).

---

die übliche Art und Weise performant auf dem Server ausgeführt, sondern muss auf den Client ausgelagert werden, was sich üblicherweise negativ auf die Performanz des Informationssystems auswirkt.

- Volltextsuche auf Attributen eines Zeichenketten-Datentyps, die längere Freitexte beinhalten (siehe Kapitel 8).

Mit $searchTypes$ kann nun, wieder anwendungsbezogen, die Menge der erforderlichen verschlüsselten Indexstrukturen, die die Datenbank des Informationssystems enthalten muss, spezifiziert werden:

$$A^{ixs} \subseteq \left(A^{ix} \times searchTypes\right) \quad \textbf{(F12)}$$

4.2 Strukturanpassungen in der verschlüsselten Datenbank

Neben der Spezifikation der zu verschlüsselnden und gegebenenfalls zu indexierenden Attribute sind weitere DBMS-seitige Anpassungen vorzunehmen. Sie betreffen die Verwaltung der Datenbankrelationen und deren Attribute in S, sowie die Dokumentation und einheitliche Handhabung der verschlüsselten Indexe im Zusammenspiel mit den von ihnen indizierten Attributen und Relationen. Weiterhin werden einige Namenskonventionen für Attribute, Index- und M:N-Relationen definiert.

4.2.1 Secure System Catalog

Die Angabe des Klartext-Datentyps der in A' enthaltenen, zu verschlüsselnden Attribute soll nicht verloren gehen, denn die Attribute werden diesen Datentyp verlieren, wenn sie verschlüsselt mit binärem Datentyp gespeichert werden.[45] Für weitere Eigenschaften von verschlüsselten Datenbankattributen, wie beispielsweise Integritätsbedingungen, gilt das gleiche, wie auch für die Information, auf welchen verschlüsselten Attributen oder Attributkombinationen welche Typen von verschlüsselten Indexen definiert sind.

Eine naheliegende Vorgehensweise zur Speicherung dieser Information ist, all diese Eigenschaften auf der Clientseite in die Anwendungslogik zu kodieren. So kann beispielsweise spezifiziert sein, dass

- eine verschlüsselte Bitfolge aus der Extension eines binären Attributs auf der Serverseite nach ihrer Entschlüsselung auf der Clientseite in ein Objekt des Datentyps $integer$ umgewandelt wird,
- eine DEFAULT-Constraint auf einem Attribut vor der Speicherung eines Tupels in der Datenbank angewendet werden kann und
- bei einer SELECT-Anfrage, bei der auf Gleichheit mit einem verschlüsselten Attribut geprüft wird, ein gemäß den Ausführungen in Kapitel 5 definierter, verschlüsselter Index herangezogen werden kann.

Eine andere Vorgehensweise mit einer besseren Strukturiertheit beim Umgang mit verschlüsselten Attributen sieht vor, dass die oben genannten Metainformationen im DBMS persistiert sind. Beim Beginn einer Sitzung ruft der Client sie vom Server ab und hält sie während der gesamten Dauer der Sitzung im Speicher.

Zu diesem Zweck wird dem Datenbankschema S eine zusätzliche Datenstruktur hinzugefügt, die im Folgenden *Secure System Catalog* genannt wird. Der Name orientiert sich am normalen Systemkatalog, der in gängigen DBMS enthalten ist und die Informationen von

[45] Im Folgenden werde stets davon ausgegangen, dass der Kryptotext des verwendeten Verschlüsselungsverfahrens stets von binärem Datentyp sei, der gemäß der Terminologie des DBMS Microsoft SQL Server $varbinary$ genannt werde.

herkömmlichen Schemas persistiert (dort aber keine verschlüsselten Attribute und/oder versschlüsselte Indexe berücksichtigt). Der Secure System Catalog besteht aus den folgenden beiden (unverschlüsselten) Relationen, die S hinzugefügt werden:

- $relations$: Die Namen aller Datenrelationen, die im Schema des Informationssystems enthalten sind (siehe Kapitel 4.2.2)[46]. Weiterhin die Namen aller verschlüsselten Indexrelationen (siehe Kapitel 4.2.3) sowie die Namen aller M:N-Relationen, die diese Index- mit den dazugehörigen Datenrelationen verbinden (siehe Kapitel 4.2.4).
- $attributes$: Die Namen aller Attribute der in $relations$ enthaltenen Datenrelationen, ein Verweis auf die sie enthaltende Datenrelation, ihr Verschlüsselungsstatus sowie spezifische Eigenschaften wie DEFAULT-Constraints, NULL- bzw. NOT NULL-Constraints, etc.

4.2.2 Datenrelationen

Diejenigen Relationen in S, deren Extensionen den eigentlichen Inhalt des Informationssystems speichern und die demnach diejenigen sind, die verschlüsselte und gegebenenfalls auch verschlüsselt indexierte Attribute beinhalten, werden im Folgenden Datenrelationen genannt. Für sie ändert sich in dem teilweise verschlüsselten Datenbankschema im Vergleich zu einem Klartextschema lediglich, dass ihre verschlüsselten Attribute anstatt ihres originären Datentyps einen binären Datentyp zugewiesen bekommen, um darin verschlüsselte Werte zu speichern.

Weiterhin können, wie schon in 4.2.1 ausgeführt, zahlreiche Eigenschaften auf den verschlüsselten Attributen, wie etwa $DEFAULT$-Constraints, nicht mehr vom DBMS allein umgesetzt, sondern müssen auf den verschlüsselten Daten und/oder im Verbund mit dem Client implementiert werden, gegebenenfalls unter Zuhilfenahme des Secure System Catalog. Andere Typen von Integritätsbedingungen können auch auf verschlüsselten Attributen bestehen bleiben, falls dies gewünscht ist, etwa $NULL$- bzw. $NOT\ NULL$-Constraints[47].

4.2.3 Indexrelationen

Die Extension jedes verschlüsselten Indexes ist in einer herkömmlichen Datenbankrelation gespeichert, die, wie bereits erwähnt, ihrerseits konventionelle DBMS-Indexstrukturen verwendet und deren Stärken ausnutzt. Der Aufbau der Indexrelationen unterscheidet sich je nach dem Indextyp, den sie implementieren.

Allen Indexrelationen in der Basisversion ist gemein, dass ein Indexelement immer durch genau ein Tupel in der Indexrelation repräsentiert wird: Dazu wird auf den Indexwerten eine UNIQUE-Constraint durchgesetzt, wobei diese entweder mit konventionellen DBMS-Mitteln durchgesetzt wird oder, falls dies nicht möglich ist, spezifisch implementiert ist (siehe Kapitel 6.5). Erst in späteren Versionen der verschlüsselten Indexstrukturen werden In-

[46] Diejenigen Attribute eines Relationenschemas, die unverschlüsselt gespeichert werden, müssen nicht im Secure System Catalog aufgeführt werden, da sie im herkömmlichen Systemkatalog vollständig dokumentiert sind. In der Praxis hat sich jedoch eine einheitliche Handhabung als vorteilhaft erwiesen, bei der im Secure System Catalog alle Attribute des Datenbankschemas enthalten sind.

[47] Bzgl. NULL/NOT NULL-Constraints ist zu beachten, dass sie, wenn sie auf einem verschlüsselten Attribut bestehen bleiben, einen ungewollten Informationsverlust darstellen können, da sie die Information, ob ein Wert vorhanden ist oder nicht, preisgeben.

dexelemente bewusst redundant gespeichert, um einen höheren Schutz der Vertraulichkeit der im Index gespeicherten Daten zu erreichen (siehe Kapitel 10.4.4).

Die Benennung von Indexrelationen sei im weiteren Verlauf so vereinheitlicht, dass eine Indexrelation stets mit dem Präfix „ix“ beginnt und qualifiziert ist mit

- dem Namen der referenzierten Relation (etwa r),
- dem Namen des darin indexierten Attributs, bzw. der darin indexierten Attributkombination (z. B. das verschlüsselte Attribut a' in r) und
- dem Indextyp, also einem Element aus $searchTypes$ (etwa $equal$).

Ein Beispiel für die Benennung einer Indexrelation, die einen verschlüsselten Index für die Suche auf Gleichheit auf einem verschlüsselten Attribut a′ einer Relation r implementiert, ist demnach $ix_{r.a',equal}$ bzw. $ix_{r.a'}^{equal}$.

4.2.4 M:N-Relationen

Als Bindeglied zwischen einer Daten- und einer Indexrelation fungiert zum Zwecke einer einheitlichen Behandlung stets eine M:N-Relation, die auf Index- und Datenrelation mit jeweils einem Fremdschlüssel verweist. Um einen schnellen Zugriff von Index- auf Datentupel zu ermöglichen, sei ein konventioneller, vorzugsweise geclusterter DBMS-Index auf den beiden Fremdschlüsselattributen der M:N-Relationen definiert. Dabei verläuft der primäre Zugriffspfad stets von der Index- auf die Datenrelation, weshalb der Index auf dem Fremdschlüsselattribut, das die Indexrelation referenziert, eine höhere Priorität hat als der Index auf dem Fremdschlüsselattribut, das die Datenrelation referenziert.

Das (Indextupel-Referenz, Datentupel-Referenz)-Attributpaar muss eindeutig sein, weshalb eine $UNIQUE$- bzw. die $PRIMARY\ KEY$-Constraint darauf definiert ist.

Weiterhin ist es empfehlenswert, die M:N-Relationen nach einem einheitlichen Schema zu definieren und auch die Benennung der in jeder M:N-Relation enthaltenen Attribute generisch zu halten, damit sie einheitlich zugreifbar sind. Deshalb sei festgelegt, dass in jeder M:N-Relation das die Datenrelation referenzierende Attribut stets $refData$ genannt wird und das die Indexrelation referenzierende Attribut $refIndex$. Mit diesen Spezifikationen lässt sich ein generisches Schema für die Attribute der M:N-Relationen definieren:

- seq ($integer$): Eindeutiges, indexiertes Kandidatenschlüsselattribut zur Tupelidentifikation.[48]
- $refData$ ($integer$): Fremdschlüsselattribut, das die Datenrelation referenziert.
- $refIndex$ ($integer$) Fremdschlüsselattribut, das die Indexrelation referenziert.

Die Benennung der M:N-Relation sei analog zur Benennung der Indexrelationen (siehe Kapitel 4.2.3) mit den gleichen Konventionen vereinheitlicht, lediglich mit dem Präfix „mn“

[48] Solche künstlich eingeführten Kandidatenschlüssel sind aus der Perspektive der reinen Lehre der Datenbanktheorie unnötig, da der Primärschlüssel einer Relation stets über diejenige minimale Attributmenge gebildet werden sollte, die die modellierten Entitäten in der realen Welt eindeutig identifiziert. Aus operationalen Gründen der Einheitlichkeit, Übersichtlichkeit und vor allem der Performanz beinhaltet jede Relation im gesamten Schema S stets das 32-Bit-Integer-Attribut seq zur Tupelidentifikation. Alle Fremdschlüsselbeziehungen zwischen Relationen in S beziehen sich immer auf das seq-Attribut in der referenzierten Relation.

anstelle von „ix“, so dass $mn_{r.a',equal}$ bzw. $mn_{r.a'}^{equal}$ ein beispielhafter Name für eine M:N-Relation sei.

Der nunmehr vollständige Aufbau der M:N-Relationen lässt sich im folgenden generischen $CREATE\ TABLE$-Statement der M:N-Relation ausdrücken:

```
CREATE TABLE mn<relation>.<attribute>,<searchType>
(
  seq INTEGER NOT NULL UNIQUE,
  refData INTEGER NOT NULL REFRERENCES <relation>(seq),
  refIndex INTEGER NOT NULL REFERENCES <indexRelation>(seq),
  PRIMARY KEY(refIndex,refData) CLUSTERED
)                                                                    (Q1)
```

Es ist anzumerken, dass es Sonderfälle gibt, in denen diese Form der Verknüpfung zwischen Daten- und Indexrelation nicht die effizienteste ist, da auch eine direkte Referenzierung der Indexrelation aus der Datenrelation über ein Fremdschlüsselattribut möglich wäre. Index- und Datenrelation könnten demnach mit nur einer JOIN-Operation anstatt mit zweien verknüpft werden. In einem noch einfacheren Fall könnte man sogar ganz auf die Indexrelation verzichten und lediglich die Datenrelation um ein Indexattribut vom Typ $varbinary$ erweitern, das dann mit einem herkömmlichen DBMS-Index belegt würde. Jedoch wäre der Performanzgewinn durch solche Maßnahmen vergleichsweise gering, so dass angesichts der Vorteile einer einheitlichen Zugriffsweise auf die Indexstrukturen darauf verzichtet wird.

4.2.5 $searchTypes$

Die simple Verwaltungsrelation $searchTypes$, die sich an der Definition in (F11) orientiert, speichert die Namen der Suchtypen, die von den verschlüsselten Indexstrukturen unterstützt werden. Sie beinhaltet die folgenden Attribute:

- seq ($integer$): Kandidatenschlüssel zur Tupelidentifikation.
- $searchType$ ($varchar$): Name des Suchtyps. Dieses Attribut bildet den Primärschlüssel der Relation.

In einer Basisversion enthält $searchTypes$ analog zu ihrer Definition in (F11) die Suchtypen $\{equality, inequality, prefix, infix, postfix, fulltext\}$. Werden neue verschlüsselte Indexe entwickelt, die weitere Suchtypen unterstützen, werden entsprechende Tupel hinzugefügt.

4.2.6 indexAdmin

Die Relation *indexAdmin* ist das zentrale Element der Indexverwaltung innerhalb des Secure System Catalog für ein Datenbankschema S. In ihr wird für jeden verschlüsselten Index, der auf einem Attribut oder einer Attributkombination einer Datenrelation angelegt wird, ein (unverschlüsseltes) Tupel gespeichert. *indexAdmin* enthält die folgenden Attribute:

- seq ($integer$): Kandidatenschlüssel zur Tupelidentifikation.
- $indexName$ ($varchar$): Name des Index entsprechend des in Kapitel 4.2.3 definierten Namensschemas. Dieses Attribut bildet den Primärschlüssel der Relation.

- $refDataAttribute$ ($varchar$): Fremdschlüssel auf den in $attributes$ gespeicherten Repräsentanten des Attributs, auf dem der verschlüsselte Index erstellt werden soll.[49]
- $refIndexRelation$ ($varchar$): Fremdschlüssel auf den in $relations$ gespeicherten Repräsentanten des verschlüsselten Index, der das Attribut für einen bestimmten Suchtyp indexiert.
- $refMnRelation$ ($integer$): Fremdschlüssel auf den in $relations$ gespeicherten Repräsentanten der M:N-Relation, die Daten- und Indexrelation miteinander verbindet.
- $refSearchType$ ($integer$): Art der Suche, die durch den Index ermöglicht wird, ausgedrückt durch einen Fremdschlüssel auf $searchTypes$.

Mit dem Verbund der vier Relationen $relations$, $attributes$, $searchTypes$ und $indexAdmin$ beinhaltet der Secure System Catalog nun alle Informationen, die eine Clientanwendung benötigt, um dynamisch eine SELECT-Anweisung auf einem verschlüsselten Attribut einer Datenrelation zu erstellen, die einen bestimmten Suchtyp implementiert.

4.3 Indexerstellung

Nach der Definition der Verwaltungsstrukturen für verschlüsselte Indexe gilt es noch, die nötigen Schritte festzulegen, um einen verschlüsselten Index auf einem verschlüsselten Attribut a' einer Datenrelation r zur Implementierung eines der in $searchTypes$ spezifizierten Suchtypen anzulegen. Dies umfasst die folgenden Schritte:

1. Erzeugen der Indexrelation $ix_{r.a'}$ inklusive herkömmlicher DBMS-Indexstrukturen, entsprechend ihrer jeweiligen Spezifikation in Kapitel 5 bis 8.
2. Erzeugen der M:N-Relation $mn_{r.a'}$ entsprechend Kapitel 4.2.4.
3. Aktualisierung des Secure System Catalog durch die Einfügung folgender Tupel:
    - Ein Tupel in $relations$, das die neue Indexrelation repräsentiert.
    - Ein weiteres Tupel in $relations$, das die neue M:N-Relation repräsentiert.
    - Ein Tupel in $indexAdmin$, das den Index repräsentiert und entsprechend der Spezifikation des neuen verschlüsselten Indexes dessen Namen speichert und die Relationen $attributes$, $relations$und $searchTypes$ refererenziert.

[49] Im Falle eines verschlüsselten Index auf einer Kombination von Attributen ist ein simples Fremdschlüsselattribut offenkundig nicht ausreichend, sondern müsste durch eine separate M:N-Relation zwischen $attributes$ und $indexAdmin$ realisiert werden, worauf an dieser Stelle jedoch aus Übersichtlichkeitsgründen ohne Beschränkung der Allgemeinheit verzichtet wird.

5 Indexe für die Suche auf Gleichheit

In diesem Kapitel werden verschlüsselte Indexstrukturen behandelt, die auf einem verschlüsselten Attribut a' einer Relation r beschleunigte Suchen nach verschlüsselten Werten ermöglichen, welche einem Suchbegriff s gleichen. Eine Suche sieht dabei so aus, dass aus $X(r)$ diejenigen Tupelinstanzen t ausgewählt werden, deren entschlüsselter Attributwert für a' dem Suchbegriff s gleicht:

$$\sigma_{Dec(a')=s}(r) \quad \textbf{(F13)}$$

Somit sind solche Suchanfragen eine Analogie der folgenden Klartext-SQL-Anfrage:

```
SELECT * FROM r WHERE a = s
```
(Q2)

Das DBMS greift dabei auf eine verschlüsselte Datenstruktur $ix_{r.a'}^{equality}$ zu, die die Ausführung einer verschlüsselten Variante von (Q2) beschleunigt, so wie dies in (Q2) mit einem konventionellen Datenbankindex der Fall wäre. Es gibt zunächst keine Einschränkung für den Datentyp der entschlüsselten Form von a'. Weiterhin sei angemerkt, dass sich im Folgenden alle Definitionen, Konzepte, Beispiele etc. auf Indexe auf einem einzelnen verschlüsselten Attribut beziehen, auch wenn dies auch auf Attributkombinationen möglich ist.

5.1 Semantische Gleichheit

Für das Konzept von Gleichheit, insbesondere im Kontext der Informatik, ist eine detailliertere Betrachtung erforderlich, als das naive Verständnis von Gleichheit dies nahelegt. Es ist keineswegs der Fall, dass für Gleichheit eine einheitliche, allgemein gültige Definition existiert; stattdessen ist es im praktischen Gebrauch von Informationssystemen üblich, dass Gleichheit je nach operationalem Szenario und insbesondere je nach verwendetem Datentyp unterschiedlich interpretiert wird und die jeweilige Interpretation dabei von einem streng mathematischen Gleichheitsverständnis abweicht.

Dies ist beispielsweise bei Zeichenketten der Fall (siehe Kapitel 5.5), für deren Vergleiche die Sortiervorschrift (*Collation*) DIN 5007-1[50] zugrunde liegt: Danach werden unter Anderem die Tremata der deutschen Umlaute ignoriert, so dass etwa die Wörter „Mull“ und „Müll“ unter DIN 5007-1 als gleich angesehen werden; auch kann die *case insensitive*-Konvention vorherrschen, nach der Groß- und Kleinschreibung beim Vergleich von Zeichenketten keine Rolle spielt.

Ein anderes Beispiel für ein von der strengen mathematischen Betrachtungsweise abweichendes Gleichheitsverständnis sind Gleitkommazahldarstellungen mit endlicher Präzision, bei denen Gleichheit nach Regeln der numerischen Mathematik definiert ist. Hier werden zwei Zahlen x und y genau dann als gleich angesehen, wenn ihr Abstand einen definierten Schwellwert ε unterschreitet (siehe Kapitel 5.6). So kann eine Äquivalenzrelation $floatequals$ mit Parameter ε die Gleichheit solcher Werte folgendermaßen definieren:

$$floatequals_{\varepsilon}(x, y) \Leftrightarrow |x - y| < \varepsilon \quad \textbf{(F14)}$$

[50] siehe [URL-DIN-5007-1].

Beim diskreten numerischen Datentyp $integer$, der ganze Zahlen darstellt, entspricht Gleichheit zweier Werte dagegen dem naiven Gleichheitsverständnis einer vollständigen, bitweisen Übereinstimmung der beiden verglichenen Werte (*bitwise equality*).

Angesichts der unterschiedlichen Interpretationen von Gleichheit bietet es sich an, mit *semantischer Gleichheit* eine Abstraktion für das Gleichheitskonzept einzuführen. Danach stellt Gleichheit lediglich eine nicht näher definierte Äquivalenzrelation dar[51], die je nach Rahmenbedingung, d. h. Datentyp, unterschiedlich interpretiert werden kann. Eine Interpretation $semEq_M$ von semantischer Gleichheit auf einer Menge M mit $semEq_M \subseteq M \times M$ hat dann die folgenden Eigenschaften von Äquivalenzrelationen zu erfüllen:

- Reflexivität: $\forall x \in M: (x, x) \in semEq_M$ **(F15)**
- Symmetrie: $\forall a, b \in M: (a, b) \in semEq_M \Rightarrow (b, a) \in semEq_M$ **(F16)**
- Transitivität: $\forall a, b, c \in M: (a, b) \in semEq_M \wedge (b, c) \in semEq_M \Rightarrow (a, c) \in semEq_M$ **(F17)**

Die meisten Interpretationen semantischer Gleichheit für unterschiedliche Datentypen sind in herkömmlichen DBMS, d. h. in der Klartextdatenverarbeitung, gut verstanden und umgesetzt[52]. Konventionelle Datenbankindexe können definiert werden, die Gleichheitssuchen auf unverschlüsselten Daten wirksam beschleunigen.

Für verschlüsselte Daten gilt dies jedoch nicht: Bei der Verschlüsselung werden die Daten ihrer Semantik beraubt, so dass die Rechenschritte, die die erwähnten unterschiedlichen Interpretationen semantischer Gleichheit ausmachen, nicht mehr oder nur noch eingeschränkt serverseitig ausführbar sind. So kann der Server beispielsweise weder die Kryptotexte von Zeichenketten auf Gleichheit hinsichtlich einer case insensitive-Collation prüfen, noch die Differenz zweier verschlüsselter Werte zur Ermittlung der semantischen Gleichheit auf Gleitkommazahlen bilden. Solche Berechnungen müssen zumindest teilweise in den Client verlagert werden.

Um dennoch eine hohe Performanz für die Gleichheitssuche auf verschlüsselten Daten zu ermöglichen, müssen die verschiedenen Interpretationen semantischer Gleichheit auch für verschlüsselte Daten umgesetzt und anschließend entsprechende Indexstrukturen dafür entwickelt werden. Eine Anforderung hierbei sei, die Serverseite so intensiv wie möglich mit einzubeziehen, insbesondere unter Verwendung von herkömmlichen Datenbankindexen. Die Unterkapitel 5.3, 5.4, 5.5 und 5.6 widmen sich diesen Interpretationen semantischer Gleichheit für verschlüsselte Daten auf unterschiedlichen Klassen von Datentypen.

5.2 Definite Verschlüsselung

Definite Verschlüsselung ist ein einfaches Hilfsmittel, mit dem es möglich ist, die Semantik der bitweisen Gleichheit vollständig vom Klartext- in den verschlüsselten Kontext zu überführen. Dies wird dadurch erreicht, dass ein Verschlüsselungsverfahren mit zugehöriger Verschlüsselungsfunktion Enc_{det} ausgewählt wird, mit der ein Klartextwert bei jedem Verschlüsselungsvorgang stets zum gleichem Kryptotext führt.

---

[51] siehe [Beutelspacher2003], S. 5.

[52] Andere Gleichheitsdefinitionen müssen auch in der Klartextwelt von Hand implementiert werden, wie etwa die Schwellwertunterschreitung der Differenz zweier Gleitkommazahlen (siehe Kapitel 5.6).

5.2.1 Definite Verschlüsselung über konstante Schlüssel und IV

Die meisten Verschlüsselungsverfahren sind inhärent definit und besitzen zunächst die oben genannte Eigenschaft, beispielsweise symmetrische Blockchiffren wie AES im *Cipher Block Chaining*-Ausführungsmodus (*CBC*). Diesen Verfahren wird künstlich ein nichtdeterministischer Charakter verliehen, etwa durch Beimischen von Zufallsbits (*Salzen*) bei der Initialisierung des Verschlüsselungsvorgangs in Form eines randomisierten *Initialisierungsvektors* (*IV*). Also kann bei AES definite Verschlüsselung dadurch erreicht werden, dass der IV bei der Ver- und Entschlüsselung weggelassen bzw. auf einen konstanten Wert gesetzt (und stets der gleiche symmetrische Schlüssel verwendet) wird.

Für eine definite Verschlüsselungsfunktion Enc_{det} gilt dann die folgende Äquivalenz:

$$\forall x, y: x = y \Leftrightarrow x' = Enc_{det}(x) = Enc_{det}(y) = y' \quad \textbf{(F18)}$$

Bei einer Suche auf der Extension eines verschlüsselten Attributs a' nach Werteinstanzen, die einem Suchwert s gleichen, lässt sich mit Hilfe dieser Äquivalenz berechnen, welchen Kryptotext ein gesuchter Wert haben muss, vorausgesetzt, der für die Verschlüsselung zu verwendende Schlüssel und IV sind bekannt.

5.2.2 Indizierbarkeit mit herkömmlichen Datenbankindexen

Sei a' ein definit verschlüsseltes Attribut einer Relation r mit einem zugrundeliegenden Datentyp, für den semantische Gleichheit als bitweise Gleichheit implementierbar ist (siehe Kapitel 5.3 und 5.4). Die Gleichheit von verschlüsselten Werten x' und y' aus der Extension von a' folgt dann der gleichen Semantik wie die ihrer Klartext-Pendants x und y: Diese sind genau dann (bitweise) gleich, wenn die verschlüsselten Werte x' und y' gleich sind.

Daraus folgt, dass auf a' ein konventioneller Datenbankindex definierbar ist, der für die Ermittlung von Gleichheit genau so schnell auswertbar ist wie ein Index auf einem Klartext-Attribut. Der anzuwendende Suchwert s wird dann ebenfalls definit zu $s' = Enc_{det}(s)$ verschlüsselt und an den Server gesendet. Die SQL-Anfrage auf Klartextdaten (Q2) kann umformuliert werden zu:

```
SELECT * FROM r WHERE a' = s',
```
(Q3)

Das DBMS macht sich bei der Ausführung von (Q3) den konventionellen Datenbankindex auf a' zunutze. Die Ausführung ist dementsprechend praktisch gleich performant wie die von (Q2), abgesehen von dem vergleichsweise kleinen zusätzlichen Aufwand der nötigen Verschlüsselung $s' = Enc_{det}(s)$ und einer marginal größeren Bytelänge der zu vergleichenden Werte bei (Q3), die durch die Verschlüsselung zustande kommt.

Unter Umständen könnte die Suche auf verschlüsselten Daten sogar schneller laufen als auf unverschlüsselten Daten: Da der Index auf a' lediglich auf (bitweise) Gleichheit angewendet werden muss, kann für seine technische Realisierung auch ein darauf optimierter Hash-Index gewählt werden, während sich auf dem Klartextattribut a auch ein flexibler einsetzbarer B- oder B⁺-Baum-Index lohnt, der neben Gleichheits- auch Ungleichheitsvergleiche abdeckt, bei reinen Gleichheitsvergleichen aber langsamer ist.

5.2.3 Komplexere Formen semantischer Gleichheit

Komplexere Interpretationen semantischer Gleichheit auf verschlüsselten Daten sind jedoch nicht bzw. nicht vollständig auf dem Server auswertbar. In verschlüsselter Form sind die Daten eines Großteils ihrer Semantik beraubt, und beispielsweise eine Normalisierung von Groß- und Kleinschreibung von zu vergleichenden Werten ist hier genau so wenig möglich wie die Bildung der Differenz zweier Werte.

Die Strategie für die Umsetzung unterschiedlicher Interpretationen semantischer Gleichheit auf verschlüsselten Daten muss stattdessen sein, gezielt verschlüsselte Zusatzstrukturen einzusetzen, mit deren Hilfe die jeweilige Interpretation semantischer Gleichheit auf eine bitweise Gleichheit innerhalb der Zusatzstrukturen reduziert werden kann. Letztere kann dann performant auf den verschlüsselten Zusatzstrukturen umgesetzt werden. An die Zusatzstrukturen muss gleichzeitig die Anforderung gestellt werden, dass ihre Existenz keinen bzw. keinen signifikanten Informationsverlust bedeutet.

Die folgenden Unterkapitel behandeln Interpretationen semantischer Gleichheit auf unterschiedlichen Datentypen und beschreiben, wie es jeweils gelingen kann, sie auf einem verschlüsselten Attribut umzusetzen.

5.3 Bitweise Gleichheit auf diskreten numerischen Datentypen

Datentypen, die diskrete numerische Werte beschreiben, sind in SQL-92 zum einen die ganze Zahlen darstellenden Datentypen aus der *integer*-Familie (dies umfasst alle Varianten der *integer*-Familie, wie *tinyint*, *shortint* und *bigint*) und zum anderen *decimal* bzw. *numeric* zur Darstellung rationaler Zahlen mit einer fixen Anzahl von Nachkommastellen. Alle geben die Semantik der Zahlen, die sie beschreiben, eindeutig wieder, haben also die Eigenschaft, dass zwei unterschiedliche Werteinstanzen (d. h. zwei unterschiedliche Bitfolgen) immer auch unterschiedliche Zahlen darstellen. Auf diesen Datentypen kann semantische Gleichheit also als bitweise Gleichheit interpretiert werden. Diese Eigenschaft besitzt auch der triviale boolesche Datentyp *bit*, dessen Wertebereich lediglich zwei Werte umfasst, die in genau so vielen unterschiedlichen Bitfolgen dargestellt werden.

Die Werteinstanzen der Extension $X(\pi_{a'}(r))$ des verschlüsselten Attributs a', deren Klartexte einen solchen Datentyp haben, können mit einem ebenfalls definit verschlüsselten Suchbegriff s' bitweise verglichen werden, ohne dass eine weitere Interpretation oder Modifikation der Daten notwendig ist. Die in (F18) beschriebene Äquivalenz hat hier also Gültigkeit.

5.4 Bitweise Gleichheit auf binären Datentypen

Auch wenn binäre Datentypen nicht Teil des SQL-92-Standards sind, wird hier auf sie eingegangen, da sie eine wichtige Rolle in allen gängigen kommerziellen DBMS-Produkten spielen. Analog zum *char*- bzw. *varchar*-Datentyp wird im Folgenden für binäre Datentypen der in Microsoft SQL Server gebräuchliche Datentyp *binary* bzw. *varbinary* verwendet. Für Attribute dieses Datentyps gilt die in 5.3 gegebene Gleichheitsdefinition zunächst analog, jedoch mit dem Unterschied, dass die Bitlänge von Attributinstanzen mit binären Datentypen in praktischen Anwendungen oft groß werden kann.

Eine Suche nach Gleichheit mit einem Suchwert x auf einem binären, verschlüsselten Attribut a' kann teuer werden angesichts der großen Datenmenge, die dabei in Form des Suchwerts transferiert und verglichen werden muss. Hier bietet sich eine Modifikation der in Ka-

pitel 5.3 beschriebenen Interpretation semantischer Gleichheit an: In r werde ein weiteres, abgeleitetes Attribut a'_h eingeführt, dessen Werteinstanzen einen verschlüsselten Hashwert der jeweiligen Klartext-Werteinstanz für a' beinhalten. Der Hashwert werde dabei über eine moderne kryptografische Hashfunktion wie z. B. SHA-2 berechnet.

Bei jeder INSERT- bzw. UPDATE-Operation auf einer Werteinstanz von a' wird der zugehörige Wert für a'_h auf dem Client berechnet, und beide Werte werden als Teil des zu speichernden Tupels zum Server transferiert. Bei einer Gleichheitssuche berechnet der Client aus dem Suchwert s ebenfalls den verschlüsselten Hashwert $s'_h = Enc_{det}\big(hash(s)\big)$ und schickt ihn als Suchwert zum Server. Dort wird die Extension $X(a'_h)$ anstelle von $X(a')$ nach Werten durchsucht, die s'_h entsprechend bitweiser Gleichheit gleichen.

Dieses Vorgehen kann theoretisch zu Problemen führen, da Hashfunktionen inhärent nicht injektiv sein können, wenn die Kardinalität des Wertebereichs der Funktion kleiner ist als die ihres Definitionsbereichs, so dass die folgende Erweiterung von (F18):

$$x = y \Leftrightarrow hash(x) = hash(y) \qquad \textbf{(F19)}$$

nicht mehr gilt, sondern nur noch:

$$x = y \Rightarrow\ hash(x) = hash(y) \qquad \textbf{(F20)}$$

Die Verwendung von Hashfunktionen für die Suche nach bitweiser Gleichheit kann also theoretisch false positives bewirken, auch wenn die Gefahr dafür sehr gering ist.

5.5 Normalisierung bei Zeichenketten-Datentypen

5.5.1 Sortiervorschriften (Collations)

Wie bereits in Kapitel 5.1 erwähnt, lässt sich für Zeichenketten-Datentypen die Interpretation semantischer Gleichheit nicht, bzw. nur in seltenen Fällen direkt über bitweise Gleichheit umsetzen, sondern erfordert eine Indirektion.

Auf Zeichenketten ist, wie auf numerischen Datentypen, eine lineare Ordnung definierbar, wobei hier eine lexikografische Ordnung nahe liegt. Für letzere ist erforderlich, eine Sortierreihenfolge (Collation) für das zugrundeliegende Alphabet Σ zu haben, die alle Zeichen $\sigma \in \Sigma$ berücksichtigt. In der deutschen Sprache schließt dies insbesondere die Umlaute und „ß“ als Sonderfälle mit ein. Weiterhin kann die Collation Sonderregelungen beinhalten, wie etwa die Festlegung, dass bei der Sortierung zwischen Groß- und Kleinschreibung unterschieden werden soll (oder nicht).

So gilt beispielsweise bei Verwendung der bereits in Kapitel 5.1 erwähnten Norm DIN 5007-1 für Wörter $w \in \Sigma^*$ über dem Alphabet $\Sigma = \{a, \dots, z, \text{ä}, \text{ö}, \text{ü}, \text{ß}, A, \dots, Z, \text{Ä}, \text{Ö}, \text{Ü}\}$:

- Zwischen Groß- und Kleinschreibung wird nicht unterschieden.
- „ä“ wird behandelt wie „a“.
- „ö“ wird behandelt wie „o“.
- „ü“ wird behandelt wie „u“.
- „ß“ wird behandelt wie „ss“.

5.5.2 Normalisierung und Äquivalenzrelationen

Die oben genannten Konventionen zur Gleichbehandlung eigentlich unterschiedlicher Zeichen hat Auswirkungen auf die Interpretation semantischer Gleichheit auf Zeichenkettenwerten: Die Klartext-Datenbankanfrage

```
SELECT * FROM r WHERE a = 'müsse'
```
(Q4)

muss nicht nur Tupel zurückliefern, die für a den Wert „müsse“ haben, sondern auch solche, die in a den Wert „Muße“ haben, da sie entsprechend der oben geltenden Collation als semantisch gleich mit „müsse“ anzusehen sind.

In einer praktischen Anwendung ist eine geltende Collation als Normalisierungsfunktion f implementierbar, die auf die zu prüfenden Werteinstanzen vor ihrem Vergleich anzuwenden ist. f bildet Wörter $w \in \Sigma^*$ auf eine generische Form ab, in der die Funktionswerte von f nach den Regeln bitweiser Gleichheit miteinander verglichen werden können. Daraus folgt, dass die Funktionswerte von f auch definit verschlüsselt werden und dann immer noch mit korrekten Ergebnissen auf bitweise Gleichheit überprüft werden können.

Im angeführten Beispiel von DIN-Norm 5007-1 gilt damit für f:

- Für alle Eingabewörter $w \in \Sigma^*$ bildet f alle Buchstaben eines beliebigen Eingabewortes $w \in \Sigma^*$ auf ihre Kleinbuchstabendarstellung ab.
- f ersetzt in den Buchstaben von w alle Vorkommnisse von „ä“ durch „a“, „ö“ durch „o“, „ü“ durch „u“ und „ß“ durch „ss“.

5.5.3 Darstellung durch Äquivalenzklassen

Die Einführung der Funktion f in Kapitel 5.5.2 soll in diesem Unterkapitel mit einem Blick auf Äquivalenzrelationen näher beleuchtet werden. Sei $eq_f \subseteq \Sigma^* \times \Sigma^*$ eine Äquivalenzrelation auf dem Definitionsbereich $\mathcal{D} = \Sigma^*$ für die Werte eines Zeichenkettenattributs. Entsprechend der Beschreibung in Kapitel 5.5.2 drückt eq_f eine Interpretation semantischer Gleichheit auf diesen Werten aus. f definiert dabei eine Menge von Äquivalenzklassen, die $\mathcal{D}$ partitionieren. Es gilt:

$$\forall a, b \in \mathcal{D}: (a, b) \in eq_f \Leftrightarrow a \sim_{eq_f} b \Leftrightarrow$$

$$f(a) = f(b) \Leftrightarrow Enc_{det}\big(f(a)\big) = Enc_{det}\big(f(b)\big), \quad \textbf{(F21)}$$

wobei die Gleichheit sowohl von $f(a)$ und $f(b)$ als auch von $Enc_{det}\big(f(a)\big)$ und $Enc_{det}\big(f(b)\big)$ als bitweise Gleichheit verstanden werden kann. Die Menge der Äquivalenzklassen

$$\mathcal{D}/\sim_{eq_f} = \{[a] | a \in \mathcal{D}\} \quad \textbf{(F22)}$$

bildet eine Partitionierung von $\mathcal{D}$, in der für jede Äquivalenzklasse

$$[a]_{eq_f} = \{x \in \mathcal{D} | f(x) = f(a)\} = \big\{x \in \mathcal{D} | Enc_{det}\big(f(x)\big) = Enc_{det}\big(f(a)\big)\big\}, a \in \mathcal{D} \quad \textbf{(F23)}$$

der verschlüsselte Funktionswert von a ein kanonischer Repräsentant von $[a]_{eq_f}$ ist.

5.5.4 Praktische Implementierung

Für klartextverarbeitende Informationssysteme ist der Umgang mit Collations gut verstanden und in existierende Produkte integriert. So bieten kommerzielle DBMS-Produkte oft zahlreiche Collations an, nach denen Werteinstanzen eines Zeichenkettenattributs, beispielsweise vom Datentyp $varchar$, verglichen werden können.

Für Attribute, die verschlüsselte Zeichenketten enthalten, tut sich jedoch wieder das Problem auf, dass das DBMS keine Information über die Semantik dessen hat, was gerade verglichen wird, so dass die in der Collation definierten Regeln auf dem Server nicht angewendet werden können. Somit ist der Client der einzige verbleibende Ort in der Informationssystemarchitektur, an dem der Vergleich von Zeichenketten durchgeführt werden kann. Es werden Lösungsansätze für eine praktische Implementierung der Gleichheitssuche gebraucht, die diesen Umstand einbeziehen und gleichzeitig eine gute Performanz bieten.

So können nach einem naiven Ansatz für eine Gleichheitssuche nach einem Suchwert s die verschlüsselten Werteinstanzen der durchsuchten Extension $X(\pi_{a'}(r))$ zum Client transferiert werden, wo sie dann entschlüsselt werden. Anschließend wird f sowohl auf die Klartext-Werteinstanzen als auch auf s angewendet, und die Elemente aus $\{f(Dec(x))|x \in X(a')\}$ werden mit $f(s)$ verglichen, woraus die Treffermenge der Suchanfrage entsteht.

Dieser Lösungsansatz ist offensichtlich teuer und wird daher nicht weiter berücksichtigt. Stattdessen sollte eine Serveranfrage auf verschlüsselten Daten analog zu der Form beschleunigt werden, in der Anfragen wie (Q3) von konventionellen Datenbankindexen beschleunigt werden. Hierfür bietet sich der folgende Ansatz an:

- Beim Speichern des Wortes w in der Extension eines Attributs a' einer Relation r
  - wendet der Client f auf w an und erhält $w_f = f(w)$.
  - Er berechnet anschließend die verschlüsselten Werte $w' = Enc(w)$ und $w'_f = Enc_{det}(w_f)$ und transferiert beide Werte zum Server, welcher sie in der Datenbank speichert.
  - Das binäre Attribut a'_f, in dem w'_f gespeichert wird, ist Teil der Indexrelation $ix^{equal}_{r.a'}$, welche mit r über eine Fremdschlüsselbeziehung assoziiert ist.
  - Auf a'_f sei ein herkömmlicher Datenbankindex, vorzugsweise ein Hash-Index, definiert.
- Bei einer Suche auf a', die (Q4) entspricht, geht das Informationssystem so vor, dass
  - der Suchwert s zunächst vom Client mit f zu $s_f = f(s)$ normalisiert wird.
  - Anschließend verschlüsselt der Client s_f zu $Enc_{det}(s_f) = s'_f$ mit dem gleichen Verschlüsselungsverfahren (und dem gleichen Schlüssel), mit dem w zuvor zu w' versschlüsselt wurde, und transferiert s'_f zum Server.
  - Dort führt das DBMS eine Gleichheitssuche nach den Regeln bitweiser Gleichheit auf der Extension von $\pi_{a'_f}(ix^{equal}_{r.a'})$ durch. Diese Suche wird durch den auf a'_f definierten Hash-Index beschleunigt.
  - Zur Ergebnismenge von a'_f-Werteinstanzen werden die damit assoziierten Tupel aus r ermittelt und an den Client als Treffermenge übermittelt.

Anmerkung: Für die Berechnung von w' kann ein beliebiges Verschlüsselungsverfahren gewählt werden; für die Berechnung von w'_f ist dagegen ein definites Verfahren erforderlich, da hierauf die bitweise Gleichheit ausgewertet wird.

5.6 Schwellwertunterschreitung bei Gleitkommazahlen

Numerische Werte mit einer fixierten Anzahl von Nachkommastellen, wie sie etwa in den SQL-92-Datentypen *decimal* bzw. *numeric* dargestellt werden können, wurden in Kapitel 5.3 behandelt. Sie können als ganzzahlige Werte interpretiert werden, bei denen lediglich das Komma um eine definierte Anzahl von Stellen nach links verschoben wurde. Deshalb kann semantische Gleichheit auf *decimal*- bzw. *numeric*-Attributen, wie in Kapitel 5.3 gezeigt, leicht in Form von bitweiser Gleichheit implementiert werden.

Im Gegensatz dazu stehen Gleitkommazahlwerte, die in SQL-92 im Datentyp $float$ dargestellt werden[53]. Sie werden durch das Tripel (v, m, e) repräsentiert, mit v als Vorzeichen, m als Mantisse und e als Exponent zur Basis 2. Üblicherweise[54] sieht die gewählte Darstellungskonvention vor, dass (neben weiteren Spezifikationen) von e ein konstanter Bias-Wert b abgezogen werde, so dass Gleitkommazahlen das folgende Produkt zur Darstellung reeller Zahl verwenden:

$$(-1)^v \cdot m \cdot 2^{e-b} \qquad \textbf{(F24)}$$

Damit sind Gleitkommazahlen aufgrund ihrer limitierten Präzision weniger als exakte Abbilder denn als Näherungswerte zu reellen Zahlen zu verstehen. Die Exaktheit der Näherung ist parametrisierbar über die Anzahl der Bits, die zur Zahldarstellung zur Verfügung stehen; sie wird bei der Deklaration eines $float$-Attributs angegeben, beispielsweise als $float(53)$.

Der approximative Charakter von Gleitkommazahlen macht Definition und Implementierung einer Interpretation semantischer Gleichheit darauf schwierig. Eine Interpretation in Form von bitweiser Gleichheit wäre etwa keine gute Wahl, da sie enorm fehleranfällig ist: Mit ihr wäre es wahrscheinlich, dass zwei Werte semantisch als gleich anzusehen, bitweise aber verschieden wären und somit als nicht gleich bewertet würden. Es würden also viele "false negative"-Fehler auftreten. So produzieren beispielsweise unter Microsoft SQL Server die Zuweisungen zu zwei Variablen $@x$ und $@y$ vom Datentyp $float(53)$*:*

```
SET @x = 1.0 / 7.0
```
(Q5)

und

```
SET @y = 1000.0 / 7000.0
```
(Q6)

zwei bitweise unterschiedliche Werte, und der binäre Vergleich $@x = @y$ liefert entsprechend $false$. Dieses Verhalten dürfte in den meisten Fällen der praktischen Benutzung von $float$-Datentypen unerwünscht sein. Folglich ist bitweise Gleichheit auch im Kontext verschlüsselter Daten keine Option für die Implementierung semantischer Gleichheit auf Gleitkommazahlwerten: Auch hier werden Werte als nicht gleich angesehen, die von einem pragmatisch-numerischen Verständnis her als gleich angesehen werden sollten.

---

[53] Der weitere Fließkommazahl-Datentyp $real$ ist lediglich eine Variante von $float$.

[54] Entsprechend der Norm IEEE754, die die Darstellung von Gleitkommazahlen standardisiert (siehe [URL-IEEE-754]).

5.6.1 Semantische Gleichheit über fuzzy match

Stattdessen wird in der Klartextwelt oft eine als *fuzzy match* bezeichnete Methode vorgeschlagen. Eine weitverbreitete Implementierung davon ist die sogenannte *Epsilon-Methode* (*ε-Methode*), bei der zwei Gleitkommazahlwerte x und y genau dann als gleich angesehen werden, wenn ihr Abstand einen definierten Schwellwert ε unterschreitet:

$$equals(x, y) \Leftrightarrow |x - y| < \varepsilon \quad \textbf{(F25)}$$

Für ein Informationssystem, dessen Datenbank-Backend auf Klartextdaten operiert, kann dies beispielsweise in der folgenden SQL-Anfrage implementiert werden:

```
SELECT (…) FROM r WHERE ABS(x - 0.5019357112e002) < ε
```
(Q7)

Die ε-Methode kann selbst als eine datentypspezifische Implementierung semantischer Gleichheit auf Klartextdaten angesehen werden. Sie benutzt eine funktionale Indirektion, um die Gleichheit von $float$-Werten zu bestimmen, die sich folgendermaßen beschreiben lässt:

$$equal_\varepsilon: \mathbb{R}^2 \to \{true, false\}, equal_\varepsilon(x, y) := \begin{cases} true, & |x - y| < \varepsilon \\ false, & |x - y| \geq \varepsilon \end{cases} \quad \textbf{(F26)}$$

5.6.2 Vordefinierte Partitionierung des Wertebereichs

Beim Versuch, die ε-Methode in der verschlüsselten Welt nachzubilden, besteht die Schwierigkeit darin, die darin enthaltene Subtraktionsoperation und die „<“-Relation in den Kontext verschlüsselter Daten zu übertragen. Der Einsatz homomorpher Verschlüsselungsalgorithmen hierzu wird aufgrund deren mangelnder Performanz und geringer Akzeptanz nicht in Betracht gezogen. Stattdessen wird eine Normalisierungsfunktion f_{float} für $float$-Werte ähnlich der für Zeichenketten (siehe Kapitel 5.5) angestrebt, mit der die Funktionswerte $Enc_{det}\left(f_{float}(x)\right)$ und $Enc_{det}\left(f_{float}(y)\right)$ der $float$-Werte x und y nach den Regeln der bitweisen Gleichheit effizient auf semantische Gleichheit geprüft werden können:

$$Enc_{det}\left(f_{float}(x)\right) = Enc_{det}\left(f_{float}(y)\right)$$

$$\Leftrightarrow f_{float}(x) = f_{float}(y) \Leftrightarrow |x - y| < \varepsilon^{55} \quad \textbf{(F27)}$$

Diese Vorgehensweise ist inspiriert von einem Ansatz, den die Autoren von [Hacigümüş2002-2] bereits im Jahr 2002 verfolgt haben (siehe Kapitel 2). Dort wird der Wertebereich eines Datentyps zunächst in benannte Partitionen aufgeteilt und die zu indizierenden Klartextwerte auf deren jeweilige Partition abgebildet.

Die Idee einer vorgegebenen Partitionierung des Wertebereichs hat jedoch den Nachteil, dass sie false positives verursachen kann: Eine Partition $p_i = [a, b[$ des Wertebereichs mit $b - a > 2\varepsilon$ kann zwangsläufig Werte beinhalten, die von einem $x \in p_i$ einen größeren Abstand als ε haben. Darüber hinaus können diese false positives erst auf dem Client erkannt werden, da sie nur dort im Klartext vorliegen. Potenzielle, vom Server ermittelte Trefferkandidaten müssen deshalb zum Client transferiert werden, um dort gefiltert zu werden.

---

[55] Es bietet sich an, ε anstelle eines konstanten Wertes eher als einen funktional abgeleiteten Wert von x anzusehen, da seine Größe stark von der Größenordnung von x und der Präzision der gegenwärtig vorliegenden $float$-Implementierung abhängt. Eine mögliche Definition von ε ist etwa: $\varepsilon: \mathbb{R} \to \mathbb{R}, \varepsilon(x) = 10^{\lfloor \log_{10} x \rfloor - 10}$.

Darüber hinaus produziert die Methode auch false negatives, die selbst mit Hilfe des Clients nicht als solche erkannt werden können: Zu einem $x \in p_i = [a_i, b_i[$, das nah am linken Rand, also a_i, liegt, kann es Werte y geben, die nahe des rechten Rands von $p_{i-1} = [a_{i-1}, b_{i-1}[$ liegen und einen kleineren Abstand als ε zu x haben. Dennoch werden solche Werte vom Server nicht als Treffer-Kandidaten erkannt, da sie in der falschen Partition liegen. Eine Abhilfe hierfür ist, zu einer zu einem Suchwert x ermittelten Partition p_i diejenigen Werte ebenfalls zum Client zu transferieren, die in die umgebenden Partitionen p_{i-1} und p_{i+1} gehören. Dies vermeidet zwar false negatives, erhöht aber die Anzahl von false positives erheblich.

5.6.3 Alternativer Ansatz

Um die oben genannten Nachteile bezüglich false positives und false negatives zu vermeiden, wird in diesem Kapitel eine alternative Methode vorgeschlagen, die SQL-Anfragen auf verschlüsselten Extensionen von Attributen mit präzisen Ergebnismengen erlaubt, also ohne false positives und false negatives. Zur Indizierung eines verschlüsselten Attributs a' mit Klartext-Datentyp $float$ in einer Datenrelation r wird dafür die in Kapitel 5.6.2 genannte Normalisierungsfunktion f_{float} folgendermaßen definiert:

$$f_{float}: \mathbb{R} \to \mathbb{R} \times \mathbb{R} \times [0, \varepsilon[,$$

$$f_{float}(x) := \left(\left\lfloor \frac{x}{\varepsilon} \right\rfloor \varepsilon \, , \left(\left\lfloor \frac{x}{\varepsilon} \right\rfloor + 1 \right) \varepsilon \, , x - \left\lfloor \frac{x}{\varepsilon} \right\rfloor \varepsilon \right) = \left(x_{low}, x_{high}, x_{rel} \right) \qquad \textbf{(F28)}$$

Mit anderen Worten, f_{float} bildet einen $float$-Wert x auf drei weitere Werte x_{low}, x_{high} und x_{rel} ab, von denen x_{low} und x_{high} das jeweilige nächstgelegene ganzzahlige Vielfache von ε sind, das vor bzw. hinter x liegt. Sei im Folgenden das sogenannte *ε-Intervall* von x als das halboffene Intervall $I_\varepsilon(x) = [x_{low}, x_{high}[$ definiert. Der dritte Wert x_{rel} gibt die relative Position von x in $I_\varepsilon(x)$ an. Es gelte: $0 \leq x_{rel} < \varepsilon$ und: $x_{rel} = 0 \Leftrightarrow x = x_{low}$.

Die Funktion $f_{float}(x)$, die aus x und ε das Tripel $\left(x_{low}, x_{high}, x_{rel} \right)$ berechnet, wird nun verwendet, um das verschlüsselte Attribut a' zu indizieren. Bei der Speicherung eines Tupels t_x mit $\pi_{a'}(t_x) = x' = Enc(x)$ in r wird dazu ein weiteres Tupel $t_{ix,x} = \left(x'_{low}, x'_{high}, x_{rel} \right) = \left(Enc_{det}(x_{low}), Enc_{det}(x_{high}), x_{rel} \right)$ in einer Indexrelation $ix_{r.a'}^{equal}$ gespeichert.

$ix_{r.a'}^{equal}$ und r seien durch die M:N-Relation $mn_{r.a'}^{equal}$ verknüpft (siehe Kapitel 4.2.4), so dass einem beliebigen Indextupel $t_{ix,x}$ schnell genau ein Datentupel t_x zugeordnet werden kann. a'_{low}, a'_{high} und a_{rel}, die Attribute in $ix_{r.a'}^{equal}$ zur Speicherung von x'_{low}, x'_{high} und x_{rel}, seien jeweils mit einem konventionellen Datenbankindex belegt, der schnelle Zugriffspfade darauf erlaubt. Für x'_{low} und x'_{high} bieten sich Hash-Indexe an, für x_{rel} ein B- oder B⁺-Baum-Index.

Nun folgt die Definition der Verwendung von f_{float}, um semantische Gleichheit auf verschlüsselten $float$-Wertinstanzen festzustellen. Vom ersten Entwurf der oben vorgestellten „fuzzy match"-Methode in (F27), weicht sie in der folgenden Form ab:

$$\begin{aligned} & (x'_{low} = y'_{low}) \vee \left(x'_{low} = y'_{high} \wedge x_{rel} < y_{rel} \right) \vee \left(x'_{high} = y'_{low} \wedge x_{rel} > y_{rel} \right) \\ \Leftrightarrow \; & x_{low} = y_{low} \vee \left(x_{low} = y_{high} \wedge x_{rel} < y_{rel} \right) \vee \left(x_{high} = y_{low} \wedge x_{rel} > y_{rel} \right) \\ \Leftrightarrow \; & |x - y| < \varepsilon \end{aligned} \qquad \textbf{(F29)}$$

Diese Äquivalenz wird im Folgenden näher erläutert. Angenommen, x sei ein fester Wert (so wie es in der SQL-Suchanfrage in (Q2) bezüglich des eingesetzten Suchparameters s der Fall ist); dann haben diejenigen $float$-Werte $y \in \{y = Dec(y') \mid y' \in \pi_{a'}(X(r))\}$, die (F29) erfüllen, die folgenden Eigenschaften:

1. Falls der erste Operand ($x_{low} = y_{low}$) der Disjunktion erfüllt ist, dann befinden sich x und y im selben ε-Intervall. Wegen $x_{high} - x_{low} < \varepsilon$ gilt $|x - y| < \varepsilon$.
2. Falls der zweite oder der dritte Operand der obigen Disjunktion erfüllt ist, dann befinden sich x und y in zwei angrenzenden ε-Intervallen, und es gilt:

$$|x - y| = \begin{cases} y_{rel} - (\varepsilon - x_{rel}), & x < y \\ x_{rel} - (\varepsilon - y_{rel}), & x > y \end{cases} \Rightarrow |x - y| = y_{rel} + x_{rel} - \varepsilon < \varepsilon \qquad \textbf{(F30)}$$

In einer praktischen DBMS-Implementierung der oben angegebenen Lösung wird für eine SQL-Anfrage, die in der Datenrelation r nach Werten sucht, die dem Suchparameter s „gleichen" zunächst $f_{float}(s) = (s_{low}, s_{high}, s_{rel})$ berechnet. Anschließend werden s_{low} und s_{high} zu $Enc_{det}(s_{low}) = s'_{low}$ und $Enc_{det}(s_{high}) = s'_{high}$ definit verschlüsselt. Schließlich wird die folgende SQL-Anfrage beim DBMS abgesetzt:

```
SELECT (…)
FROM r JOIN mn r.a' ON (…) JOIN ix r.a' AS ix ON (…)
WHERE ix.a'low = s'low
   OR (ix.x'high = s'low AND ix.arel > srel)
   OR (ix.x'low = s'high AND ix.arel < srel)
```
(Q8)

Es ist zu beachten, dass sämtliche serverseitigen Operationen, die sich aus der WHERE-Klausel ergeben, entweder Prüfungen von definit verschlüsselten Zahlenwerten auf Gleichheit sind, oder die „<"- bzw. „>"-Ungleichheitsrelation auf Klartextdaten. Erstere können als bitweise Gleichheitsprüfung implementiert werden (und profitieren somit von einem Hash-Index auf dem betreffenden Attribut); für letztere existieren die herkömmlichen Möglichkeiten zur Zugriffsbeschleunigung, etwa über B- bzw. B⁺-Baum-Indexe. Somit profitieren alle im WHERE-Kriterium aufgeführten Operationen von herkömmlichen DBMS-Indexen, weswegen die SQL-Anfrage schnell ausgeführt werden kann.

5.6.4 Informationsverlust

Im gesamten Kapitel 5 stellt sich angesichts des Einsatzes definiter Verschlüsselung die Frage, ob der mit Letzterer einhergehende Informationsverlust eine signifikante Beeinträchtigung der Vertraulichkeit der indexierten Daten bedeuten kann (Kapitel 10 widmet sich dieser Thematik ausführlicher). Dies scheint auch im Falle der im vorangegangenen Unterkapitel vorgestellten Kodierung von Gleitkommazahlen unter Verwendung von exponierter Klartextinformation zuzutreffen. Tatsächlich jedoch trägt diese Klartextinformation neben der Nutzung definiter Verschlüsselung kaum etwas zum Informationsverlust bei:

Eine Gleitkommazahl x wird hier vor allem beschrieben durch x_{low} und x_{high}, die beiden Vielfachen von ε mit $x_{low} \leq x < x_{high}$. Es werden also zwei quasi-diskrete Werte angegeben, die sehr nah beieinander liegen, zusammen mit der Information, dass der indexierte Wert x zwischen ihnen liegt. Die zusätzlich in x_{rel} exponierte Klartextinformation besagt lediglich, an welchem der beiden Werte x näher liegt und in welchem Maße. Ohne Kenntnis von x_{low} und/oder x_{high} bringt diese Information einem Angreifer jedoch wenig, verschiebt sie den Wert von x doch nur um einen Bruchteil des ebenfalls unbekannten Wertes ε.

5.6.5 cTree als weitere Alternative

Eine Alternative zur in Kapitel 5.6.3 vorgestellten Lösung ist eine verschlüsselte Index-Datenstruktur, die die lineare Ordnung der indizierten Daten in zwei verschiedenen Klartextrepräsentationen beinhaltet. Sie wird ausführlich in Kapitel 6 unter dem Namen *cTree* behandelt.

Der ursprüngliche Zweck des cTree ist die Implementierung von Ungleichheits- und Intervallsuchen auf linear geordneten Daten, mit denen sich auch Präfixsuchen auf lexikografisch geordneten Daten implementieren lassen. Dennoch könnte er mit einigen Modifikationen auch auf anderen Datentypen, etwa $float$, für eine Gleichheitssuche eingesetzt werden.

Gegen die Verwendung der cTree-Datenstruktur spricht allerdings, dass darauf basierende Gleichheitssuchen deutlich höhere Laufzeiten aufweisen als die in Kapitel 5.6.3 vorgestellte Methode, die direkt von der Beschleunigung herkömmlicher Datenbankindexe profitiert.

5.7 Indexerstellung und -wartung

5.7.1 Indexerstellung

Entsprechend der in Kapitel 4.3 in Schritt 1 vorgestellten generischen Vorgehensweise zur Indexerstellung wird für ein beliebig verschlüsseltes Attribut a' einer Datenrelation r, das mit einem Index zur Gleichheitssuche belegt werden soll, eine Indexrelation $ix_{r.a'}^{equal}$ erzeugt. Sie enthält die erforderlichen Indexattribute in Abhängigkeit des Datentyps von a': entweder ein einzelnes Attribut, das den definit verschlüsselten Klartext von a' enthält, bzw. eine definit verschlüsselte Ableitung davon, oder, im Falle eines indexierten Fließkommaattributs, drei Attribute a'_{low}, a'_{high} und a_{rel} wie in Kapitel 5.6.3 beschrieben.

Anschließend wird entsprechend Schritt 2 von Kapitel 4.3 eine M:N-Relation $mn_{r.a'}^{equal}$ erstellt, die r und $ix_{r.a'}^{equal}$ referenziert. Weiterhin wird entsprechend Schritt 3 in Kapitel 4.3 ein Tupel in die Verwaltungsrelation $indexAdmin$ eingefügt, das die in Kapitel 4.2.6 spezifizierte Statusinformation über den Gleichheitsindex auf $r.a'$ enthält.

5.7.2 Indexbenutzung und -pflege

Da die Indexwerte verschlüsselt sind, muss der Client einen Teil der Indexpflege bei den üblichen Kommandos der Datenmanipulationssprache (*Data Manipulation Language*, *DML*) und der Datenanfragesprache (*Data Retrieval Language*, *DRL*), also den Operationen $INSERT$, $UPDATE$, $DELETE$ und $SELECT$, in der folgenden Form übernehmen:

5.7.2.1 INSERT

Sei t ein Tupel mit Attributwerten, das in r eingefügt wird und $x' = \pi_{a'}(t)$ die Projektion von t auf das für Gleichheitssuche indizierte Attribut a'. Dann sind für die INSERT-Operation folgende Schritte auszuführen:

- Berechne und verschlüssele zu $x = Dec(x')$ den datentypspezifischen Indexwert x'_{ix}. Entsprechend den vorangegangenen Unterkapiteln handele es sich bei x'_{ix} entweder um den definit verschlüsselten Wert $x' = Enc_{det}(x)$, um den definit verschlüsselten Hashwert von x, $x'_h = Enc(x_h) = Enc_{det}(h(x))$, um die definit verschlüsselte Normalisierung von x, $x'_f = Enc(x_f) = Enc_{det}(f(x))$ oder um das in Kapitel 5.6.3 eingeführte Tripel (x'_{low}, x'_{high}, x_{perc}). Letzteres ist in der Indexrelation in drei separaten Attri-

buten zu speichern. Die Berechnung von x'_{ix} wird vom Client durchgeführt, da nur er auf den dafür erforderlichen Klartext x zugreifen kann.

- Prüfe, ob x'_{ix} bereits in der Extension $X\left(ix_{r.a}^{equal}\right)$ enthalten ist. Dies ist erforderlich, weil die in 4.2.3 formulierte Forderung nach Eindeutigkeit der Indexwerte in der Extension der Indexrelation gilt.
    - Falls x'_{ix} nicht in $X\left(ix_{r.a'}^{equal}\right)$ enthalten ist, füge ein entsprechendes Indextupel t_{ix} mit $\pi_{a'}(t_{ix}) = x'_{ix}$ in $ix_{r.a'}^{equal}$ ein. Füge anschließend ein weiteres Tupel t_{mn} in $mn_{r.a'}^{equal}$ ein, das t und t_{ix} verknüpft.
    - Füge anderenfalls nur ein t_{ix} und t verknüpfendes Tupel t_{mn} in $mn_{r.a'}^{equal}$ ein.

5.7.2.2 UPDATE

Die Aktualisierungsoperation UPDATE auf dem Attributwert x' für Attribut a' in Tupel t, das bereits per M:N-Tupel t_{mn} mit einem Indextupel $t_{ix_{alt}}$ verknüpft ist, wird analog zur INSERT-Operation erweitert:

- Berechne einen neuen Indexwert $x'_{ix_{neu}}$ für den neuen Wert von $x = Dec\left(\pi_{a'}(t)\right)$.
- Falls $x'_{ix_{neu}}$ noch nicht in $ix_{r.a'}^{equal}$ enthalten ist, speichere $x'_{ix_{neu}}$ in einem neuen Indextupel $t_{ix_{neu}}$.
- Ermittle ansonsten das Indextupel $t_{ix_{neu}}$ in $X\left(ix_{r.a'}^{equal}\right)$ mit $\pi_{a'}\left(t_{ix_{neu}}\right) = x'_{ix_{neu}}$.
- Aktualisiere t_{mn}, so dass der Fremdschlüssel auf $ix_{r.a'}^{equality}$ nunmehr auf $t_{ix_{neu}}$ verweist. $t_{ix_{alt}}$ ist anschließend für t obsolet und kann gelöscht werden, jedoch nur dann, wenn keine weiteren M:N-Tupel darauf verweisen.[56]

5.7.2.3 DELETE

Bezüglich der Löschung des Datentupels t aus r sind folgende Schritte durchzuführen:

- Lösche das M:N-Tupel t_{mn} aus $mn_{r.a'}^{equal}$, das t und sein Indextupel t_{ix} verknüpft.
- Lösche t aus r.
- Lösche das Indextupel t_{ix}, auf das t_{mn} bisher verwiesen hat. Wie bei der Aktualisierungsoperation gilt jedoch, dass dies nur dann erlaubt ist, wenn keine weiteren M:N-Tupel auf t_{ix} verweisen. Ebenso ist das Inkaufnehmen von verwaisten Indextupeln wie in Kapitel 5.7.2.2 eine alternative Strategie.

5.7.2.4 SELECT

Die Klartextversion der Datenanfrageoperation $SELECT$ (siehe (Q2)) sieht vor, dass das DBMS bei einem gegebenen Suchwert s eine Menge von Tupeln $\{t_1, \ldots, t_n\}$ aus der angefragten Datenrelation r zurückgibt, bei denen der jeweilige Wert des Attributs a mit s das Kriterium der Gleichheit (d. h. die jeweilige Implementierung des Konzepts der semantischen Gleichheit) erfüllt. Weiterhin können im DBMS mit konventionellen Datenbankindexen Zusatzstrukturen angelegt werden, die die Ausführung dieser Gleichheitssuche beschleunigen.

---

[56] Es ist je nach Anwendungsszenario auch denkbar, aus einer optimistischen Perspektive heraus ein Indextupel, auf das keine M:N-Tupel mehr verweisen, nicht zu löschen, wenn die Wahrscheinlichkeit, dass es in Zukunft wieder gebraucht werden wird, als hoch angesehen werden kann.

Im bisherigen Verlauf des Kapitels 5 wurden Mittel eingeführt, um dieses Verhalten in der verschlüsselten Welt nachbilden und ebenfalls mit Hilfe von Zusatzstrukturen beschleunigt ausführen zu können. Eine Implementierung einer solchen Gleichheitssuche auf verschlüsselten Daten in Form einer SQL-Anweisung hat die folgenden Eigenschaften:

- In der FROM-Klausel wird ein Verbund (JOIN) aus Daten-, M:N- und Indexrelation, also r, $mn_{r.a'}^{equal}$ und $ix_{r.a'}^{equal}$, hergestellt.
- In der SELECT-Klausel findet eine Projektion auf eine Teilmenge der aus der Datenrelation stammenden Attribute statt.
- Die Selektion der Ergebnis-Tupel, die in der WHERE-Klausel formuliert wird, findet anhand von Kriterien statt, die sich auf die Indexrelation beziehen.

Beispielsweise wird die folgende SQL-Anfrage ausgeführt:

```
SELECT r.*
FROM r
  JOIN mn_r.a' AS mn ON mn.refData  = d.seq
  JOIN ix_r.a' AS ix ON mn.refIndex = ix.seq
WHERE ix.a'_f = x'                                                    (Q9)
```

Dieses Beispiel bezieht sich auf einen Index auf Zeichenketten entsprechend Kapitel 5.5. Bei anderen Datentypen ist das WHERE-Kriterium entsprechend anders ausgebildet.

Wenn, wie bei der in (Q9) dargestellten Gleichheitssuche, nicht nur auf einem einzigen indizierten Attribut a' von r gesucht werden soll, sondern auf mehreren Attributen wie etwa a', b' und c' (sei beispielsweise a' ein Zeichenkettenattribut, b' ein diskret numerisches und c' ein Gleitkommazahlattribut), so sind in der FROM-Klausel einer entsprechenden SQL-Anweisung zusätzliche JOIN-Operationen mit den entsprechenden Indexrelationen nötig.

Die WHERE-Klausel dieser SQL-Anfrage enthält die Selektionskriterien für die drei Attribute a', b' und c'. Sie können in einer beliebigen logischen Kombination angegeben werden (beispielsweise, wie in diesem Beispiel, in einer Konjunktion), so dass sich die folgende beispielhafte SQL-Anfrage ergibt:

```
SELECT r.*
FROM r
  JOIN mn_r.a' AS mn_1 ON mn_1.refData  = d.seq
  JOIN ix_r.a' AS ix_1 ON mn_1.refIndex = ix_1.seq
  JOIN mn_r.b' AS mn_2 ON mn_2.refData  = d.seq
  JOIN ix_r.b' AS ix_2 ON mn_2.refIndex = ix_2.seq
  JOIN mn_r.c' AS mn_3 ON mn_3.refData  = d.seq
  JOIN ix_r.c' AS ix_3 ON mn_3.refIndex = ix_3.seq
WHERE ix_1.a'_f = x'
  AND ix_2.b' = y'
  AND (ix_3.c'_low = z'_low
  OR (ix_3.c'_high = z'_low AND ix_3.c_perc > z_rel)
  OR (ix_3.c'_low = z'_high AND ix_3.c_perc < z_rel))[57]              (Q10)
```

[57] Da jedes Tupel in r über jede M:N-Relation stets mit genau einem Indextupel verknüpft ist, sind durch den JOIN verursachte Duplikate in dieser Anfrage ausgeschlossen.

6 Indexe für die Suche auf Ungleichheit: cTree

6.1 Einführung

In diesem Kapitel wird die *cTree*-Indexdatenstruktur (auch: der cTree-Index bzw. der cTree) vorgestellt. Sie ist die zentrale Komponente, auf der alle in dieser Dissertation vorgestellten, verschlüsselten Indexstrukturen basieren, die Ungleichheitssuchen bzw. Varianten davon auf linear geordneten Daten ermöglichen. Der cTree unterliegt dem Paradigma, dass bei seiner Erstellung, Pflege und Verwendung stets eine Arbeitsteilung zwischen Client und Server vorherrscht. Dem Server wird strukturelle, jedoch keine inhaltliche Information über die Indexeinträge im Klartext offenbart, so dass der cTree die Vertraulichkeit der Indexeinträge nicht bzw. nur unwesentlich beeinträchtigt.

6.1.1 Abgrenzung zu OPE-Verfahren

Beim cTree wird ähnlich zu OPE-Verschlüsselungsverfahren die lineare Ordnung, der die indexierten Elemente im Klartext unterliegen, in den Indexeinträgen im Klartext auf dem Server exponiert, so dass dieser sie für die beschleunigte Bearbeitung von Datenbankanfragen heranziehen kann. Dagegen bleiben die Inhalte der Indexeinträge genau wie die Werteinstanzen, die sie indexieren, verschlüsselt, weshalb der Client ihre inhaltliche Verarbeitung übernehmen muss.

Dabei gebietet die Forderung nach Praxistauglichkeit des cTree-Indexes, dass beliebige Verfahren für die Verschlüsselung der Indexeinträge gewählt werden können; man sollte nicht wie im Falle von OPE-basierten Verfahren zur Verwendung einer bestimmten Gruppe von Algorithmen gezwungen sein, noch dazu, wenn diese neben der linearen Ordnung der Indexeinträge auch Teile der verschlüsselten Klartextinformation offenbaren[58]. Gegen OPE-Verfahren spricht weiterhin ihr Mangel an Verbreitung, Akzeptanz und Erfüllung gesetzlicher Vorgaben.

Stattdessen sollte ein praxistaugliches System, das den cTree-Index einsetzt, auf beliebige etablierte Standards für (symmetrische) Verschlüsselung zurückgreifen können. Im Rahmen der praktischen Implementierung der in dieser Dissertation vorgestellten cTree-Indexstrukturen wurde für die Verschlüsselung der cTree-Indexeinträge das symmetrische Kryptosystem AES im CBC-Modus mit einer Schlüssellänge von 256 Bit eingesetzt. Sollte sich darüber hinaus ein einmal gewähltes Verfahren als unsicher herausstellen, darf es im cTree-Index keine Beschränkungen geben, es gegen ein anderes Verfahren auszutauschen.

Weiterhin kann beim cTree im Gegensatz zu den Indexen für Gleichheitssuche aus Kapitel 5 sowohl definite als auch randomisierte Verschlüsselung verwendet werden.[59]

6.1.2 Prinzipieller Aufbau

Wie eingangs erwähnt, exponiert die cTree-Indexrelation neben den AES-256-verschlüsselten Indexwerten deren lineare Ordnung in unverschlüsselter Form. Aufgrund der freien Wahl des Verschlüsselungsverfahrens ist die lineare Ordnung nicht wie bei OPE-Verfahren inhä-

[58] siehe [Popa2013], S. 1.

[59] In Kapitel 6.5.3 wird ein Sonderfall behandelt, in dem sich der Einsatz einer definiten Verschlüsselung der im cTree-Indexeinträge lohnt. Abgesehen von diesem Sonderfall können beliebige Verfahren gewählt werden.

rent in den Kryptotexten der verschlüsselten Indexeinträge enthalten, sondern muss innerhalb des cTree in einer separaten Struktur repräsentiert werden.

Tatsächlich gibt es im cTree sogar zwei Darstellungen der linearen Ordnung: eine in Form eines binären Baums[60], dessen Knoten die Indexeinträge repräsentieren, und die andere als lineare Liste von unverschlüsselten Surrogatwerten, deren Inhalte nicht auf die tatsächlichen Indexinhalte schließen lassen, sondern nur deren lineare Ordnung abbilden.

Im DBMS wird der cTree-Index in einer Datenbankrelation persistiert. Diese nimmt die Position der in Kapitel 4 eingeführten Indexrelation ein und wird zur Speicherung einer cTree-Index-Instanz verwendet, die ein oder mehrere Attribute in einer Datenrelation für die Ungleichheitssuche indexiert[61]. Mit der Datenrelation ist die cTree-Indexrelation über eine M:N-Relation verbunden wie in Kapitel 4 beschrieben, wobei ein Indextupel von mehreren Tupeln der Datenrelation referenziert sein kann.[62]

6.1.3 Einsatzmöglichkeiten

Der cTree-Index ermöglicht beschleunigte Ungleichheitssuchen auf verschlüsselten Daten, wobei insbesondere „größer"- bzw. „kleiner"-Vergleiche („>" und „<") möglich sind. Diese lassen sich leicht zu „größer oder gleich"- („≥") bzw. „kleiner oder gleich"-Vergleichen („≤") erweitern. Durch Kombination eines „größer (oder gleich) als s_1"-Vergleichs mit einem „kleiner (oder gleich) als s_2"-Vergleich können beschleunigte Intervallsuchen der Form $a \in [s_1, s_2]$, $a \in [s_1, s_2[$, $a \in]s_1, s_2]$ oder $a \in]s_1, s_2[$ auf der Extension eines verschlüsselten Attributs a' einer Datenrelation r durchgeführt werden, das einer linearen Ordnung unterliegt. Diese Suchen auf verschlüsselten Daten entsprechen (für das oben genannte Kriterium $a \in [s_1, s_2[$) der folgenden schematischen Klartext-SQL-Anfrage:

```
SELECT (…)FROM r WHERE a >= s1 AND a < s2
```
(Q11)

Eine besondere Anwendungsmöglichkeit des cTree-Index bietet sich bei Daten, die einer linearen Ordnung in Form einer lexikografischen Ordnung unterliegen: Mit Hilfe des cTree-Indexes können hier reguläre Ausdrücke der Form „$abc\Sigma^*$", also eine Präfixsuche, ebenfalls beschleunigt ausgewertet werden. Sie bilden damit ein Pendant zur folgenden schematischen Klartext-SQL-Anfrage auf verschlüsselten Daten:

```
SELECT (…) FROM r WHERE a LIKE s + '%'
```
(Q12)

Kombiniert mit der in Kapitel 7 vorgestellten „Shrinking Window"-Methode kann dieser Ansatz zu einer Infix- und Postfixsuche erweitert werden.

6.2 Die cTree-Indexrelation

Dieses Unterkapitel betrachtet die cTree-Datenbankrelation als Repräsentation der cTree-Indexdatenstruktur im DBMS genauer. Im Folgenden beziehen sich alle praktischen Beispie-

[60] Die Darstellung der linearen Ordnung als binärer Baum ist die zentale Eigenschaft der Relation während ihrer Verarbeitung; daher wurde dem ganzen Konstrukt der Name cTree gegeben.

[61] Darüber hinaus kann die cTree-Indexrelation auch die Extensionen mehrerer cTree-Indexinstanzen aufnehmen.

[62] Dies ist zumindest in der Basisversion von cTree der Fall; in einer späteren Ausbaustufe wird die Option angeboten, zur Verschleierung von inferierbarer Information den gleichen Inhalt redundant in der Indexrelation zu speichern.

le auf den cTree-Index $ix_{r.a'}^{cTree}$ (bzw. auf seine Datenbankrelation), der zur Indexierung eines beliebig verschlüsselten Attributs a' in einer Datenrelation r angelegt wurde. Weiterhin gelte im Folgenden, dass a' einer linearen Ordnung unterliege.

6.2.1 Attribute

Die cTree-Datenbankrelation enthält die folgenden Attribute inklusive Datentypen:

- seq ($integer$): Kandidatenschlüssel zur Tupelidentifizierung. Optional ist auf seq ein konventioneller, geclusterter Datenbankindex definiert.
- $leftChild$, $rightChild$, $parent$ (alle $integer$): Fremdschlüsselattribute, die andere Tupel der cTree-Relation referenzieren. Sie bzw. ihre Extensionen konstituieren die in Kapitel 6.1.2 genannte Binärbaum-Darstellung der linearen Ordnung, der die Elemente der von $X(\pi_{a'}(r))$ unterliegen. Diese Darstellung wird zum schnellen Auffinden von Indextupeln verwendet (siehe Kapitel 6.3). Auf $parent$ ist ein konventioneller Datenbankindex definiert.
- $ixData$ ($varbinary$): Dieses Attribut enthält den verschlüsselten, von $Dec(a')$ abgeleiteten Indexwert und damit den eigentlichen Informationsgehalt der Indexeinträge. Im normalen Betriebsmodus des cTree-Indexes ist auf $ixData$ kein konventioneller Index definiert; in einer Modifikation, die in Kapitel 6.5.3.1 vorgestellt wird, kann sich dies jedoch ändern.
- $sortNumber$ ($bigint$): Dieses Attribut bildet die lineare Ordnung, der die Elemente von $X(a')$ unterliegen, auf eine andere Menge von (Surrogat-)Werten ab, die der gleichen linearen Ordnung unterliegen. Hierbei gelten für zwei beliebige cTree-Indextupel t_1 und t_2 die folgenden Äquivalenzen bezüglich ihrer entschlüsselten Werte für $ixData$, $d_1 = Dec(\pi_{ixData}(t_1))$ und $d_2 = D(\pi_{ixData}(t_2))$, sowie $sn_1 = \pi_{sortNumber}(t_1)$ und $sn_2 = \pi_{sortNumber}(t_2)$:

  $$d_1 < d_2 \Leftrightarrow sn_1 < sn_2,$$
  $$d_1 > d_2 \Leftrightarrow sn_1 > sn_2 \text{ und}$$
  $$d_1 = d_2 \Leftrightarrow sn_1 = sn_2 \quad \textbf{(F31)}$$

  Diese Darstellung der linearen Ordnung wird bei Intervallsuchen zur Parametrisierung des WHERE-Kriteriums einer Datenbankanfrage benötigt. Weiterhin wird $sortNumber$ zur serverseitigen Sortierung von Tupelmengen verwendet, dient also zur Parametrisierung des ORDER BY-Kriteriums einer Anfrage (siehe Kapitel 6.3). Auf $sortNumber$ ist ein konventioneller Datenbankindex definiert.

6.2.2 Tupel als Knoten

Im Hinblick auf die Binärbaumdarstellung der linearen Ordnung der im cTree-Index gespeicherten Einträge werden die Begriffe „Indexeintrag“, „Tupel“ und „Knoten“ im Bezug auf den cTree-Index synonym verwendet.

6.2.3 Wurzel-Tupel t_{root} bzw. v_{root}

Ein besonderes Element (bzw. ein besonderer Knoten) in der Extension eines cTree-Indexes $ix_{r.a'}^{cTree}$ ist dasjenige, das die Wurzel der Baumdarstellung konstituiert. Es wird im Folgenden t_{root} (bzw. v_{root}) genannt. Weiterhin werden im Folgenden die Notationen für die Projektionsoperation $\pi_a(t_x)$ und $v_x.a$ synonym verwendet. t_{root} ist für das DBMS leicht auffindbar,

da es als einziges Tupel in $X\left(ix_{r.a'}^{cTree}\right)$ in $parent$ den Wert $NULL$ hat. Da auf $parent$ ein konventioneller Datenbankindex definiert ist, kann t_{root} schnell gefunden werden.

6.2.4 Horizontale Partitionierung

Ein cTree-Index $ix_{r.a'}^{cTree}$, der für ein Attribut a' in Relation r mit Extension $X(a')$ erstellt wurde, resultiert in einer Menge von Tupeln. Diese werden in der in Kapitel 6.2.1 definierten Datenbankrelation gespeichert.

In dieser Relation können jedoch auch die Extensionen von mehreren cTree-Indexen $ix_{r_i.a_j'}^{cTree}$ gespeichert werden, ohne dass Inkonsistenzen auftreten. Die Relation muss lediglich um ein zusätzliches Attribut erweitert werden, das zur horizontalen Partitionierung der Extension verwendet werden kann. Ein solches Attribut könnte etwa ein Fremdschlüsselattribut $indexAdmin$ sein, das die gleichnamige Verwaltungsrelation referenziert und ebenfalls indiziert ist (vorzugsweise mit einem Hash-Index, da dort nur auf Gleichheit gesucht wird). In Suchanfragen auf $ix_{r.a'}^{cTree}$ muss dann stets ein konkreter Wert für $indexAdmin$ ins WHERE-Kriterium mit aufgenommen werden.

6.3 Indexbenutzung und –pflege anhand eines Beispiels

In den folgenden Unterkapiteln wird detailliert auf die Funktionsweise des cTree-Index in seinen verschiedenen Benutzungsszenarios eingegangen. Auch wenn der cTree-Index in seiner Grundform die Ungleichheitsrelationen „<“ und „>“ abbildet, wird in diesem Unterkapitel ein in einer praktischen Implementierung naheliegendes Anwendungsbeispiel gewählt, das eine beschleunigte Präfixsuche auf einem lexikografisch geordneten Attribut implementiert. Die Präfixsuche lässt sich leicht in eine Intervallsuche umformulieren, so dass im folgenden eigentlich eine Suche auf einem halboffenen Intervall behandelt wird.

Sei $X\left(\pi_{a'}(r)\right)$ die Extension des verschlüsselten Beispielattributs von a'. Sie enthalte verschlüsselte, lexikografisch geordnete Zeichenkettenwertinstanzen. Sei weiterhin eine Normalisierungsfunktion f definiert, die auf den Klartext von Instanzen aus $X\left(\pi_{a'}(r)\right)$ angewendet werden kann; f implementiere beispielsweise die in Kapitel 5.1 genannte Collation DIN 5007-1. Sei darüber hinaus auf a' ein cTree-Index $ix_{r.a'}^{cTree}$ definiert, der beschleunigte Präfixsuchen entsprechend Kapitel 6.8 ermögliche und in einer gleichnamigen Datenbankrelation gespeichert sei. Eine M:N-Relation $mn_{r.a'}^{cTree}$ verbinde $ix_{r.a'}^{cTree}$ und r.

6.4 INSERT

Es werde ein neues Tupel t in r eingefügt. Sei $s = Dec\left(\pi_{a'}(t)\right)$[63] der entschlüsselte Attributwert für a' in t. Durch die Einfügung von t muss sowohl die Indexrelation $ix_{r.a'}^{cTree}$ als auch M:N-Relation $mn_{r.a'}^{cTree}$ aktualisiert werden. Hierzu sind mehrere Schritte erforderlich, deren Ausführung zwischen Client und Server aufgeteilt ist, was in der folgenden Beschreibung entsprechend gekennzeichnet ist.

6.4.1 Normalisierung und Verschlüsselung

Zunächst sind einige vorbereitende Maßnahmen erforderlich:

[63] An dieser Stelle ist es unerheblich, ob definite oder randomisierte Verschlüsselung verwendet wird.

1. **(*Client*)** Normalisiere s zu $s_f = f(s)$.[64]
2. ***(Client)*** verschlüssele s_f zu $s'_f = Enc(f(s))$.

6.4.2 Lookup

Anschließend ist zu prüfen, ob in $X(ix_{r.a'}^{cTree})$ bereits ein Tupel t_{s_f} existiert, dessen entschlüsselter Wert von $ixData$ dem Suchwert s bzw. seiner Normalisierung s_f gleicht:

$$\exists t_{f_s} \in X(ix_{r.a'}^{cTree}) : Dec\left(\pi_{ixData}\left(t_{s_f}\right)\right) = f(s), \qquad \textbf{(F32)}$$

oder ob t_{s_f} dem Index neu hinzugefügt werden muss. In ersterem Fall muss kein neues Indextupel hinzugefügt werden. Stattdessen wird lediglich t_{s_f} mit t über ein neu eingefügtes Tupel in $mn_{r.a'}^{cTree}$ verbunden.

Falls t_{s_f} nicht in $X(ix_{r.a'}^{cTree})$ gefunden werden kann, muss es dort eingefügt werden. Dazu ist neben dem Attributwert $Enc(s_f) = s'_f$ in den beiden Darstellungen der linearen Ordnung von $ix_{r.a'}^{cTree}$ die jeweils korrekte, dem Wert s entsprechende Einfügeposition zu ermitteln. Sie drückt sich für die Binärbaumdarstellung in der Belegung der Attribute $leftChild$, $rightChild$ und $parent$ aus und für die lineare Darstellung im Attributwert für $sortNumber$.

Die Positionsermittlung in der Binärbaumdarstellung T wird im Folgenden *Lookup*-Schritt genannt. Er besteht im Wesentlichen aus der Traversierung von T, für die der Client iterativ Indexelemente (*Knoten*) bzw. ganze Teilbäume vom Server anfordert, entschlüsselt und verarbeitet. Der Algorithmus ist im folgenden Pseudocode-Fragment beschrieben, wobei gekennzeichnet ist, welcher Teilschritt auf welcher Seite der Client-Server-Architektur ausgeführt wird.

1. ***(Client)*** Setze die Wurzel des gegenwärtig verarbeiteten Teilbaums $v_{currentRoot}$ auf die Wurzel von T: $v_{currentRoot} = v_{root}$.
2. ***(Client)*** Frage den Teilbaum $\tau_{currentRoot}$ mit Wurzel $v_{currentRoot}$ vom Server an.
3. ***(Server)*** Ermittle $\tau_{currentRoot}$ mit Höhe h und sende ihn an den Client.
4. ***(Client)*** Entschlüssele den empfangenen Teilbaum $\tau_{currentRoot}$.
5. ***(Client)*** Setze den gegenwärtig verarbeiteten Knoten $v_{current}$ auf die Wurzel des gegenwärtig vorhandenen Teilbaums: $v_{current} = v_{currentRoot}$. Traversiere anschließend $\tau_{currentRoot}$ entsprechend den folgenden Anweisungen:
6. ***(Client)*** Falls $Dec(\pi_{ixData}(v_{current})) = s_f$, gib $\pi_{seq}(v_{current})$ als Traversierungsergebnis zurück, zusammen mit der Information, dass $v_{current}$ ein *direkter Treffer* ist, d. h. es wurde in der cTree-Relation ein Tupel gefunden, das dem Suchwert gleicht. Terminiere den Lookup-Schritt.
7. ***(Client)*** Falls $Dec(\pi_{ixData}(v_{current})) > s_f$, setze $v_{current} = \pi_{leftChild}(v_{current})$
8. ***(Client)*** Falls $Dec(\pi_{ixData}(v_{current})) < s_f$, setze $v_{current} = \pi_{rightChild}(v_{current})$.
9. ***(Client)*** Falls der Knoten, auf den $v_{current}$ in Schritt 7 bzw. 8 gesetzt wurde, nicht in $\tau_{currentRoot}$ enthalten ist, setze $v_{currentRoot} = v_{current}$ und gehe zu Schritt 2.
10. ***(Client)*** Falls die Blattebene von T erreicht wurde und dementsprechend an der in Schritt 7 bzw. 8 ermittelten neuen Position für $v_{current}$ kein Knoten vorhanden ist,

[64] Falls keine Normalisierung von s notwendig sein sollte, interpretiere f als die Identitätsfunktion id.

enthält $v_{current}$ genau die Position, an der der neue Knoten in die Binärbaumdarstellung der linearen Ordnung der Elemente des cTree-Index eingefügt werden muss.

11. ***(Client)*** Gib $seq_{ins} = \pi_{seq}(v_{current})$ als Ergebnis des Lookup-Schritts zurück, zusammen mit der Information, dass s_f nicht in $X\left(ix_{r.a'}^{cTree}\right)$ gefunden wurde und im $ixData$-Attribut eines neuen Indextupels eingefügt werden muss.

In der obigen Beschreibung des Lookup-Schritts werden Teilbäume τ von T der Höhe h, die bis zu $2^h - 1$ Knoten enthalten können, vom Server angefordert, entschlüsselt und verarbeitet. In einer naiveren Form würden anstatt ganzer Teilbäume einfach einzelne Knoten (d. h. Teilbäume der Höhe 1) angefordert, entschlüsselt und mit dem Suchwert verglichen. Anhand des Ergebnisses des Vergleichs würde der Algorithmus eine Entscheidung treffen, welchen Knoten er als nächsten anfordert. Dies würde wiederholt werden, bis ein direkter Treffer gefunden oder der Algorithmus auf der Blattebene angekommen wäre.

Da jedoch absehbar ist, dass die Verzögerung des Anforderns und Empfangens von Daten vom Server einen erheblichen Teil der Gesamtlaufzeit des Algorithmus in Anspruch nimmt, wurde die obige Form des Algorithmus gewählt, da sie sich eignet, die Anzahl solcher *Database-Roundtrips* zu minimieren; sie stellt so eine der Möglichkeiten dar, die Performanz des cTree-Indexes zu erhöhen und wird in Kapitel 9.3 ausführlich behandelt.

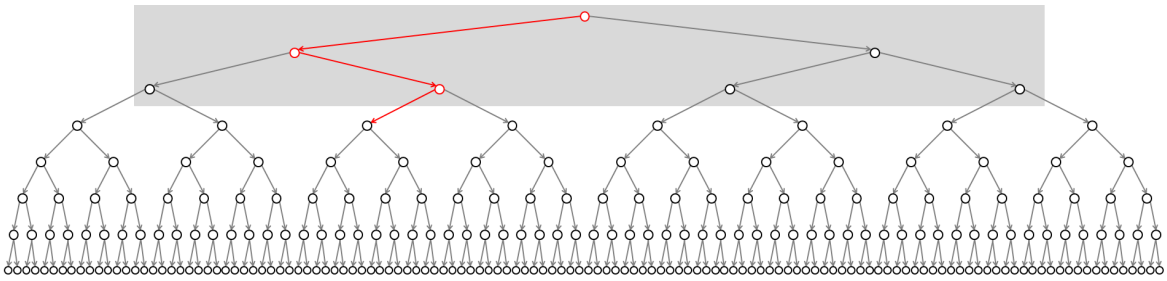

Abb. 3: Lookup-Operation auf dem cTree-Index, erste Teilbaum-Iteration.

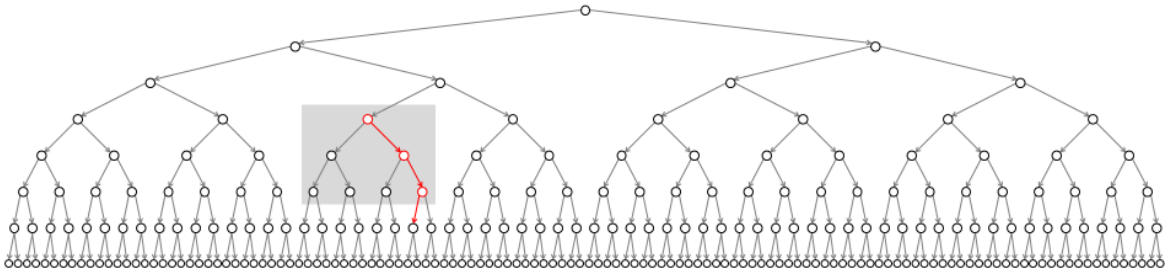

Abb. 4: Lookup-Operation auf dem cTree-Index, zweite Teilbaum-Iteration.

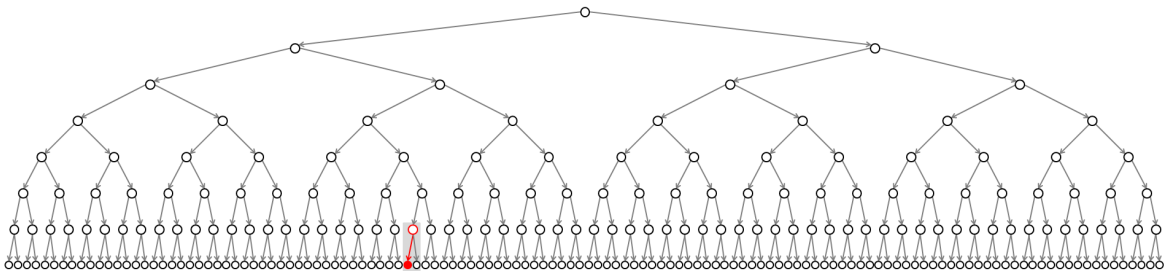

Abb. 5: Lookup-Operation auf dem cTree-Index, dritte und letzte Teilbaum-Iteration.

Abb. 3, Abb. 4 und Abb. 5 illustrieren den Lookup-Algorithmus mit einer Teilbaumhöhe $h = 3$. Das graue Rechteck repräsentiert dabei den jeweils gegenwärtig vom Server abgerufenen Teilbaum $\tau_{currentRoot}$. In dem hier vorliegenden Szenario tritt ein direkter Treffer auf der Blattebene des Baums auf.

6.4.3 Einfügeoperationen

Nach der Beschreibung des Lookup-Algorithmus kann nun die darauf folgende Einfügeoperation behandelt werden. Hier muss eine Fallunterscheidung bezüglich der Verarbeitung des Lookup-Ergebnisses vorgenommen werden, je nachdem, ob im Lookup-Schritt ein direkter Treffer stattgefunden hat oder nicht.

6.4.3.1 Einfügeoperation im Falle eines direkten Treffers

Im Falle eines direkten Treffers, also eines in $X\left(ix_{r.a'}^{cTree}\right)$ gefundenen Tupels t_{s_f} mit $Dec\left(\pi_{ixData}\left(t_{s_f}\right)\right) = s_f$:

- ***(Client)*** Gib die Anweisung an den Server aus, ein entsprechendes, t und t_{s_f} verbindendes Tupel $t_{t,t_{s_f}}^{mn}$ in $mn_{r.a'}^{cTree}$ einzufügen.
- ***(Server)*** Führe die Einfügeoperation aus.

6.4.3.2 Einfügeoperationen im Falle keines direkten Treffers

Falls kein Tupel t_{s_f} in $X(ix_{r.a'}^{cTree})$ gefunden wurde mit $Dec\left(\pi_{ixData}\left(t_{s_f}\right)\right) = s_f$, sondern nur der Knoten seq_{ins} als Einfügeposition in der Binärbaumdarstellung der linearen Ordnung:

- ***(Client)*** Gib dem Server die Anweisung, ein neues Tupel t_{s_f} in $ix_{r.a'}^{cTree}$ einzufügen mit $\pi_{refParent}\left(t_{s_f}\right) = seq_{ins}$, $\pi_{refLeft}\left(t_{s_f}\right) = \pi_{refRight}\left(t_{s_f}\right) = null$ und $\pi_{ixData}\left(t_{s_f}\right) = s_f'$.
- ***(Client)*** Gib dem Server die Anweisung, ein t und t_{s_f} verbindendes Tupel $t_{t,t_{s_f}}^{mn}$ in $mn_{r.a'}^{cTree}$ einzufügen.
- ***(Server)*** Führe beide vom Client angewiesenen Einfügeoperationen aus.

6.4.4 Weitere Operationen im Falle keines direkten Treffers

Darüber hinaus sind, falls t_{s_f} neu in $ix_{r.a'}^{cTree}$ eingefügt werden musste (siehe Kapitel 6.4.3.2), die folgenden Operationen ebenfalls erforderlich:

(Server) Setze das $leftChild$- bzw. $rightChild$-Attribut des Tupels $v_{current}$, welches nun der Vaterknoten von t_{s_f} ist, mit einer entsprechenden UPDATE-Operation, um die bidirektionale Konsistenz der Binärbaumdarstellung der linearen Ordnung zu wahren.

(Server) Entsprechend der Position von s_f in der linearen Ordnung der in $X\left(\pi_{ixData}\left(ix_{r.a'}^{cTree}\right)\right)$ verschlüsselt gespeicherten Werte muss die im $sortNumber$-Attribut (welches ganzzahlige Werte speichert) gehaltene Nachbildung dieser linearen Ordnung ebenfalls konsistent gehalten werden; $\pi_{sortNumber}\left(t_{s_f}\right)$ muss also korrekt gesetzt werden. Der Wert kann auf beliebige Art und Weise berechnet werden, so lange er außer der linearen Ordnung keine Information über die eigentlichen, in $X\left(\pi_{ixData}\left(ix_{r.a'}^{cTree}\right)\right)$ gespeicherten Werte exponiert. Die folgende Heuristik berechnet $sortNumber$-Werte auf einfache Weise:

- Ermittle den $sortNumber$-Attributwert des Elternknotens von t_{s_f}, $sn_{p_1} = \pi_{sortNumber}(v_{current})$
- Falls t_{s_f} das rechte Kind von $v_{current}$ ist, ermittle den zu sn_{p_1} nächstgrößeren in $X\left(ix_{r.a'}^{cTree}\right)$ enthaltenen Attributwert:

$$sn_{p_2} = \min\left(\sigma_{sortnumber>sn_{p_1}}\left(\pi_{sortNumber}\left(X\left(ix_{r.a'}^{cTree}\right)\right)\right)\right) \quad \textbf{(F33)}$$

- Falls t_{s_f} stattdessen das linke Kind von $v_{current}$ ist, ermittle den zu sn_{p_1} nächstkleineren in $X\left(ix_{r.a'}^{cTree}\right)$ enthaltenen Attributwert:

$$sn_{p_2} = \max\left(\sigma_{sortnumber<sn_{p_1}}\left(\pi_{sortNumber}\left(X\left(ix_{r.a'}^{cTree}\right)\right)\right)\right) \quad \textbf{(F34)}$$

- Wähle für $\pi_{sortNumber}\left(t_{s_f}\right)$ das arithmetische Mittel von sn_{p_1} und sn_{p_2}: $\pi_{sortNumber}\left(t_{s_f}\right) = \frac{sn_{p_1}+sn_{p_2}}{2}$ **(F35)**

6.5 Weitere Aspekte der INSERT-Operation

Neben der in Kapitel 6.4 beschriebenen Grundform der Einfügeoperation auf der cTree-Indexrelation kommen die folgenden weiteren Aspekte zur Geltung:

6.5.1 Neunummerierung von $\pi_{sortNumber}\left(X\left(ix_{r.a'}^{cTree}\right)\right)$

(***Server***) Vor dem letzten Schritt der INSERT-Operation, also dem Einfügen eines neuen Indextupels in $ix_{r.a'}^{cTree}$, kann bei Verwendung der in Kapitel 6.4.4 beschriebenen Heuristik zur Berechnung von $sortNumber$-Attributwerten eine ordnungserhaltende Neunummerierung der Elemente von $X\left(\pi_{sortNumber}\left(ix_{r.a'}^{cTree}\right)\right)$ nötig werden, und zwar dann, wenn der berechnete Wert $\pi_{sortNumber}\left(t_{s_f}\right)$ eine Kollision mit einem bereits existierenden $sortNumber$-Attributwert verursacht. Dies ist dann der Fall, wenn $\pi_{sortNumber}\left(t_{s_f}\right)$ zwischen zwei existierende $sortNumber$-Attributwerte eingefügt werden soll, die den Abstand 1 haben.

In diesem Fall muss die gesamte cTree-Indexstruktur neu nummeriert werden. Dies kann nach einer Heuristik geschehen die autonom vom Server ausgeführt wird. Eine solche Heuristik ist im folgenden Pseudocode-Fragment angegeben. Es eignet sich für die Nummerierung von cTree-Instanzen mit einer Kardinalität von bis zu $2^n - 1$ Elementen, $n \in \mathbb{N}$.

```
renumber(sq,sn,i){
  v := σ_seq=sq(ix_r.a');
  if(v == null) return;
  π_sortNumber(v) := sn;
  i := i - 1;
  sq_l := π_leftChild(v); sq_r := π_rightChild(v);
  sn_l := sn - 2^i; sn_r := sn + 2^i;
  if(i >= 0){
    renumber(sq_l,sn_l,i);
    renumber(sq_r,sn_r,i);
  }
}

renumber(π_seq(v_root),0,n-1);
```

(Q13)

6.5.2 Rebalancierung der Binärbaum-Darstellung der linearen Ordnung

(***Server***) Wie zuvor erwähnt, wurde für die Implementierung der Binärbaumdarstellung der linearen Ordnung des cTree-Indexes die AVL-Baum-Datenstruktur gewählt. Sie muss nach Einfügeoperationen oft rebalanciert werden, um ihre Invariante wiederherzustellen: Für jeden Knoten im AVL-Baum muss gelten, dass sich die Höhe seines rechten und linken Teilbaums um maximal 1 unterscheidet.

Dabei wird so vorgegangen, dass der Rebalancierungsalgorithmus vom neu eingefügten Blatt aus startend in Richtung Wurzelknoten des Baumes navigiert und für jeden Knoten auf diesem Pfad prüft, ob die AVL-Invariante erfüllt ist. Ist dies der Fall, navigiert der Algorithmus zum nächsten Knoten; falls nicht, muss eine von vier verschiedenen sogenannten Rotationsoperationen ausgeführt werden

Die Rebalancierung der Binärbaumdarstellung im cTree-Index ist vergleichbar mit Reorganisationsoperationen von konventionellen DBMS-Indexstrukturen. Sie kann vom Server autonom ohne Unterstützung des Clients durchgeführt werden. Etwa können Ruhezeiten des Servers, beispielsweise nachts, geeignet dafür sein.

6.5.3 Direkte Treffer unter definiter / randomisierter Verschlüsselung

In Kapitel 6.1.1 wurde angegeben, das Verfahren zur Verschlüsselung des Inhalts von $ixData$ sei beliebig wählbar; insbesondere sei die Frage, ob das Verfahren randomisiert oder definit verschlüsselnd sei, frei wählbar. Beim AES-Verfahren sind diese Betriebsarten beispielsweise durch einen zufällig gewählten bzw. einen konstanten Initialisierungsvektor realisierbar (siehe Kapitel 3.3.1 und 3.3.2).

Für die Prüfung, ob für einen gegebenen Klartextwert ein entsprechendes Tupel bereits im Index enthalten ist (damit handelt es sich um eine Gleichheitssuche auf der cTree-Indexrelation), hat die Wahl dieses Betriebsmodus jedoch weit reichende Konsequenzen:

6.5.3.1 Direkte Treffer unter definiter Verschlüsselung

Bei der Verwendung definiter Verschlüsselung gilt: Wenn im Index nach einem Wert s gesucht wird (etwa bei der Einfügeoperation (siehe Kapitel 6.4) oder der Suchoperation (siehe Kapitel 6.8)), reicht es, s zu normalisieren und zum binären Wert $s_f' = Enc_{det}(f(s))$ zu verschlüsseln. Anschließend kann auf $\pi_{ixData}\left(X\left(ix_{r.a'}^{cTree}\right)\right)$ aufgrund der Äquivalenz

$$a = b \Leftrightarrow Enc_{det}(a) = Enc_{det}(b), \text{ bzw.} \quad \textbf{(F36)}$$

$$a = b \Leftrightarrow Enc_{det}\big(f(a)\big) = Enc_{det}\big(f(b)\big) \quad \textbf{(F37)}$$

direkt nach dem konkret angegebenen Wert s_f' auf $X\left(\pi_{ixData}\left(ix_{r.a'}^{cTree}\right)\right)$ gesucht werden, entsprechend der schematischen SQL-Anfrage:

```
SELECT seq FROM ix_r.a' WHERE ixData = s_f'
```
(Q14)

Die Suche auf dem binären Attribut $ixData$ kann durch einen konventionellen DBMS-Index signifikant beschleunigt werden.

6.5.3.2 Direkte Treffer unter randomisierter Verschlüsselung

Bei Verwendung randomisierter Verschlüsselung gelten die in (F36) und (F37) ausgedrückten Äquivalenzen nicht, so dass die erforderliche Gleichheitssuche über iterierte Knoten-

bzw. Teilbaumanfragen durchgeführt werden muss wie in Kapitel 6.4 beschrieben. Diese ist erheblich langsamer als die in Kapitel 6.5.3.1 beschriebene Vorgehensweise, die die Eigenschaften definiter Verschlüsselung und konventioneller Datenbankindexe ausnutzt. Letztere sollte daher bevorzugt werden, wenn sich die Möglichkeit dazu bietet.

Dies gilt insbesondere dann, wenn applikationsbezogen absehbar ist, dass in eine Datenrelation viele Einfügeoperationen von gleichen Werten stattfinden: Wenn für einen eingefügten Wert eine hohe Wahrscheinlichkeit besteht, dass sein zugehöriger Indexwert bereits in der cTree-Indexrelation enthalten ist, kann die Performanz des cTree-Index von dem stark beschleunigten Prüfschritt auf einen direkten Treffer entsprechend Kapitel 6.5.3.1 profitieren. Sollte stattdessen ein Szenario vorherrschen, in dem ausschließlich bzw. vorwiegend eindeutige Werte in die Indexrelation eingefügt werden, ist dieser vorangestellte Schritt der Prüfung auf einen direkten Treffer überflüssig, da er stets bzw. meist kein Ergebnis liefert und die darüber hinaus gesamte Einfügeoperation durch einen zusätzlichen Client-Server-Roundtrip verlangsamt.

Weiterhin sollte in Betracht gezogen werden, dass die Verwendung definiter Verschlüsselung ein erhöhtes Risiko von Informationsverlust mit sich bringt. Kapitel 10.1.1 widmet sich dieser Problematik detaillierter.

6.6 UPDATE

Nach der ausführlichen Beschreibung der INSERT-Operation in den Kapiteln 6.4 und 6.5 sind UPDATE- und DELETE-Operation leicht beschreibbar.

Bei einem UPDATE eines Tupels t in einer cTree-indizierten Relation r sollte das zugehörige Indextupel $t_{ix_{alt}}$ nicht wie t einfach überschrieben werden. Zum einen können, ähnlich zu Indexen für die Gleichheitssuche, andere Tupel in r mit $t_{ix_{alt}}$ über $mn_{r.a'}^{cTree}$ verknüpft sein, welche dann mit einem falschen Indexwert assoziiert wären. Zudem ist es wahrscheinlich, dass $t_{ix_{alt}}$ nach dem Überschreiben an der falschen Stelle der Darstellungen der linearen Ordnung positioniert ist, und die Behebung dieser Inkonsistenz wäre sehr aufwändig.

Stattdessen sollte ein neues Indextupel $t_{ix_{neu}}$ für den aktualisierten Wert $\pi_{a'}(t)$ in die Indexrelation eingefügt, bzw. ein bereits in $X\left(ix_{r.a'}^{cTree}\right)$ existierendes Indextupel $t_{ix_{neu}}$ ermittelt werden, wie in Kapitel 6.4 beschrieben. Weiterhin muss das Tupel t_{mn} in $mn_{r.a'}^{cTree}$, das bisher t und $t_{ix_{alt}}$ referenzierte, aktualisiert werden, so dass es nun t und $t_{ix_{neu}}$ referenziert.

Schließlich kann $t_{ix_{alt}}$ aus $ix_{r.a'}^{cTree}$ gelöscht werden, falls dies applikationsbezogen gewünscht ist und es nicht von anderen Tupeln in $mn_{r.a'}^{cTree}$ referenziert wird. Unter Umständen werden dadurch analog zur Einfügeoperation Schritte zur Indexreorganisation[65] notwendig, die jedoch wiederum autonom auf dem Server ausgeführt werden können. Ein Verzicht auf die Löschung von $t_{ix_{alt}}$ kann dagegen den Vorteil haben, dass das Tupel nicht neu in den Index eingefügt werden muss, falls ein entsprechender Wert in r zu indexieren ist.

[65] Hiermit sind insbesondere etwaige Rebalancierungsoperationen auf der binären Baumdarstellung der linearen Ordnung gemeint.

6.7 DELETE

Wird ein Tupel t aus r gelöscht, muss das t referenzierende M:N-Tupel t_{mn} ebenfalls aus $mn_{r.a'}^{cTree}$ gelöscht werden. Das von t_{mn} referenzierte Indextupel t_{ix} kann ebenfalls aus $ix_{r.a'}^{cTree}$ gelöscht werden, falls dies applikationsbezogen gewünscht ist und es in $mn_{r.a'}^{cTree}$ keine anderen Tupel gibt, die t_{ix} referenzieren. Dieser Schritt, wie auch die gegebenenfalls notwendigen Maßnahmen zur Indexreorganisation[66], können vom Server autonom ausgeführt werden.

6.8 RANGE SELECT

Bei $SELECT$, der letzten der vier Basisoperationen auf dem cTree-Index, liegt der Fokus in dieser Dissertation auf der Bereichs- oder Intervallsuche ($RANGE\ SELECT$). Im Folgenden spezifiziere eine Datenbankanfrage Q_I auf einem verschlüsselten, cTree-indizierten Attribut a' einer Datenrelation r eine Intervallsuche nach allen Tupeln t, für die $Dec(\pi_{a'}(t))$ in einem definierten Intervall I liege. I sei ein offenes, geschlossenes oder halboffenes Intervall mit den Parametern p_1 und p_2 als Intervallgrenzen.

Unter Zuhilfenahme des cTree-Indexes sollen nun beschleunigte Datenbankanfragen auf den verschlüsselten Daten in $X(\pi_{a'}(r))$ konstruiert werden, die das Äquivalent zu einer der folgenden SQL-Anfragen auf einem Klartextattribut a darstellen:

```
a ∈]−∞, p1[:  SELECT (…) WHERE a < p1
a ∈]−∞, p1]:  SELECT (…) WHERE a <= p1
a ∈]p1, ∞[:   SELECT (…) WHERE a > p1
a ∈ [p1, ∞[:  SELECT (…) WHERE a >= p1
a ∈]p1, p2[:  SELECT (…) WHERE a > p1 AND a < p2
a ∈]p1, p2]:  SELECT (…) WHERE a > p1 AND a <= p2
a ∈ [p1, p2[:  SELECT (…) WHERE a >= p1 AND a < p2
a ∈ [p1, p2]:  SELECT (…) WHERE a >= p1 AND a <= p2
```
(Q15)

6.8.1 Intervallsuchen auf lexikografisch geordneten Daten

Auf Klartextdaten, die einer linearen Ordnung unterliegen, können Intervallsuchen ausgeführt werden, und konventionelle Datenbankindexe beschleunigen diese Intervallsuchen signifikant. Dies betrifft etwa Anfragen auf numerischen Daten. Jedoch lassen sich auch auf Daten, die einer anderen Form einer linearen Ordnung unterliegen, ebenfalls Intervallsuchen durchführen:

6.8.1.1 Lexikografische Ordnung

Eine *lexikografische Ordnung* ist eine Methode, um eine lineare Ordnung auf Sammlungen von ihrerseits linear geordneten Elementen zu definieren. Ein Beispiel dafür ist der „Medaillenspiegel“ bei olympischen Spielen, eine Rangliste der bei den Spielen teilnehmenden Nationen, bei der für die Platzierung einer Nation zuerst die Anzahl der gewonnenen Goldmedaillen relevant ist, anschließend die Anzahl der gewonnenen Silber- und zuletzt die Anzahl der gewonnenen Bronzemedaillen. Ein weiteres Beispiel ist die Tabelle der Fußball-Bundesliga, bei der am wichtigsten für die Platzierung einer Mannschaft die Anzahl der erspielten Punkte ist, anschließend die Differenz zwischen selbst erzielten und Gegentoren und an dritter Stelle die Anzahl der selbst erzielten Tore.

[66] Siehe Kapitel 6.6

Das naheliegendste Beispiel jedoch, das gleichzeitig eine wichtige Rolle bei der praktischen Anwendung des cTree-Indexes spielt, ist eine lexikografische Ordnung für Wörter $w \in \Sigma^*$, die aus beliebig vielen Symbolen eines n-stelligen Alphabets

$$\Sigma = \{\sigma_1, \dots, \sigma_n\} \tag{F38}$$

zusammengesetzt sind. Dabei sei über den Symbolen $\sigma_i \in \Sigma$ eine lineare Ordnung definiert, die durch die binäre Relation „$<_\sigma$“$\subseteq \Sigma \times \Sigma$ ausgedrückt sei:

$$\forall i, j \in \{1, \dots, n\}, i \neq j: i < j \Leftrightarrow \sigma_i <_\sigma \sigma_j \tag{F39}$$

Dann ist für Wörter $w^* \in \Sigma^*$, für die gilt:

$$w \in \Sigma^*, w = \sigma_{i_1} \dots \sigma_{i_m}, m \in \mathbb{N}, \forall j \in \{1, \dots, m\}: i_j \in \{1, \dots, n\} \tag{F40}$$

die lexikografische Ordnung als binäre Relation „$<$“ $\subseteq \Sigma^* \times \Sigma^*$ folgendermaßen definiert:

$$\forall v, w \in \Sigma^*, v = \sigma_{k_1} \dots \sigma_{k_r}, w = \sigma_{l_1} \dots \sigma_{l_s}:$$

$$v < w \Leftrightarrow \left(\exists\, c > 0: \left(\forall j < c: \sigma_{k_j} = \sigma_{l_j}\right) \wedge \left(\sigma_{k_c} <_\sigma \sigma_{l_c}\right)\right) \vee$$

$$(r < s \wedge \forall j \in \{1, \dots, r\}: \sigma_{k_j} = \sigma_{l_i}) \tag{F41}$$

6.8.1.2 Die Inkrementierungsfunktion *inc*

Sei weiterhin für nichtleere Wörter $w \in \Sigma^*$ mit $w = \sigma_{i_1} \dots \sigma_{i_m}$ eine Funktion inc zur Wortinkrementierung folgendermaßen definiert:

$$inc: \Sigma^* \to \Sigma^* \cup \{\infty\},$$

$$inc(w) := \begin{cases} \sigma_1, \text{falls } w = \varepsilon \\ \sigma_{i_1+1}, \text{falls } |w| = 1 \wedge i_1 < n \\ \infty, \text{falls} \forall j \in \{1, \dots, m\}: i_j = n \\ \sigma_{i_1} \dots \sigma_{i_{k-2}} \sigma_{i_{k-1}+1}, \text{falls } \exists k \in \{2, \dots, m\}: i_{k-1} \in \{1, \dots, n-1\} \wedge \\ \qquad\qquad \forall j \in \{k, \dots, m\}: i_j = n \end{cases} \tag{F42}$$

6.8.1.3 Umformulierung regulärer Ausdrücke in Intervallsuchen

Reguläre Ausdrücke $w\Sigma^*$, die Präfixsuchen auf lexikografischen Daten definieren, kommen oft in der praktischen Anwendung von Informationssystemen zum Einsatz, wie beispielsweise die Suche nach Personen in einem Stammdatenbestand, deren Nachname mit einer bestimmten Buchstabenkombination beginnt. Somit ist ein Index, der Präfixsuchen auf verschlüsselten Daten beschleunigt, von Vorteil für die praktische Anwendbarkeit eines Informationssystems, das auf verschlüsselten Daten operiert.

Hier lässt sich die Eigenschaft regulärer Ausdrücke $w\Sigma^*$ ausnutzen, dass sie sich unter Zuhilfenahme von inc leicht in Intervallsuchen umformulieren lassen: Es gilt, dass die Menge von Wörtern, die durch den regulären Ausdruck $w\Sigma^*$ definiert ist, genau die gleichen Elemente enthält wie das halboffene Intervall $[w, inc(w)[$:

$$w\Sigma^* = [w, inc(w)[\tag{F43}$$

Beispielsweise beschreibt die Fragestellung: *„Bei welchen unserer Kunden fängt der Nachname mit ‚schm' an?"*[67] einen gängigen Anfragetyp beim praktischen Einsatz eines Informationssystems und lässt sich in einer SQL-Anfrage auf dem lexikografisch geordneten Attribut $lastName$ einer Klartextrelation $customers$ folgendermaßen ausdrücken:

```
SELECT * FROM customers WHERE lower(lastName) LIKE 'schm%',        (Q16)
```

wobei ein konventioneller Datenbankindex auf $lastName$ die Performanz von Anfragen darauf beschleunigt.

Wenn man sich die lexikografische Ordnung auf $lastName$ und die in (F42) definierte inc-Funktion zunutze macht, kann der SQL-Term `'schm%'`, der den regulären Ausdruck $schm\Sigma^*$ beschreibt, in das Intervall $[schm, schn[$ transformiert werden. Also ist auch (Q16) in eine äquivalente Intervallsuchen-Anfrage umformulierbar, welche sich durch die folgende SQL-Anfrage ausdrücken lässt:

```
SELECT * FROM customers
WHERE lower(lastName) >= 'schm' AND lower(lastName) < 'schn'      (Q17)
```

Ein anderes Beispiel betrifft die fünfstellige, hierarchisch aufgebaute Postleitzahl (PLZ), die Postzustellbezirke in der Bundesrepublik Deutschland angibt. Obwohl sie aus numerischen Ziffern bestehen, unterliegen PLZ-Werte einer lexikografischen Ordnung. In Datenbanken werden sie meist als Zeichenkettenattribute gespeichert, damit führende „0"-Ziffern nicht weggelassen werden.

Eine typische Fragestellung in einem Informationssystem auf einem solchen, eine PLZ speichernden Attribut ist etwa: „Welche unserer Kunden wohnen im PLZ-Bereich 59?" Die entsprechende (wieder zunächst auf Klartextdaten operierende) SQL-Anfrage lässt sich analog zum obigen Beispiel in (Q16) von der Form

```
SELECT * FROM customers WHERE plz LIKE '59%'                      (Q18)
```

transformieren zu:

```
SELECT * FROM customers WHERE plz >= '59' AND plz < '6'           (Q19)
```

6.8.2 Prinzipielles Vorgehen

Sei Q_I eine $RANGE\ SELECT$-Anfrage mit Suchintervall I auf einem verschlüsselten Attribut a' einer Relation r, die durch einen in der in der Indexrelation $ix_{r.a'}^{cTree}$ gespeicherten cTree-Index beschleunigt werde. Dann ist die Vorgehensweise bei der Abarbeitung dieser Anfrage in drei Phasen aufgeteilt. Im Folgenden wird beschrieben, wie die beiden Darstellungen der linearen Ordnung auf den indizierten Daten, welche in Kapitel 6.2 vorgestellt wurden, in diesen Phasen mit ihren jeweiligen Vorteilen ausgenutzt werden.

1. Falls p_1 und p_2, die Intervallgrenzen des auszuwertenden Suchintervalls I, nicht explizit angegeben sind, sondern im Form eines regulären Ausdrucks $w\Sigma^*$ (der eine Präfixsuche auf lexikografisch geordneten Daten definiert), leite p_1 und p_2 aus letzterem her, bevor die Auswertung von Q_I beginnen kann. Dies geschieht entsprechend Kapitel 6.8.1.3 wie folgt:

[67] Von Groß- und Kleinschreibung sei in diesem Beispiel abstrahiert.

$p_1 := w$ und **(F44)**

$p_2 := inc(w)$, **(F45)**

so dass entsprechend Kapitel 6.8.1.3 gilt:

$I = [p_1, p_2[= [w, inc(w)[$ **(F46)**

2. Ermittele für p_1 und p_2 die cTree-Indextupel t_{p_1} bzw. t_{p_2}, deren jeweiliger entschlüsselter $ixData$-Attributwert $Dec\left(\pi_{ixData}(t_{p_i})\right)$ das Infimum bzw. Supremum der Ergebnismenge der Intervallsuche bildet. Nutze für eine beschleunigte Ermittlung von t_{p_i} die Binärbaumdarstellung der linearen Ordnung der Indexdaten. Ergebnis beider t_{p_i}-Ermittlungsoperationen sei der jeweilige $sortNumber$-Attributwert von t_{p_i}:

   $sn_{p_i} = \pi_{sortNumber}(t_{p_i})$. **(F47)**

   Erweitere das Ergebnis dieses Schrittes um die boolesche Information $direct_{p_2}$, die besage, ob t_{p_2} ein exakter Treffer von p_2 ist oder nur ein Näherungswert. Dies hat Auswirkungen auf die finale SELECT-Operation (siehe Kapitel 6.8.3.4): Da I ein nach oben halboffenes Intervall ist, wird, falls t_{p_2} kein exakter Treffer von p_2 ist, das Tupel in die Ergebnismenge des finalen SELECTs aufgenommen, anderenfalls nicht. Der in t_{p_1} gefundene Wert wird in jedem Fall in die Ergebnismenge eingeschlossen, so dass das Gesamtergebnis von Schritt 2 lautet:

   $(sn_{p_1}, sn_{p_2}, direct_{p_2})$ **(F48)**

3. Nutze die beiden sn_{p_i}-Werte zusammen mit $direct_{p_2}$ als Parameter für eine SQL-Anfrage auf $ix_{r.a'}^{cTree}$. Diese Anfrage macht sich die strikt lineare Darstellung der linearen Ordnung der Indexdaten und den darauf definierten konventionellen Datenbankindexen zunutze: Zum einen wird sie zum beschleunigten Auffinden aller Elemente der Ergebnismenge genutzt und zum anderen zu deren schneller, serverseitiger Sortierung. Im Folgenden sei die SQL-Anfrage beispielhaft dargestellt:

```
SELECT * FROM ix_r.a'
WHERE sortNumber >= sn_p1 AND
((NOT direct_p2 AND sortNumber <= sn_p2) OR
(direct_p2 AND sortNumber < sn_p2))
```
(Q20)

6.8.3 Detailliertes Vorgehen

Die in Kapitel 6.8.2 skizzierte RANGE SELECT-Operation wird im Folgenden anhand eines Beispiels detaillierter beschrieben. Es wird eine Relation $customers$ mit einem verschlüsselten, cTree-indizierten Attribut $lastName'$ betrachtet. Auf $customers$ wird eine Anfrage nach allen Tupeln abgesetzt, bei denen der entschlüsselte, auf Kleinbuchstaben normierte Inhalt von $lastName'$ mit der Zeichenkette $w = "schm"$ beginnt.

Entsprechend dem in Kapitel 6.8.2 beschriebenen Schritt 1 wird der reguläre Ausdruck $w\Sigma^*$ in zwei Intervallgrenzparameter $p_1 = "schm"$ mit dem Ungleichheitsoperator " $\geq$ " und $p_2 = inc("schm") = "schn"$ mit dem Ungleichheitsoperator " $<$ " transformiert, woraus sich das Suchintervall $I = [schm, schn[$ ergibt. Weiterhin soll die Ergebnismenge der Anfrage sortiert nach dem entschlüsselten Inhalt von $lastName$ vom Server zurückgegeben werden. Dies entspricht auf einem Klartext-Attribut $lastName$ der folgenden SQL-Anfrage:

```
SELECT * FROM customers
WHERE lastName >= p1 AND lastName < p2
ORDER BY lastName[68]
```
(Q21)

Im Folgenden wird der in Kapitel 6.8.2 skizzierte Schritt 2 detailliert beschrieben. Analog zur in Kapitel 6.4 behandelten Einfügeoperation wird bei der Beschreibung der Teilschritte Wert auf die Arbeitsaufteilung zwischen Client und Server gelegt, was im Text entsprechend kenntlich gemacht wird.

6.8.3.1 Vorbehandlung

(Client) Als Collation sei DIN 5007-2 definiert (siehe Kapitel 5.1, 5.5.1). Um diese nachzubilden, muss eine entsprechende Normalisierungsfunktion f auf die Intervallgrenzparameter p_1 und p_2 angewendet werden:

1. ***(Client)*** Normalisiere p_1 zu $f(p_1) = schm$.
2. ***(Client)*** Normalisiere p_2 zu $f(p_2) = schn$.

In diesem Fall verändert die Anwendung von f den jeweiligen Eingabewert nicht.

6.8.3.2 Ermittlung des Tupels für die untere Intervallgrenze

Für diesen Schritt wird eine Variante des Lookup-Schritts aus Kapitel 6.4 angewendet: Dabei wird nach dem besten Kandidaten für die untere Intervallgrenze in $X\left(ix^{cTree}_{customers.lastName'}\right)$ gesucht, d. h. nach demjenigen Tupel t_{p_1}, das den kleinsten Wert für $Dec\left(\pi_{ixData}\left(t_{p_1}\right)\right)$ enthält, der größer oder gleich „schm“ ist.

Der Rückgabewert dieses Schrittes zur Ermittlung einer Intervallgrenze sei $sn_{p_1} = \pi_{sortNumber}\left(t_{p_1}\right)$, sowie der boolesche Wert $direct_{p_1}$, der die Art der Ermittlung von t_{p_1} näher spezifiziere. Die Ausführung des Schritts umfasse die folgenden, in Pseudocode angegebenen Teilschritte:

1. **(Client)** Initialisiere den Wert des gegenwärtig besten Kandidaten für das Infimum von I mit $val_{p_1}^{current} = \infty$ und den $sortNumber$-Wert dieses Kandidaten ebenfalls mit $sn_{p_1}^{current} = \infty$[69].
2. ***(Client)*** Initialisiere die Wurzel $v_{currentRoot}$ des gegenwärtig verarbeiteten Teilbaums $\tau_{currentRoot}$, mit der Wurzel von T: $v_{currentRoot} = v_{root}$.
3. ***(Client)*** Sende eine Anfrage nach $\tau_{currentRoot}$ mit Wurzel $v_{currentRoot}$ an den Server.
4. ***(Server)*** Ermittle $\tau_{currentRoot}$ mit der definierten Höhe h und sende ihn an den Client.
5. ***(Client)*** Entschlüssele $\tau_{currentRoot}$.
6. ***(Client)*** Setze den gegenwärtig verarbeiteten Knoten $v_{current}$ auf die Wurzel von $\tau_{currentRoot}$: $v_{current} = v_{currentRoot}$. Traversiere $\tau_{currentRoot}$ anschließend entsprechend den folgenden Anweisungen:

---

[68] Bzw.: `SELECT * FROM customers WHERE lastName LIKE 'schm%' ORDER BY lastName.`

[69] Der jeweils verwendete symbolische Initialisierungswert ∞ diene zur Sicherstellung, dass der erste in $X\left(ix^{cTree}_{r.lastName'}\right)$ gefundene Wert, der größer ist als $val_{p_1}^{current}$, der erste Kandidat für $val_{p_1}^{current}$ werde. In der praktischen Implementierung kann man $val_{p_1}^{current}$ mit einem entsprechenden Sonderzeichen und $sn_{p_1}^{current}$ mit dem Maximalwert des Datentyps von $sortNumber$, im Falle eines 64-Bit-Integers etwa 9.223.372.036.854.775.807, initialisieren.

7. ***(Client)*** Falls $Dec\left(\pi_{ixData}(v_{current})\right) = f(p_1)$, terminiere mit dem Ergebnis $sn_{p_1} = \pi_{sortNumber}(v_{current})$ und $direct_{p_1} = true$. Letzteres zeige einen direkten Treffer an, $f(p_1)$ ist also in $X\left(Dec\left(\pi_{ixData}\left(ix_{r.a'}^{cTree}\right)\right)\right)$ enthalten.
8. ***(Client)*** Falls $Dec\left(\pi_{ixData}(v_{current})\right) > f(p_1)$ und $Dec\left(\pi_{ixData}(v_{current})\right) < val_{p_1}^{current}$, setze $val_{p_1}^{current} = Dec\left(\pi_{ixData}(v_{current})\right)$ und $sn_{p_1}^{current} = \pi_{sortNumber}(v_{current})$.
9. ***(Client)*** Falls gilt: $Dec\left(\pi_{ixData}(v_{current})\right) > f(p_1)$, setze den Nachfolgerknoten von $v_{current}$ wie folgt: $v_{current} = \pi_{leftChild}(v_{current})$;
   Falls stattdessen gilt: $Dec\left(\pi_{ixData}(v_{current})\right) < f(p_1)$, setze den Nachfolgerknoten von $v_{current}$ wie folgt: $v_{current} = \pi_{rightChild}(v_{current})$.
10. ***(Client)*** Falls der in Schritt 9 gesetzte Knoten $v_{current}$ nicht in $\tau_{current}$ enthalten ist, setze $v_{currentRoot} = v_{current}$. Gehe anschließend zu Schritt 3.
11. ***(Client)*** Falls die Blattebene von T erreicht ist, verlasse den modifizierten Lookup-Schritt und gib $sn_{p_1} = sn_{p_1}^{current}$ als Ergebnis zurück, zusammen mit der Information $direct_{p_1} = false$, die besage, dass das Ergebnis nicht durch einen direkten Treffer, sondern durch Approximation gefunden wurde.

6.8.3.3 Ermittlung der oberen Intervallgrenze

Führe einen zur in Kapitel 6.8.3.2 angegebenen Vorgehensweise analogen Schritt zur Ermittlung der oberen Intervallgrenze durch, der als Ergebnis $sn_{p_2} = \pi_{sortNumber}(t_{p_2})$ den $sortNumber$-Attributwert von demjenigen Tupel t_{p_2} ermittele, das in $ixData$ den größten Wert in $X\left(Dec\left(\pi_{ixData}\left(ix_{customers.lastName'}^{cTree}\right)\right)\right)$ enthalte, für den gelte:

$$Dec\left(\pi_{ixData}(t_{p_2})\right) < f(p_2) \qquad \textbf{(F49)}$$

Ermittle dabei analog zu Kapitel 6.8.3.2 ob bei der Ermittlung von sn_{p_2} ein direkter Treffer stattgefunden hat oder nicht. Speichere diese Angabe im booleschen Wert $direct_{p_2}$.

6.8.3.4 Durchführung der finalen SELECT-Operation

Diese Operation korrespondiert zum in Kapitel 6.8.2 skizzierten Schritt 3 der grundlegenden Vorgehensweise bei der RANGE SELECT-Operation. Hier wird anstelle der Binärbaumdarstellung der linearen Ordnung der Indexdaten deren lineare, im Attribut $sortNumber$ gespeicherte Darstellung verwendet.

Nachdem sn_{p_1} und sn_{p_2} vorliegen, wird die in (Q22) angegebene SQL-Anfrage beim Server abgesetzt. Sie gibt die gewünschte Ergebnismenge von Tupeln aus $customers$ zurück, sortiert nach deren $sortNumber$-Attribut, was einer Sortierung nach $lastName$ in unverschlüsselter Form entspricht. Hier wird deutlich, dass das $sortNumber$-Attribut in $ix_{customers.lastName'}^{cTree}$ sowohl zum schnellen Auffinden der Elemente der Ergebnismenge genutzt werden kann als auch für deren Sortierung:

```
SELECT c.*
FROM customers AS c
  JOIN mn_Customers.LastName' AS mn ON mn.refData = c.seq
  JOIN ix_Customers.LastName' as ix ON mn.refIndex = ix.seq
WHERE ix.sortNumber >= sn_p1
  AND ((direct_p2 AND ix.sortNumber < sn_p2)
  OR (NOT direct_p2 AND ix.sortNumber <= sn_p2))
ORDER BY ix.sortNumber
```
(Q22)

Weiterhin zeigt sich anhand von (Q22), dass der Einsatz des cTree-Index von den Stärken konventioneller Datenbankindexe profitiert. In (Q22) sind solche Datenbankindexe auf den Attributen $customers.seq$, $mn_{customers.lastName'}.refIndex$, $mn_{customers.lastName'}.refData$, $ix_{customers.lastName'}.seq$ und insbesondere $ix_{customers.lastName'}.sortNumber$ definiert, was zu einer signifikant beschleunigten Ausführung der Anfrage führt. Diese und weitere Möglichkeiten zur Steigerung der Performanz des cTree-Index werden in Kapitel 9 ausführlich behandelt.

Es sei betont, dass die (Q22) keine „Ergebniskandidaten" zurückgibt, die vom Client gefiltert werden müssen, wie dies etwa in [Hacigümüş2002-2] der Fall ist (siehe Kapitel 2). Stattdessen liefert (Q22) die exakte Ergebnismenge zurück.

6.8.4 Verwendung des cTree-Index zur Gleichheitssuche

Zum Abschluss des Kapitels wird ein weiterer Aspekt des cTree-Indexes beleuchtet: Unter einer leicht abgewandelten Verwendung kann damit auch eine Gleichheitssuche implementiert werden; ähnlich dem B- bzw. B⁺-Baum-Index auf Klartextdaten bedient also auch der cTree-Index unterschiedliche Suchtypen. Dies ist sogar auf zwei verschiedenen Wegen möglich, die in diesem Unterkapitel kurz behandelt werden.

6.8.4.1 Spezielle Parametrisierung der Präfixsuche

Eine naheliegende Art und Weise, die Gleichheitssuche nach einem Suchwert s auf einem verschlüsselten Attribut a' einer Relation r unter Verwendung des cTree-Indexes zu implementieren, ist, sie als Spezialfall der Intervallsuche umzusetzen. Hierbei wird s zunächst zu $s_f = f(s)$ normalisiert und das Suchintervall I als geschlossenes Intervall $I = [s_f, s_f]$ angesehen. Alle weiteren Schritte laufen analog zur in Kapitel 6.8.3 angegebenen Beschreibung der Intervallsuche.

Diese Implementierung hat zwar gegenüber der in Kapitel 6.8.4.2 angegebenen Implementierung einen erheblichen Performanznachteil, kann aber in einer weiter modifizierten Variante einen höheren Schutz der Vertraulichkeit der indexierten Daten bewirken. Dies wird im Kapitel 10 ausführlich behandelt.

6.8.4.2 Definit verschlüsseltes $ixData$

Eine weitere Art, die Gleichheitssuche auf dem cTree-Index zu implementieren, wurde bereits in Kapitel 6.5.3.1 angesprochen: Wenn $ixData$ definit verschlüsselt und ein konventioneller Datenbankindex darauf definiert ist, kann die Gleichheitssuche auf genau diesem Attribut ausgeführt werden. $ixData$ erfüllt in $ix_{r.a'}^{cTree}$ dann die gleiche Funktionalität, wie es $ix_{r.a'}^{equality}$ in Kapitel 5.7.2.4 tut und beschleunigt die Gleichheitssuche auch im gleichen Maße.

7 Shrinking Window

Im vorangegangenen Kapitel wurde die cTree-Index-Datenstruktur detailliert behandelt, mit der sich Intervallsuchen auf verschlüsselten Daten durchführen lassen, deren Klartexte einer linearen Ordnung unterliegen. Eine besondere Anwendung des cTree-Index, die dadurch möglich wird, sind Präfixsuchen auf lexikografisch geordneten Daten, also Suchen nach solchen Tupelmengen, die durch reguläre Ausdrücke über ein Alphabet Σ in der Form

$$w\Sigma^*, w \in \Sigma^* \quad \textbf{(F50)}$$

ausgedrückt werden können. In diesem Kapitel wird eine Modifikation von Struktur und Benutzung des cTree-Index vorgestellt, die im Folgenden als *Shrinking Window*-Ansatz bezeichnet wird. Damit lässt sich die Menge der über einen cTree-Index auf verschlüsselten Daten beschleunigt auswertbaren regulären Ausdrücke auf Ausdrücke der Form

$$\Sigma^* w \Sigma^* \quad \textbf{(F51)}$$

erweitern. Somit werden neben Präfixsuchen und Suchen auf Gleichheit (siehe auch Kapitel 6.8.4) auch Infix- und Postfixsuchen auf verschlüsselten Daten möglich.

7.1 Erweiterung der Index-Datenstrukturen

Sei wieder r eine Datenrelation, die ein verschlüsseltes Attribut a' enthält, dessen Klartextextension $X\left(Dec\left(\pi_{a'}(r)\right)\right)$ Werte enthält, die einer lexikografischen Ordnung unterliegen. Sei a' darüber hinaus mit einem in der Relation $ix_{r.a'}^{cTree}$ gespeicherten cTree-Index indiziert und r über die M:N-Relation $mn_{r.a'}^{cTree}$ mit $ix_{r.a'}^{cTree}$ verknüpft.

Dann lässt sich die oben erwähnte Erweiterung der möglichen Suchtypen auf den verschlüsselten Werten der Extension von a' durch zwei Änderungen an Aufbau und Benutzung des cTree-Index erreichen, die in den folgenden Unterkapiteln detaillierter beschrieben werden:

1. Der Speicherung von Suffixen der indexierten Klartextwerte in $ix_{r.a'}^{cTree}$ und
2. der zusätzlichen Speicherung der respektiven Länge des jeweils in $ix_{r.a'}^{cTree}$ referenzierten verschlüsselten Klartextwertes in $mn_{r.a'}^{cTree}$.

7.1.1 cTree mit Suffixen

Der Aufbau der cTree-Relation bleibt beim Shrinking Window-Ansatz so bestehen wie in Kapitel 6 beschrieben. Bei ihrer Verwendung wird dagegen eine grundlegende Änderung vorgenommen:

Für ein Tupel $t \in r$ mit $w = Dec\left(\pi_{a'}(t)\right)$ wird für jeden Suffix, der sich aus $f(w)$ bilden lässt, je ein Indextupel nach dem in Kapitel 6.4 beschriebenen Vorgehen in $ix_{r.a'}^{cTree}$ gespeichert. Es werden also wesentlich mehr Indextupel in $ix_{r.a'}^{cTree}$ gespeichert als in dem in Kapitel 6.4 beschriebenen Vorgehen, bei dem jedes Datentupel mit nur genau einem Indextupel verknüpft wurde, welches im $ixData$-Attribut $Enc(w)$ bzw. $Enc\left(f(w)\right)$ enthält.

Sei im Folgenden die Menge Σ^* über dem Alphabet $\Sigma = \{\sigma_1, \dots, \sigma_n\}$ die Domäne der Klartextversion von Attribut a. Die Ermittlung der Suffixe eines Wortes $w \in \Sigma^*$ mit $w = \sigma_{i_1} \dots \sigma_{i_n}$ geschehe durch sukzessives Weglassen des jeweils ersten Zeichens des vorliegenden (Teil-)

Worts, bis das verbliebene Wort keine Symbole mehr enthält. Für die Ermittlung aller Suffixe von w sei demnach die rekursive Funktion suf folgendermaßen definiert:

$$suf: \Sigma^* \rightarrow \wp(\Sigma^*);$$

$$suf(\varepsilon) \coloneqq \varepsilon$$

$$suf(\sigma_i) \coloneqq \sigma_i$$

$$suf(\sigma_{i_1}\sigma_{i_2} \dots \sigma_{i_n}) \coloneqq suf(\sigma_{i_2} \dots \sigma_{i_n}) \cup \{\sigma_{i_1}\sigma_{i_2} \dots \sigma_{i_n}\}, n > 1 \quad \textbf{(F52)}$$

Eine Präfixsuche, die auf der Menge aller Suffixe von Wörtern aus einer Wortmenge ausgeführt wird, generiert als Treffermenge genau diejenige Tupelmenge, die auch eine Infixsuche der Form $\Sigma^* w \Sigma^*$ generieren würde. Sie implementiert die Infixsuche also.

7.1.2 M:N-Relation

7.1.2.1 Zusätzliche Attribute

Mit der in Kapitel 7.1.1 beschriebenen Erweiterung des cTree-Indexes lassen sich bereits Infixsuchen der Form $\Sigma^* w \Sigma^*$ durchführen. Um diese Infixsuche darüber hinaus flexibel parametrisieren zu können, ist es erforderlich, das Schema der M:N-Relation $mn_{r.a'}^{cTree}$, die die Datenrelation r und die Indexrelation $ix_{r.a'}^{cTree}$ verknüpft, zu erweitern. Sei beispielhaft t_{suf} ein Indextupel, das einen Suffix suf beinhaltet. Sei weiterhin t_w ein Datentupel, das in a' das Wort w (verschlüsselt) enthält, welches suf als Suffix ab der dritten Stelle beinhaltet und $t_{suf,w}$ ein Tupel aus $mn_{r.a'}^{cTree}$, das t_{suf} und t_w verknüpft. Sei nun $mn_{r.a'}^{cTree}$ um die folgenden beiden Attribute erweitert:

- ein Klartextattribut $offset$ vom Typ $integer$, das angibt, wie viele Stellen in suf vom Ursprungswort w „abgeschnitten" wurden. Dieses Attribut wird für Infix- bzw. Postfixsuchen gebraucht, bei denen die Anzahl der Zeichen, die vor dem Suchstring vorkommen dürfen, explizit angegeben wird (siehe Kapitel 7.3.2 und 7.3.3).
- ein Klartextattribut $length$, ebenfalls vom Typ $integer$, das die Länge von w angibt. Dieses Attribut wird im Zusammenspiel mit $offset$ für Infixsuchen gebraucht, bei denen die Anzahl der Zeichen, die nach dem Suchstring vorkommen dürfen, explizit angegeben ist (siehe Kapitel 7.3.2 und 7.3.4).

Beide Attribute sollen im WHERE-Kriterium von SELECT-Anfragen zum Einsatz kommen. Deshalb ist es sinnvoll, sie jeweils zur Beschleunigung der Anfragen mit einem konventionellen DBMS-Index zu belegen, vorzugsweise mit einem auf B- bzw. B⁺-Baum-Index.

7.1.2.2 Anpassung der Primärschlüsseldefinition

Eine weitere Änderung am generischen Schemaaufbau der M:N-Relation folgt unmittelbar aus der in 7.1.1 beschriebenen veränderten Benutzung der Indexrelation und betrifft den Primärschlüssel der M:N-Relation. Bisher bestand dieser aus den beiden Fremdschlüsselattributen $refIndex$ und $refData$, die die zu verbindenden Relationen referenzieren. Dies war bisher sinnvoll, da diese beiden Attribute im Verbund zum einen Kandidatenschlüssel darstellten, also eine minimale, die Tupel eindeutig identifizierende Attributteilmenge. Zum anderen drückten sie die vorrangige inhärente Semantik der M:N-Relation aus, nämlich dass ein bestimmtes Tupel in der Indexrelation mit genau einem Tupel in der Datenrelation assoziiert ist.

Dies hat sich nun geändert, da ein Datentupel t_w in einem Shrinking Window-cTree-Index mit mehreren Indextupeln $t_{suf_1}, \ldots, t_{suf_n}$ verknüpft sein kann. Die Semantik der cTree-Indexrelation hat sich dahingehend geändert, dass sie nicht mehr singuläre kanonische Repräsentanten von Äquivalenzklassen (in verschlüsselter Form) beinhaltet, denen die einzelnen Attributwerteinstanzen w per Abbildungsfunktion f zugeordnet werden.

Stattdessen werden diese Repräsentanten in alle ihre möglichen Suffixe zerlegt, und jeder Suffix wird als eigenständiges Tupel in der cTree-Indexrelation als eigenständiger Indexeintrag gespeichert. Der Primärschlüssel der M:N-Relation muss also erweitert werden, um die semantische Repräsentation der Suffixwerte und den Kandidatenschlüsselstatus wiederzuerlangen. Dafür ist das neue Attribut $offset$ (siehe Kapitel 7.1.2.1) geeignet, so dass der Primärschlüssel der M:N-Relation auf $refIndex$, $refData$ und $offset$ besteht.

Das generische $CREATE\ TABLE$-Statement der M:N-Relation ändert sich damit von seinem in (Q1) angegebenen Basiszustand zu:

```
CREATE TABLE mn<relation>.<attribute>,<searchType>
(
  seq INTEGER NOT NULL UNIQUE,
  refData INTEGER NOT NULL REFERENCES <relation>(seq),
  refIndex INTEGER NOT NULL REFERENCES <indexRelation>(seq),
  offset INTEGER NOT NULL,
  length INTEGER NOT NULL,
  PRIMARY KEY(refIndex,refData,offset) CLUSTERED
)
CREAT INDEX <ixName> ON mn<relation>.<attribute>,<searchType>(refData)        (Q23)
```

7.1.3 Beispiel

Der veränderte Aufbau des cTree-Indexes unter dem Shrinking Window-Ansatz sei in diesem Unterkapitel an einem Beispiel illustriert: Seien in der Datenrelation r im verschlüsselten Attribut a' die Nachnamen „Lehnhardt“, „Spalka“ und „Rho“ verschlüsselt gespeichert. Seien die seq-Werte der entsprechenden Tupel die Werte 1, 2, und 3. Weiterhin gelte für a' eine beliebige Collation, die durch die Funktion f ausgedrückt sei.

7.1.3.1 Verändertes Schema der M:N-Relation

Die sich dann aus den (Normalisierungen der) drei Nachnamen ergebenden Suffix-Werte sind in Abb. 6 dargestellt. Weiterhin werden die Verknüpfungstupel in der M:N-Relation dargestellt, inklusive den jeweiligen Werten des $offset$- (linker Wert) und des $length$-Attributs (rechter Wert).

Es sei zu beachten, dass die Fremdschlüsselbeziehung zwischen der M:N-und der Indexrelation sich nicht immer als 1:1-Beziehung ausbilden muss, so wie es in Abb. 6 der Fall ist; vielmehr dürfte es die Regel sein, dass mehrere M:N-Tupel auf dasselbe Indextupel verweisen.

7.1.3.2 Veränderte Extension der Indexrelation

Weiterhin sei in Abb. 7 die Binärbaumdarstellung der linearen Ordnung dargestellt, die sich aus den in Abb. 6 angegebenen 18 Suffix-Indexwerten ergibt. Man sieht leicht, dass die Suffixe sich genauso in dem binären (AVL-)Baum anordnen, wie es zuvor die Indexwerte getan haben, die ausschließlich vollständige Wörter indexiert haben.

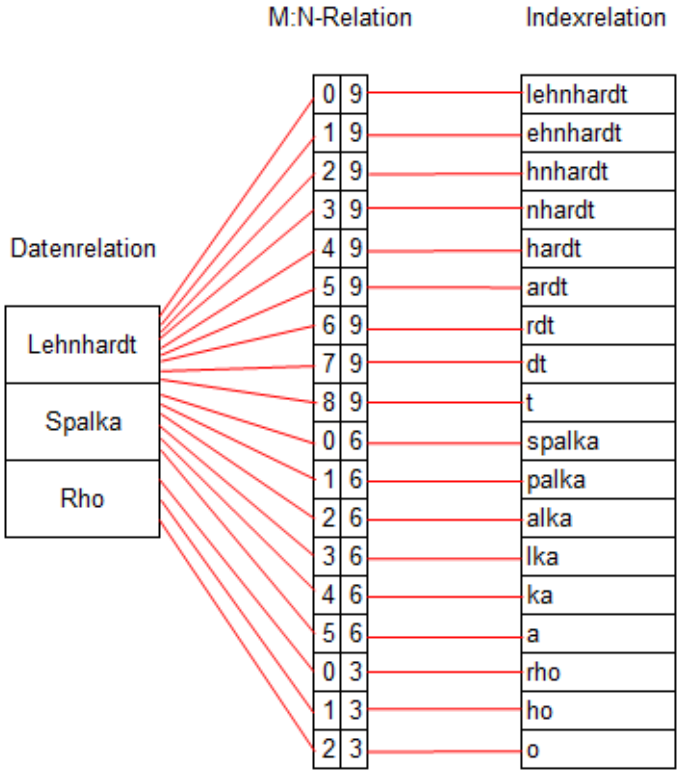

Abb. 6: Erweiterte cTree-Index- und M:N-Relation für die Indexsuche.

7.1.3.3 Veränderte finale SELECT-Anweisung

Ein weiterer Unterschied zur bisherigen, in Kapitel 6 beschriebenen Funktionsweise des Index liegt darüber hinaus in der finalen SELECT-Anweisung auf Daten-, M:N und Indexrelation; dies wird in Kapitel 7.3 detailliert beschrieben. Hier kann die in den neu hinzugefügten Attributen $offset$ und/oder $length$ gespeicherte Information als Auswahlkriterium in die WHERE-Klausel miteinbezogen werden, um verschiedene Ausprägungen von Infix- und Postfixsuche zu implementieren, welche in Kapitel 7.3.1 behandelt werden.

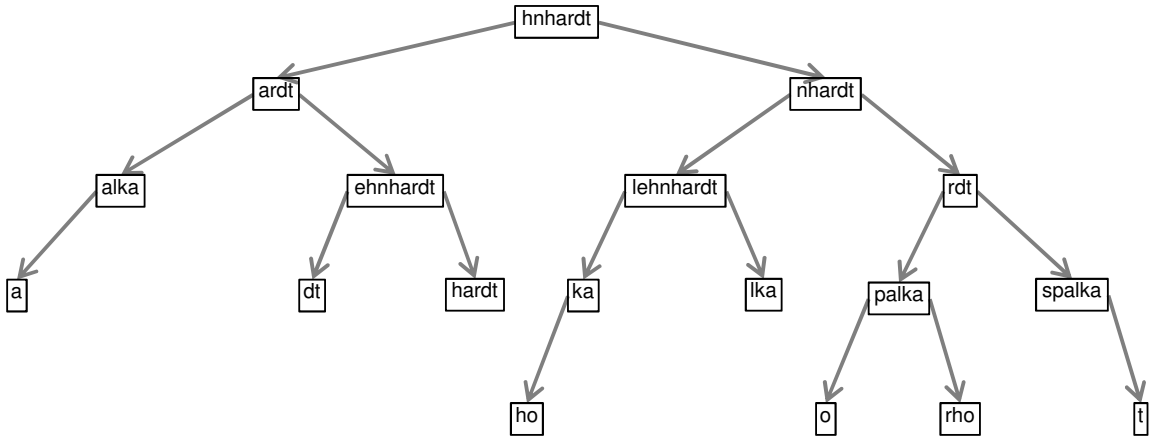

Abb. 7: cTree-Suffix-Indexbaum für die Werte „lehnhardt“, „spalka“ und „rho“.

7.2 Änderungen in DML-Operationen

Die DML-Basisoperationen auf einer cTree-Indexrelation, INSERT, UPDATE und DELETE (siehe Kapitel 6.4 bis 6.7), müssen entsprechend den in Kapitel 7.1 beschriebenen Erweiterungen der cTree-Indexdatenstruktur angepasst werden. Diese Änderungen werden im Folgenden beschrieben.

7.2.1 INSERT

Sei t ein Tupel einer Datenrelation r, welche ein verschlüsseltes, in der Indexrelation $ix_{r.a'}^{cTree}$ indexiertes Attribut a' enthalte. r und $ix_{r.a'}^{cTree}$ seien über die M:N-Relation $mn_{r.a'}^{cTree}$ miteinander verknüpft. Sei weiterhin $f\left(Dec\left(\pi_{a'}(t)\right)\right) = w = \sigma_{i_1} \dots \sigma_{i_n}$ mit $\sigma_{i_j} \in \Sigma$, $|w| = n$.

Zur Indexierung eines Tupels t wird dieses nicht mehr mit nur genau einem Tupel $t_{ix} \in ix_{r.a'}^{cTree}$ über genau ein Tupel in der M:N-Relation $mn_{r.a'}^{cTree}$ verknüpft (wobei t_{ix} gegebenenfalls zu $ix_{r.a'}^{cTree}$ hinzugefügt werden muss), wobei gelte:

$$Dec\left(\pi_{ixData}(t_{ix})\right) = f\left(Dec\left(\pi_{a'}(t)\right)\right) \quad \textbf{(F53)}$$

Stattdessen werden in $ix_{r.a'}^{cTree}$ genau $|w| = n$ Tupel lokalisiert bzw., falls notwendig, neu eingefügt. Ihr $ixData$-Attributwert ist jeweils einer der n Suffixe $s_i \in suf(w), 1 \leq i \leq n$ von w in verschlüsselter Form. Deshalb bestehe die INSERT-Operation bezüglich t im Kontext des Shrinking Window-Ansatzes aus

- dem Einfügen von t in r,
- n Lokalisierungs- bzw. INSERT-Operationen auf $ix_{r.a'}^{cTree}$ sowie
- n INSERT-Operationen auf der modifizierten M:N-Relation $mn_{r.a'}^{cTree}$, bei denen auch die neuen Attribute $offset$ und $length$ mit entsprechenden Werten belegt werden.

7.2.2 UPDATE

Die in Kapitel 7.2.1 beschriebene Änderung der INSERT-Operation setzt sich analog bei der UPDATE-Operation fort, erweitert um die Einbeziehung des vorhergehenden Zustands des aktualisierten Attributs. Sei t ein Tupel aus r, dessen Attributwert $\pi_{a'}(t)$ im Zuge einer UPDATE-Operation von $Enc(w)$ auf $Enc(x)$ aktualisiert werde. Dann sind die folgenden Schritte durchzuführen:

- Alle Suffixe aus $suf_{old} = suf(w) \backslash suf(x)$, die im neuen Wert x nicht mehr enthalten sind, müssen aus der Indexierung von t entfernt werden. Hierzu werden alle Tupel aus $mn_{r.a'}^{cTree}$ gelöscht, die Indextupel t_{ix} referenzieren mit $Dec\left(\pi_{ixData}(t_{ix})\right) \in suf_{old}$.
- Die oben genannten Indextupel t_{ix} mit $Dec\left(\pi_{ixData}(t_{ix})\right) \in suf_{old}$ können aus $ix_{r.a'}^{cTree}$ gelöscht werden, sofern sie nicht von anderen M:N-Tupeln referenziert werden. Sie können auch im Index verbleiben, wenn etwa zu erwarten ist, dass sie demnächst wieder für eine Indexierung benötigt werden können.
- Alle M:N-Tupel, die Indextupel t_{ix} referenzieren mit $Dec\left(\pi_{ixData}(t_{ix})\right) \in suf_{\cap} = suf(x) \cap suf(w)$, bleiben in $mn_{r.a'}^{cTree}$ enthalten. Allenfalls müssen ihre $offset$- bzw. $length$-Attributwerte aktualisiert werden.

- Für jedes $s \in suf_{new} = suf(x) \backslash suf(w)$ wird ein entsprechendes Indextupel t_{ix} mit $Dec\left(\pi_{ixData}(t_{ix})\right) = s$ in $ix_{r.a'}^{cTree}$ eingefügt, falls es dort nicht bereits enthalten ist. Weiterhin wird für alle $s \in suf_{new}$ ein Tupel in $mn_{r.a'}^{cTree}$ eingefügt, das t und t_{ix} verbindet.

7.2.3 DELETE

Bei der Löschung eines Datentupels t werden analog zur in 7.2.2 beschriebenen UPDATE-Operation alle t referenzierenden Tupel in $mn_{r.a'}^{cTree}$ gelöscht, bevor t selbst gelöscht werden kann. Auch hier gilt, dass diejenigen Indextupel, die nach den oben genannten Löschungen aus $mn_{r.a'}^{cTree}$ von keinem weiteren Tupel in $mn_{r.a'}^{cTree}$ mehr referenziert werden, ebenfalls gelöscht werden können, falls dies anwendungsseitig gewünscht sein sollte.

7.3 Änderung bei Range SELECT

Mit den in den Kapiteln 5 und 6 vorgestellten Indexierungstechniken sind bisher mit Gleichheits- und Präfixsuche nur zwei verschiedene Suchtypen möglich. In diesem Unterkapitel werden weitere Suchtypen vorgestellt, die durch die oben beschriebene Modifikation der cTree-Indexstruktur auf verschlüsselten Daten ermöglicht werden.

7.3.1 Neue Suchtypen und reguläre Ausdrücke

Wieder wird von einer Datenrelation r ausgegangen, die ein verschlüsseltes Attribut a' enthält. Die Klartextwerte von $X\left(Dec\left(\pi_{a'}(r)\right)\right)$, der entschlüsselten Extension von a', seien lexikografisch geordnete Zeichenketten über dem Alphabet $\Sigma = \{\sigma_1, \dots, \sigma_n\}$. Weiterhin sei die Interpretation von semantischer Gleichheit auf Wörtern $w_1, w_2 \in \Sigma^*$ dann gegeben, wenn bezüglich einer Normalisierungsfunktion $f \subseteq \Sigma^* \times \Sigma^*$ gilt: $f(w_1) = f(w_2)$. Gemäß Kapitel 5 liefert dann eine Gleichheitssuche auf r nach einem Suchbegriff $s = \sigma_{s_1} \dots \sigma_{s_m}$ mit $s_f = f(s)$ alle Tupel $t \in X(r)$ zurück, für die gilt:

$$f\left(Dec\left(\pi_{a'}(t)\right)\right) = s_f \quad \textbf{(F54)}$$

[70]

Eine Präfixsuche auf dem Attribut liefert dagegen gemäß Kapitel 6.8 alle Tupel $t \in X(r)$ zurück, bei denen s_f den ersten $|s_f|$ Zeichen von $f\left(Dec\left(\pi_{a'}(t)\right)\right)$ bitweise gleicht und $f\left(Dec\left(\pi_{a'}(t)\right)\right)$ nach dem Präfix s_f keine oder beliebig viele Symbole $\sigma \in \Sigma$ enthält.

Im Folgenden werde eine Präfixsuche mit dem regulären Ausdruck $\omega_1 = s\Sigma^*$ ausgedrückt, während die Gleichheitssuche durch $\omega_2 = s$ repräsentiert werde.

Neben den beiden oben genannten, durch die regulären Ausdrücke s und $s\Sigma^*$ ausgedrückten Suchtypen sind weitere Suchvarianten denkbar. Eine Variante der Präfixsuche ist etwa, die Anzahl der dem Suchbegriff nachfolgenden beliebigen Symbole konkret festzulegen, etwa eine Suche nach allen Tupeln $t \in X(r)$, bei denen auf den Präfix s_f genau n beliebige Symbole folgen dürfen. Dies sei im regulären Ausdruck durch einen entsprechenden Exponent vom Symbol des Alphabets Σ ausgedrückt, welches mit s konkateniert wird:

$$\omega_3 = s\Sigma^n \quad \textbf{(F55)}$$

[70] Das hier angegebene Gleichheitszeichen sei als bitweise Gleichheit zu verstehen.

Verallgemeinert ergeben sich drei Möglichkeiten für Symbole, die den regulären Ausdruck modifizieren (sie werden im Folgenden *Modifikatoren* genannt) und die Semantik der spezifizierten Suche beeinflussen:

- Σ^0: keine Modifikation
- Σ^n: eine präzise Anzahl von n beliebigen Zeichen
- Σ^*: 0 oder beliebig viele beliebige Zeichen

Je ein Modifikator mit den drei genannten möglichen Ausprägungen kann sowohl vor als auch nach s angegeben werden. Kombinatorisch ergeben sich daraus also 9 verschiedene Suchtypen, die auf die vier Kategorien Gleichheitssuche, Präfix-, Infix- und Postfixsuche verteilt werden können. Sie sind in Tabelle 1 aufgelistet.

Ausdruck	Suchtyp	Erläuterung
s	Gleichheitssuche	
$s\Sigma^n$	Präfixsuche	Feste Suffixlänge
$s\Sigma^*$	Präfixsuche	Beliebige Suffixlänge
$\Sigma^n s$	Postfixsuche	Feste Präfixlänge
$\Sigma^m s\Sigma^n$	Infixsuche	Feste Präfix- und feste Postfixlänge
$\Sigma^n s\Sigma^*$	Infixsuche	Feste Präfix- und beliebige Postfixlänge
$\Sigma^* s$	Postfixsuche	Beliebige Präfixlänge
$\Sigma^* s\Sigma^n$	Infixsuche	Beliebige Präfix- und feste Postfixlänge
$\Sigma^* s\Sigma^*$	Infixsuche	Beliebige Präfix- und Postfixlänge

Tabelle 1: Verschiedene Suchtypen-Ausprägungen.

Während bisher nur die in den Zeilen 1 und 3 genannten Suchtypen realisiert werden konnten, werden nun durch die in Kapitel 7.1 beschriebene Erweiterung der cTree-Indexstrukturen alle in Tabelle 1 aufgelisteten Suchtypen möglich. Dazu sind Veränderungen an der bisherigen Form der RANGE SELECT-Anweisung erforderlich, die in den folgenden Unterkapiteln vorgestellt werden.

7.3.2 Infixsuche

In diesem Unterkapitel werden die vier Ausprägungen der Infixsuche $\Sigma^* s\Sigma^*$, $\Sigma^* s\Sigma^n$, $\Sigma^n s\Sigma^*$ und $\Sigma^n s\Sigma^m$ behandelt, die auf der in Kapitel 6.8 vorgestellten RANGE SELECT-Operation aufbauen. Dabei werden insbesondere die nötigen Modifikationen und konkrete Implementierungen der von der Präfixsuche bekannten finalen SQL-Anweisung betrachtet.

7.3.2.1 $\Sigma^* s\Sigma^*$

Die Infixsuche in ihrer simpelsten Form, die keine Einschränkungen bezüglich der Anzahl von Zeichen macht, die sich vor oder hinter s befinden dürfen, kann durch die folgende beispielhafte Klartext-SQL-Anfrage illustriert werden:

```
SELECT * FROM r WHERE a LIKE '%sch%'
```
(Q24)

Sie ähnelt der Präfixsuche: Wie bisher wird im FROM-Teil der SQL-Anfrage eine JOIN-Operation zwischen Index-, M:N- und Datenrelation durchgeführt, bei der die Antwortmenge über ein in der WHERE-Klausel angegebenes, durch zwei $sortNumber$-Attributwerte und entsprechende Ungleichheitsoperatoren spezifiziertes Intervall ermittelt wird. Die $sortNumber$-Attributwerte sind jeweils in einem vorangegangenen Lookup-Schritt ermittelt

worden wie in Kapitel 6.8 beschrieben. Dennoch ergeben sich im Vergleich zur dortigen Vorgehensweise die folgenden Änderungen:

- Die SELECT-Klausel muss durch das Schlüsselwort DISTINCT ergänzt werden, da durch die möglicherweise mehrfach vorkommenden Verknüpfungen zwischen Daten- und Indextupeln auch entsprechende Duplikate in der Antwortmenge entstehen können. Diese müssen durch den DISTINCT-Operator herausgefiltert werden.
- Die Sortierung der Antwortmenge über das $sortNumber$-Attribut wird anders realisiert als bei der Präfixsuche: Hier muss die Indexrelation, die das sortierende Attribut indexiert, ein zweites Mal durch eine JOIN-Operation in den FROM-Teil der SQL-Anfrage aufgenommen werden. Bei diesem zweiten JOIN werden nur solche M:N-Tupel einbezogen, die einen $offset$-Attributwert von 0 haben, damit eine korrekte Sortierung durchgeführt wird.[71]

Die folgende, über die Variablen $@sn_{p1}$ und $@sn_{p2}$ parametrisierte SQL-Anfrage implementiert den finalen Teil der RANGE SELECT-Anfrage für den Infixsuchen-Term $\Sigma^* s \Sigma^*$. Die cTree-Indexrelation $ix_{r.a'}$ wird, wie oben beschrieben, zwei Mal mit der Datenrelation r über eine JOIN-Operation verbunden: Das erste Mal (Alias **`ix`**) dient zur Auswertung des Suchintervalls im WHERE-Kriterium und das zweite Mal (Alias **`ix2`**) zur Sortierung.

```
SELECT DISTINCT r.*
FROM ix_r.a' AS ix
  JOIN mn_r.a' AS mn ON mn.refIndex = ix.seq
  JOIN r ON mn.refData = r.seq
  JOIN mn_r.a' AS mn2 ON mn2.refData = r.seq AND mn2.Offset = 0
  JOIN ix_r.a' AS ix2 ON mn2.refIndex = ix2.seq
WHERE ix.sortNumber >= @sn_p1
  AND ix.sortNumber <= @sn_p2
ORDER BY ix2.sortNumber[72]                                    (Q25)
```

7.3.2.2 $\Sigma^n s \Sigma^*$

Analoge Klartext-SQL-Anfrage:

```
SELECT * FROM r WHERE a LIKE '__sch%'                          (Q26)
```

Falls die Anzahl der Zeichen, die vor s vorkommen dürfen, auf einen bestimmten, in einem im Parameter $@prefixOffset$ spezifizierten Wert beschränkt ist, muss die WHERE-Klausel der in (Q25) angegebenen Anfrage um den folgenden Term erweitert werden:

```
AND mn.Offset = @prefixOffset                                  (Q27)
```

7.3.2.3 $\Sigma^* s \Sigma^n$

Analoge Klartext-SQL-Anfrage:

```
SELECT * FROM r WHERE a LIKE '%sch_'                           (Q28)
```

[71] Angesichts dessen, dass für den Zweck der Sortierung ein zweiter JOIN durchgeführt werden muss, dürfte es jedoch in vielen Anwendungsfällen, insbesondere bei Ergebnismengen mit wenigen Tupeln, günstiger sein, darauf zu verzichten und die Treffermenge im Client zu sortieren.

[72] Aus Übersichtlichkeitsgründen wurden einige, für die Kernaussage dieses Programmfragments unwesentliche Implementierungsdetails weggelassen.

Falls dagegen die Anzahl der Zeichen, die nach s vorkommen dürfen, auf einen bestimmten Wert beschränkt und dieser im Parameter $@postfixOffset$ spezifiziert ist, muss die WHERE-Klausel der Anfrage stattdessen um den folgenden Term erweitert werden:

```
AND mn.length - dt.Offset = @postfixOffset
```
(Q29)

7.3.2.4 $\Sigma^m s \Sigma^n$

Analoge Klartext-SQL-Anfrage:

```
SELECT * FROM r WHERE a LIKE '__sch_'
```
(Q30)

Wenn sowohl Präfix als auch Postfix längenmäßig jeweils auf einen konkreten Wert beschränkt sind, sind beide oben genannten Terme in der WHERE-Klausel anzugeben:

```
AND mn.Offset = @prefixOffset
AND mn.[length] - mn.Offset = @postfixOffset
```
(Q31)

7.3.3 Postfixsuche

Die Postfixsuche ist einfacher zu implementieren als die Infixsuche. Dies hat den Grund, dass im cTree-Index alle Suffixe (*Postfixe*) der indizierten Zeichenkettenwerte explizit gespeichert sind, so dass man nach ihnen mit den Mitteln der Gleichheitssuche suchen kann.

7.3.3.1 $\Sigma^* s$

Analoge Klartext-SQL-Anfrage:

```
SELECT * FROM r WHERE a LIKE '%sch'
```
(Q32)

Im Folgenden werde eine Postfixsuche nach einem (normalisierten) Suchbegriff s_f betrachtet, bei der die Anzahl der Symbole, die vor dem Suchbegriff vorkommen dürfen, 0 oder beliebig groß sein darf. Sie entspricht technisch einer Gleichheitssuche auf den in der erweiterten cTree-Indexstruktur gespeicherten Suffixen.

Für Gleichheitssuchen wurde bereits in Kapitel 6.8.4 gezeigt, dass sie auch unter Verwendung der cTree-Indexstruktur beschleunigt durchgeführt werden können. Für die erweiterte cTree-Indexstruktur gilt dies ebenso. Bei einer Postfixsuche werden diejenigen Tupel aus r in die Ergebnismenge aufgenommen, die mit einem Indextupel t_{ix} verknüpft sind, für das der in Kapitel 6.8.3.2 vorgestellte Lookup-Schritt einen exakten Treffer mit s_f produziert:

$$Dec\left(\pi_{ixData}(t_{ix})\right) = s_f \qquad \textbf{(F56)}$$

Das Ergebnis eines solchen Lookup-Schritts sei der $sortNumber$-Attributwert $sn_{tix} = \pi_{sortNumber}(t_{ix})$, so dass die finale SELECT-Operation unter Verwendung des Parameters $@sn_{tix}$ folgendermaßen ausgedrückt werden könne:

```
SELECT r.*
FROM ix_r.a, AS ix
  JOIN mn_r.a, AS mn ON mn.refIndex = ix.seq
  JOIN r ON mn.refData = r.seq
  JOIN mn_r.a, AS mn2 ON mn2.refData = r.seq AND mn2.Offset = 0
  JOIN ix_r.a, AS ix2 ON mn2.refIndex = ix2.seq
WHERE ix.sortNumber = @sn_tix
ORDER BY ix2.sortNumber
```
(Q33)

Eine DISTINCT-Duplikateliminierung ist nicht erforderlich, da bei den hier angegebenen JOIN-Operationen keine Duplikate auftreten können. Die Sortierung hingegen muss wieder über einen separate JOIN-Operation mit der Indexrelation realisiert werden.

Alternativ zu dem oben beschriebenen Vorgehen, das bei beliebigen Verschlüsselungsverfahren von $ixData$-Attributwerten anwendbar ist, kann im Falle der Verwendung von definiter Verschlüsselung eine alternative, performantere Variante der Gleichheitssuche durchgeführt werden, die für die ursprüngliche Form der cTree-Indexstruktur bereits in den Kapiteln 6.5.3.1 und 6.8.4.2 vorgestellt wurde. Hier wird der normalisierte Suchwert s_f mit dem selben Schlüssel definit verschlüsselt wie die $ixData$-Werteinstanzen des cTree-Indexes:

$$s'_f = Enc_{det}(s_f) = Enc_{det}(f(s)) \quad \textbf{(F57)}$$

Bei Anwendung auf die erweiterte cTree-Indexstruktur kann die Postfixsuche ohne vorangehenden Lookup-Schritt in Form der folgenden SQL-Anweisung ausgeführt werden:

```
SELECT r.*
FROM ix_r.a, AS ix
  JOIN mn_r.a, AS mn ON mn.refIndex = ix.seq
  JOIN r ON mn.refData = r.seq
  JOIN mn_r.a, AS mn2 ON mn2.refData = r.seq AND mn2.Offset = 0
  JOIN ix_r.a, AS ix2 ON mn2.refIndex = ix2.seq
WHERE ix.ixData = @s_f'
ORDER BY ix2.sortNumber
```
(Q34)

7.3.3.2 $\boldsymbol{\Sigma^n s}$

Analoge Klartext-SQL-Anfrage:

```
SELECT * FROM r WHERE a LIKE '__sch'
```
(Q35)

Soll nur eine feste Anzahl von Zeichen vor s vorkommen, dann muss der in (Q33) bzw. (Q34) spezifizierten SELECT-Anfrage in der WHERE-Klausel der bereits in (Q27) angegebene Term **`AND mn.Offset = @prefixOffset`** hinzugefügt werden.

7.3.4 Präfixsuche

7.3.4.1 $\boldsymbol{s\Sigma^*}$

Analoge Klartext-SQL-Anfrage:

```
SELECT * FROM r WHERE a LIKE 'sch%'
```
(Q36)

Abweichend von ihrer bisherigen Definition in Kapitel 6.8 lässt sich die Präfixsuche auf den veränderten cTree-Indexstrukturen als Spezialfall der Infixsuche ausführen, der lediglich leicht anders parametrisiert werden muss. Ein entsprechendes, über $@sn_{p1}$ und $@sn_{p2}$ parametrisierbares SQL-Statement für eine Präfixsuche mit einer beliebigen Anzahl von auf den Suchbegriff folgenden Zeichen lautet dann:

```
SELECT r.*
FROM ix_r.a, AS ix
  JOIN mn_r.a, AS mn on mn.refIndex = ix.seq AND mn.offset = 0
  JOIN r ON mn.refData  = r.seq
WHERE ix.sortNumber >= @sn_p1
  AND ix.sortNumber < @sn_p2
ORDER BY ix.sortNumber
```
(Q37)

7.3.4.2 $s\Sigma^n$

Analoge Klartext-SQL-Anfrage:

```
SELECT * FROM r WHERE a LIKE 'sch___'
```
(Q38)

Wenn die Anzahl der dem Suchbegriff folgenden Zeichen auf einen durch den Parameter $@postfixOffset$ festgelegten Wert beschränkt sein soll, ist der in (Q29) angegebene Term „**`AND mn.[length] - mn.Offset = @postfixOffset`**“ der SQL-Anfrage hinzuzufügen.

7.3.5 Gleichheitssuche

Analoge Klartext-SQL-Anfrage:

```
SELECT * FROM r WHERE a = 'sch'
```
(Q39)

Die Gleichheitssuche auf der erweiterten cTree-Indexstruktur entspricht technisch der in 7.3.3 behandelten Postfixsuche, bei der in den Elementen der Treffermenge vor dem normalisierten Suchbegriff keine Zeichen vorkommen dürfen. Dementsprechend kann die Gleichheitssuche bei beliebiger Wahl des Verschlüsselungsverfahrens von $ixData$ in $ix_{r.a'}^{cTree}$ durch die SQL-Anweisung (Q33), bzw. bei definiter Verschlüsselung von $ixData$ auch durch (Q34) ausgedrückt werden, wobei in beiden Fällen dem WHERE-Kriterium der bereits erwähnte Term „**`AND mn.offset = 0`**“ hinzugefügt werden muss.

7.3.6 Weitere Variationsmöglichkeiten

7.3.6.1 Komplexere reguläre Ausdrücke

Auch komplexere reguläre Ausdrücke wie z. B. $\Sigma^* s \Sigma^* t \Sigma^*$ lassen sich mit den bisher in Kapitel 7 vorgestellten Techniken implementieren; sie müssten nur aus mehreren einfacheren Ausdrücken zusammengesetzt werden. So kann der Ausdruck $\Sigma^* s \Sigma^* t \Sigma^*$ als Schnittmenge zweier regulärer Ausdrücke dargestellt werden:

$$\Sigma^* s \Sigma^* t \Sigma^* = (\Sigma^* s \Sigma^*) \cap (\Sigma^* t \Sigma^*) \quad \textbf{(F58)}$$

Hier würden zwei Infixsuchen nach $\Sigma^* s \Sigma^*$ und $\Sigma^* t \Sigma^*$ ausgeführt werden, und die Schnittmenge ihrer Antwortmengen würde dem Client als Ergebnismenge zurückgegeben werden.

7.3.6.2 Spezifische Zeichen

Für die in Tabelle 1 aufgeführten Suchtypen, bei denen dem Suchbegriff ein oder mehrere Zeichen vorangehen oder folgen können, gilt, dass es sich dabei um beliebige Zeichen des zugrunde liegenden Alphabets Σ handeln kann. Bei der praktischen Anwendung eines Informationssystems kann es jedoch Anfragen geben, bei denen dem Suchbegriff beispielsweise eine Anzahl von Ziffern folgen soll, aber keine Buchstaben, wie es etwa die folgende SQL-Klartextanfrage beschreibt:

```
SELECT * FROM r WHERE a LIKE 'xyz[0-9][0-9][0-9]'
```
(Q40)

Da auf dem Server in einem verschlüsselten Informationssystem lediglich Informationen über die Struktur, nicht aber über den Inhalt der gespeicherten Daten vorliegen, kann er bei einer solchen Anfrage keine Beihilfe leisten.

Dennoch sind solche Suchtypen mit wenig Aufwand auch unter Verwendung des cTree-Index implementierbar, wenn die Filterung nach Ziffern in die Clientlogik verlagert wird.

Dann liefert der Server eine Antwortmenge von Tupeln an den Client, bei denen die spezifizierte Anzahl von Zeichen dem Suchbegriff vorangeht bzw. folgt. Die Einschränkung auf Tupel, bei denen diese Zeichen z. B. ausschließlich Ziffern sind, erfolgt im Client. Unter Inkaufnahme von Overhead beim Datentransfer würde also auch dieser Typ von komplexeren Suchanfragen implementiert werden können.

7.4 Informationsverlust

Es soll nicht unerwähnt bleiben, dass die in diesem Kapitel behandelte Erweiterung der cTree-Indexstruktur eine Preisgabe von Klartextinformation bedeutet, die in der ursprünglichen Form des cTree-Indexes nicht erfolgt. So lassen sich durch die Verwendung von Suffixen und insbesondere die expliziten Angaben in der M:N-Relation die Längen der im Index und damit auch der in der indizierten Datenrelation gespeicherten Wörter erkennen.

Dieser Aspekt sollte miteinbezogen werden, wenn zur Designzeit eines Informationssystems entschieden wird, welche Daten auf welche Art und Weise indiziert werden sollen. Kapitel 10 widmet sich dem Thema der Informationspreisgabe durch die vorgestellten Indexstrukturen und behandelt auch mögliche Abwehrstrategien, die die Gefahr von Informationsverlust eliminieren oder zumindest signifikant abschwächen.

8 Volltext-Indexe auf verschlüsselten Daten

Nach Indexierungstechniken für Gleichheits-, Intervall-, Präfix-, Infix- und Postfixsuche wird in diesem Kapitel eine Anwendung dieser Techniken behandelt, die einen weiteren Suchtyp auf verschlüsselten Daten ermöglicht. Er operiert auf verschlüsselten Freitextdaten und implementiert darauf einen verschlüsselten *Volltext-Index*. Es wird nicht der Anspruch erhoben, den gesamten Funktionsumfang, den gängige Volltextindexe für die Suche auf Klartextdaten bieten, in den verschlüsselten Kontext zu transferieren. Stattdessen soll eine möglichst große Teilmenge dieses Funktionsumfangs erarbeitet werden, die ohne signifikanten Informationsverlust auf verschlüsselten Daten zum Einsatz kommen kann.

Im Folgenden sei die Erstellung eines Volltext-Index auf verschlüsselten Daten in zwei Phasen unterteilt, für die jeweils unterschiedliche Implementierungsansätze vorgestellt und bewertet werden. In der ersten Phase erfolge die Zerlegung eines zu indexierenden Freitext-Zeichenkettenwerts x in einzelne Tokens (*Tokenisierung*), welche anschließend verschlüsselt gespeichert werden. Die in Frage kommenden Formen der Tokenisierung werden in Kapitel 8.1 behandelt.

Nach der Tokenisierung werde eine Indexierungstechnik für verschlüsselte Daten auf die Tokens angewendet. Hierfür können prinzipiell alle Techniken verwendet werden, die in den Kapiteln 5, 6 und 7 vorgestellt wurden. Jedoch sind dabei manche Techniken besser mit manchen Formen der Tokenisierung kombinierbar als andere. Kapitel 8.2 behandelt diese Indexierungstechniken und deren Kombinierbarkeit mit Tokenisierungsansätzen. Weiterhin enthält Kapitel 8.2 auch eine Betrachtung, welche Suchformen bei den jeweiligen Kombinationen von Tokenisierungs- und Indexierungstechniken möglich werden.

Die so erarbeiteten Techniken werden in Kapitel 8.3 bewertet. Weiterhin werden ihnen vergleichbare, aus der Literatur bekannte Techniken gegenüber gestellt. Dabei liegt der Fokus auf den *keyword search*-basierten Verfahren, insbesondere mit ihren Ausprägungen *fuzzy search* und *ranked search*, wie sie am Anfang von Kapitel 2 vorgestellt wurden.

8.1 Token-Zerlegung

Alle in diesem Unterkapitel beschriebenen Tokenisierungsschritte operieren auf Klartextdaten. Daher müssen sie in einem Informationssystem mit verschlüsseltem Backend zwingend auf der Clientseite ausgeführt werden.

8.1.1 Vorarbeiten und erste Stufe der Tokenisierung

Der natürlichsprachliche Textstring x, im Folgenden *Freitext* genannt, sei die Ausgangssituation der Tokenisierung. x sei der entschlüsselte Wert einer Instanz x' eines beliebig verschlüsselten Attributs a' einer Datenbankrelation r (die Verschlüsselung schließe sowohl definite als auch randomisierte Verschlüsselung ein).

x werde in einer ersten Stufe der Tokenisierung anhand eines oder mehrerer Spezialzeichen aus dem Alphabet, über dem x gebildet wurde, zerlegt. Es sind zahlreiche Formen der Implementierung dieser ersten Stufe der Tokenisierung denkbar; eine ihrer einfachsten Formen ist die sogenannte *Whitespace-Tokenisierung*, bei der Leer- und/oder Interpunktionszeichen die Rolle der zur Segmentierung verwendeten Spezialzeichen übernehmen. Ergebnis der ersten Stufe der Tokenisierung ist eine Menge von Zeichenfolgen, die in einer zweiten Stufe

der Tokenisierung weiter zerlegt werden (im Folgenden werden die Ergebnisse der ersten Stufe *Tokens* und die der zweiten Stufe *Sub-Tokens* genannt). Die Sub-Tokens werden dann verschlüsselt als Tupel in einer Relation $subTokens_{r.a'}$ gespeichert und können dann mit den Indexierungstechniken, die in den Kapiteln 5 bis 7 vorgestellt wurden, mit verschlüsselten Indexen versehen werden.

In den folgenden Unterkapiteln 8.1.2 bis 8.1.5 werden Implementierungsmöglichkeiten für die zweite Stufe der Tokenisierung kurz vorgestellt.

8.1.2 Naiver Ansatz

In einem naiven Ansatz werden alle Tokens von x ohne weitere Tokenisierungs-Stufe als Eingabewerte für die zweite Phase der Volltext-Indexierung verwendet; die Menge der Sub-Tokens sei also gleich der Menge der Tokens. Allenfalls werden die Tokens einer simplen Normalisierung[73] (siehe Kapitel 5.5) unterzogen. Dementsprechend kann bei einer Volltextsuche nach Suchbegriff s auf derart tokenisierten Freitexten ein indexiertes Token nur dann gefunden werden, wenn seine Schreibweise nicht durch morphologische Varianten wie Deklinationen oder Konjugationen von s abweicht.

8.1.3 n-Gramme

Eine gängige Methode für die Zerlegung von Tokens in Sub-Tokens ist ihre Zerlegung in *Fragmente*. Dabei kann es sich um Buchstaben, Phoneme, Wörter oder ähnliches handeln. Von den Fragmenten werden dann jeweils n aufeinanderfolgende zu *n-Grammen* zusammengefasst und als Sub-Tokens interpretiert. So lässt sich beispielsweise der Nachname „Lehnhardt“ in die folgenden (auf Kleinschreibung normierten) 3-Gramme (*Trigramme*) zerlegen: „leh“, „ehn“, „hnh“, „nha“, „har“, „ard“ und „rdt“.

8.1.4 Suffixe

Die Suffix-Zerlegung orientiert sich an der in Kapitel 7 vorgestellten „shrinking window“-Zerlegung und generiert als Sub-Tokens alle Suffixe eines Eingabewertes, welcher üblicherweise entsprechend Kapitel 5.5 normiert sei. So wird der normalisierte Eingabewert „lehnhardt“ in folgende Suffixe, d. h. Sub-Tokens, zerlegt: „lehnhardt“, „ehnhardt“, „hnhardt“, „nhardt“, „hardt“, „ardt“, „rdt“, „dt“ und „t“.

Entsprechend Kapitel 7 eignen sich diese Suffixe zum einen für eine Suche auf Gleichheit, was eine Postfixsuche auf den indexierten Tokens implementiert (siehe Kapitel 7.3.3). Wird dagegen eine Präfixsuche (siehe Kapitel 7.3.2) auf die Suffixe angewendet, implementiert dies stattdessen eine Infixsuche.

8.1.5 Stemming

In einer besonderen Form der Verarbeitung der Tokens werden höher entwickelte, aus der Computerlinguistik stammende sogenannte *Stemming*-Verfahren zur Tokenisierung angewendet, bei denen verschiedene morphologische Varianten eines Wortes auf ihren gemeinsamen Wortstamm abgebildet werden. Letzterer nehme hier die Rolle des Sub-Tokens ein, auf den ein Token abgebildet wird. So wird es beispielsweise möglich, bei einer Suche, deren

[73] Diese Normalisierung muss keine komplexen Transformationsschritte beinhalten; es kann sich auch einfach um die Umwandlung des Eingabewertes in Kleinbuchstaben handeln.

Suchbegriff „sehen“ lautet, einen indizierten Freitext in die Treffermenge aufzunehmen, der das Satzfragment „Ich sah“ enthält.

Als Beispiel lässt sich der Satz anführen „Wer anderen eine Grube gräbt, fällt selbst hinein“, dessen Wort-Tokens vom Stemming-Algorithmus „Snowball German“[74] in die folgenden Sub-Tokens zerlegt werden: {„wer“, „and“, „ein“, „grub“, „grabt“, „fallt“, „selb“, „hinein“}.

Aus Platzgründen wird hier nicht auf Details solcher Stemming-Algorithmen eingegangen, zumal diese nicht im Fokus des Themas der Dissertation liegen. Es sei jedoch festgehalten, dass sie, da sie auf Klartextdaten operieren, auf der Clientseite implementiert werden müssen, da nur dort die zu verarbeitenden Tokens und Sub-Tokens im Klartext vorliegen.

Entscheidend ist an dieser Stelle dagegen, dass auch Stemming-Algorithmen diskrete, als Sub-Tokens interpretierbare Werte als Ergebnisse liefern. Diese können dann verschlüsselt und mit den in dieser Arbeit vorgestellten Methoden indexiert werden.

8.2 Indexierung

Auf die im vorangegangenen Kapitel beschriebene Tokenisierung folgt in der zweiten Phase der Volltext-Erstellung die Speicherung und Indexierung der generierten Sub-Tokens.

8.2.1 Speicherung von Sub-Tokens und Tokens

Die Menge der aus einem Freitext x generierten Sub-Tokens werde verschlüsselt in einer Datenbankrelation $subTokens_{r.a'}$ im designierten Attribut $data$ gespeichert, wobei dies vorzugsweise nichtredundant geschehe. Für jedes Sub-Token st werde ein Tupel t_{st} in $subTokens_{r.a'}$ eingefügt, das den verschlüsselten Wert st' enthalte (jedoch nur, falls ein solches Tupel nicht bereits in $subTokens_{r.a'}$ enthalten ist).

x werde verschlüsselt im Attribut a' eines Tupels t_x der Datenrelation r gespeichert. Die Assoziierung mit seinen Sub-Tokens werde durch die M:N-Relation $mn_{r.a'}^{subTokens}$ dargestellt: Für jedes assoziierte Sub-Token-Tupel t_{st} werde ein Tupel in $mn_{r.a'}^{subTokens}$ eingefügt, das t_x und t_{st} referenziere. Für den in Kapitel 8.1.3 behandelten Tokenisierungsansatz der n-Gramm-Zerlegung bietet sich dabei noch eine Erweiterung an: Dieser Ansatz generiert potenziell mehrere Sub-Tokens aus einem Token; um deren Zusammengehörigkeit darzustellen, werde $mn_{r.a'}^{subTokens}$ um ein Attribut tk_{id} erweitert, das einen beliebigen, aber einheitlichen Token-Identifikationswert speichere. Für alle Sub-Tokens von x, die aus dem selben Token tk generiert wurden, werde derselbe Wert id_{tk} in tk_{id} gespeichert.

8.2.2 Indexierung für Gleichheitssuche

Dieses Unterkapitel beschreibt die Vorgehensweise bei der Indexierung der Sub-Tokens, die eine Gleichheitssuche nach einem Suchparameter s darauf ermöglicht, unter Berücksichtigung der in Kapitel 8.1 vorgestellten Tokenisierungsansätze.

8.2.2.1 Naiver Tokenisierungsansatz

s (bzw. seine Normalisierung s_f) werde clientseitig definit zu $Enc_{det}(s) = s'$ (bzw. s_f') verschlüsselt und an den Server gesendet. Dieser ermittele entsprechend der in Kapitel 5.5.4 beschriebenen Vorgehensweise das Tupel $st_{s'}$ in $subTokens_{r.a'}$, das s' enthält. Anschließend

[74] siehe [URL-Snowball].

ermittele der Server als Treffermenge der Gleichheitssuche alle Tupel $t \in r$, die über $mn_{r.a'}^{subTokens}$ mit $st_{s'}$ verknüpft sind. Auf diese Weise ermittelt die Gleichheitssuche alle Freitext-Werteinstanzen, in denen s als Teilwort in der gleichen Schreibweise (abgesehen von der Normalisierung) mindestens einmal vorkommt.

8.2.2.2 n-Gramm-Tokenisierung

Der Client zerlege s in gleicher Weise in n-Gramme wie zuvor die Tokens bei der zweiten Stufe der Tokenisierung, (siehe Kapitel 8.1.3). Er verschlüssele alle aus s entstehenden n-Gramme $ng_1, \ldots, ng_m$ definit zu $ng'_1, \ldots, ng'_m$ und sende diese an den Server. Letzterer führe nach jedem der n-Gramme eine Gleichheitssuche auf $subTokens_{r.a'}$ aus, was im Erfolgsfall eine Menge von bis zu m Tupeln von $subTokens_{r.a'}$ liefere. Analog zu Kapitel 8.2.2.1 stelle der Server die Treffermenge der Suche zusammen, indem er alle Tupel $t \in r$ ermittele, die mit den Tupeln der Treffermenge aus $subTokens_{r.a'}$ über $mn_{r.a'}^{subTokens}$ verknüpft sind.

Diese Implementierung der Gleichheitssuche lässt sich noch dahingehend erweitern, dass ein Schwellwert $m_{th} \geq 1$ eingeführt werde mit der Forderung, dass für ein Tupel $t \in r$ mindestens m_{th} verschiedene n-Gramme „getroffen" haben müssen, damit t in die Treffermenge aufgenommen werde.

Bei einer direkten Verknüpfung von r mit $subTokens_{r.a'}$ über $mn_{r.a'}^{subTokens}$ besagt dieser Schwellwert, dass mindestens m_{th} n-Gramme über beliebige Tokens mit t verknüpft sein müssen, damit t in die Treffermenge aufgenommen wird. Im in Kapitel 8.2.1 erwähnten alternativen Szenario, in dem die Zusammengehörigkeit von Sub-Tokens in $mn_{r.a'}^{subTokens}$ mit dem Attribut tk_{id} gekennzeichnet ist, lässt sich die Aussagekraft des Schwellwerts m_{th} verfeinern: Danach gelangen nur solche Tupel $t \in r$ in die Treffermenge, die mindestens ein Token tk enthalten, das seinerseits mindestens m_{th} Sub-Tokens enthält, welche bei den m Gleichheitssuchen auf $subTokens_{r.a'}$ einen Treffer produziert haben.

8.2.2.3 Suffix-Tokenisierung

Analog zum naiven Ansatz werde s (bzw. seine Normalisierung) im Suffix-Ansatz definit zu s' verschlüsselt. Der Client übertrage s' an den Server, welcher nach einem Tupel $t_{s'} \in subTokens_{r.a'}$ suche, für das gilt: $\pi_{data}(t_{s'}) = s'$. Falls ein solches $t_{s'}$ gefunden wurde, werden analog zu Kapitel 8.2.2 die mit $t_{s'}$ über $mn_{r.a'}^{subTokens}$verbundenen Tupel $r_s \subseteq r$ als Treffermenge ermittelt. Jedes Tupel $t \in r_s$ enthält dann als entschlüsselten Wert von a' einen Freitext $x = Dec(\pi_{a'}(t))$, der mindestens ein Wort enthält, das s als Suffix beinhaltet.

8.2.2.4 Stemming

Beim Stemming-Ansatz wende der Client den bei der Tokenisierung angewendeten Stemming-Algorithmus, repräsentiert durch die Funktion $stem$, auf s an und erhalte die morphologische Grundform $s_{morph} = stem(s)$, welche der Client definit zu $s'_{morph} = Enc_{det}(s_{morph})$ verschlüssele und an den Server übertrage. Letzterer führe eine Gleichheitssuche auf $subTokens_{r.a'}$ nach einem Tupel $t_{s'_{morph}}$aus, dessen entschlüsselter $data$-Attributwert s'_{morph} bitweise gleiche. Bei einem Treffer stelle der Server wieder über $mn_{r.a'}^{subTokens}$ eine Tupelmenge $r_{s'_{morph}} \subseteq r$ als Treffermenge zusammen, wobei für jedes Tupel $t \in r_{s'_{morph}}$ gelte, dass t mindestens ein Wort enthalte, das sich mit $stem$ auf die morphologische Grundform wie s zurückführen lasse.

8.2.3 Indexierung für Präfixsuche

Dieses Unterkapitel beschreibt die Indexierung eines beliebig verschlüsselten Freitext-Attributs a' einer Datenbankrelation r gemäß Kapitel 6. Eine solche Indexierung ermöglicht unter anderem Präfixsuchen auf den Sub-Tokens der Attributinstanzen von a'. Zur Indexierung verschlüssele der Client alle bei der Tokenisierung entstandenen Sub-Tokens beliebig (auch randomisierte Verschlüsselung ist zulässig) und füge die Kryptotexte entsprechend der lexikografischen Ordnung, der die Sub-Tokens unterliegen, in eine cTree-Indexrelation $cTree_{r.a'}^{subTokens}$ ein (siehe Kapitel 6.4). Über eine M:N-Relation $mn_{r.a'}^{subTokens}$ seien diese Sub-Token-Indextupel mit jedem Tupel $t \in r$ verknüpft, in dem sie als Sub-Token enthalten sind.

Bei einer Suchoperation auf a' werde die Extension von $cTree_{r.a'}^{subTokens}$ entsprechend der in Kapitel 6.8 definierten Vorgehensweise anhand des eingegebenen Suchwerts durchsucht, was eine Tupelmenge $tuples_{s'} \subseteq cTree_{r.a'}^{subTokens}$ als Ergebnis liefere. Der Server ermittele daraufhin die Tupelmenge $r_{s'} \subseteq r$, die mit den Tupeln aus $tuples_{s'}$ über $mn_{r.a'}^{subTokens}$ verknüpft sind und sende $r_{s'}$ als Suchergebnis an den Client zurück.

Die Präfixsuche in Kombination mit jeweils einem der in Kapitel 8.1 vorgestellten Tokenisierungsansätzen ist folgendermaßen zu bewerten:

8.2.3.1 Naiver Tokensisierungsansatz

Die Implementierung der Präfixsuche bei der Wahl des naiven Tokenisierungsansatzes ähnelt der in Kapitel 6.8 vorgestellten Vorgehensweise:

- Der Client leite zunächst aus dem eingegebenen (und gegebenenfalls normalisierten) Suchbegriff s ein Suchintervall $[s, inc(s)[$ ab.
- Anschließend ermittele der Server mit Unterstützung des Client das Supremum der unteren und das Infimum der oberen Intervallgrenze in $X\left(cTree_{r.a'}^{subTokens}\right)$. Diese Suche liefere die Indextupel $t_s^{cTree}, t_{inc(s)}^{cTree} \in cTree_{r.a'}^{subTokens}$ als Ergebnis.
- Schließlich ermittele eine finale SELECT-Anweisung alle Indextupel t^{cTree}, für die gilt: $\pi_{sortNumber}(t_s^{cTree}) \leq \pi_{sortNumber}(t^{cTree}) \leq \pi_{sortNumber}\left(t_{inc(s)}^{cTree}\right)$. Aus dieser Indextupelmenge ermittele der Server alle über $mn_{r.a'}^{subTokens}$ verknüpften Tupel $t \in r$ und gebe diese als Treffermenge an den Client zurück.

Im Vergleich zur Vorgehensweise in Kapitel 6.8 gibt es lediglich den Unterschied, dass bei der Implementierung für einen Volltextindex ein Tupel $t \in r$ nicht mit genau einem, sondern mit mehreren cTree-Indextupeln über $mn_{r.a'}^{subTokens}$ verknüpft sein kann, da der indexierte Freitext mit mehr als einem Sub-Token verknüpft sein kann.

Bei der Eingabe eines Präfix-Suchbegriffs wird die Datenrelation demnach nach denjenigen Tupeln durchsucht, bei denen das cTree-indexierte Attribut mindestens ein Wort-Token enthält, das mit dem eingegebenen Suchbegriff beginnt.

8.2.3.2 n-Gramm-Tokenisierung

Der n-Gramm-Tokenisierungsansatz ist eher ungeeignet für die Präfixsuche. Analog zu Kapitel 8.2.2.2 gelte für diesen Ansatz, dass eine Suche mindestens ein zu einem Token gespeichertes n-Gramm oder aber eine Anzahl oberhalb eines definierten Schwellwerts m_{th} solcher n-Gramme in der cTree-Indexextension auffinden muss, damit ein Tupel $t \in r$ Teil des Suchergebnisses ist. Da $X\left(cTree_{r.a'}^{subTokens}\right)$ ausschließlich n-Gramme enthält, müsste bei einer Präfixsuche der Suchbegriff s ebenfalls in eine Anzahl von n-Grammen zerlegt werden, nach

denen separat in $X\left(cTree_{r.a'}^{subTokens}\right)$ gesucht würde. Dann würde der Server die ermittelten Treffermengen entsprechend m_{th} filtern und das Ergebnis an den Client senden.

Es ist zu bedenken, dass bei der Verwendung von n-Grammen nach diskreten Werten gesucht wird. Also ist eine Gleichheitssuche nach n-Grammen nötig, während eine Intervall- oder Präfixsuche keinen Mehrwert bietet. Weiterhin kann aus einem als Suchwert eingegebenen Präfix aufgrund seiner beschränkten Länge nur ein Teil der zu einem gesuchten Token gehörenden n-Gramme erzeugt werden, so dass ein etwaiger Schwellwert k_{th} unterhalb einer Mindestlänge des Präfix inhärent nicht erreichbar ist. All dies spricht gegen die Präfixsuche auf der cTree-Indexstruktur unter Verwendung des n-Gramm-Tokenisierungsansatzes.

8.2.3.3 Suffix-Tokenisierung

Bei Verwendung des in Kapitel 8.1.4 vorgestellten Tokenisierungsansatzes, bei dem die Tokens in ihre Suffixe zerlegt und letztere in der cTree-Indexstruktur gespeichert werden, ist die Präfixsuche analog zum naiven Tokenisierungsansatz (siehe Kapitel 8.2.3.1) durchführbar. Für ihr Resultat gilt analog zu Kapitel 7, dass dadurch, dass der cTree-Index Suffixe enthält, eine Präfixsuche auf dessen Extension eine Infixsuche auf den indexierten Tokens implementiert. Analog zu Kapitel 8.2.3.1 stellt ein Tupel $t \in r$ nicht eine einzelne Werteinstanz dar, auf die alle Suffixe im Index verweisen, sondern repräsentiert eine Vielzahl von Wort-Tokens, die über die M:N-Relation mit all ihren Suffixen verknüpft sind.

8.2.3.4 Stemming

Bezüglich einer Tokenisierung über Stemming-Algorithmen (siehe Kapitel 8.1.5) ist für die Präfixsuche zu beachten, dass die cTree-Indexrelation im Attribut $ixData$ nur Werte enthält, die durch die Abbildung der in einer Freitext-Werteinstanz enthaltenen Wort-Tokens auf ihren jeweiligen Wortstamm (z. B. „ess“ bei „essen“ oder „seh“ bei „sehen“) entstanden sind. Folglich können Präfixsuchen nur auf diesen Wortstamm-Instanzen durchgeführt werden. Selbige können jedoch oft nur dann gebildet werden, wenn das Ausgangswort vollständig und nicht nur zum Teil eingegeben ist.

Somit müsste der Benutzer bei einer Präfixsuche auf einer solchen Index-Ausprägung gehalten sein, entweder nur vollständige Begriffe als Suchparameter abzusetzen (dann wäre es jedoch keine Präfix-, sondern eine Gleichheitssuche), oder er müsste bereits den Wortstamm des Suchbegriffs kennen und einen Präfix von selbigem eingeben, um die gewünschten Ergebnisse zu bekommen. Somit ist auch Stemming nur eingeschränkt mit der cTree-Präfixsuche kombinierbar.

8.3 Bewertung

8.3.1 Zusammenfassung

Die Kapitel 8.2.2 und 8.2.3 haben gezeigt, dass jeder in Kapitel 8.1 vorgestellte Tokenisierungsansatz auf mindestens einen Suchtyp in verschlüsselten Volltextindexen angewendet werden kann. So ist die Gleichheitssuche gemäß Kapitel 5 mit allen vier genannten Tokenisierungsansätzen kombinierbar und die Präfixsuche mit dem naiven und dem Suffix-Tokenisierungsansatz. Mit diesen Kombinationen lassen sich die folgenden Suchtypen realisieren:

- Gleichheitssuche auf dem naiven Tokenisierungsansatz: Exact match

- Gleichheitssuche auf *n*-Gramm- bzw. Stemming-basiertem Tokenisierungsansatz: Fuzzy match
- Gleichheitssuche auf dem Suffix-Tokenisierungsansatz: Postfixsuche
- Präfixsuche auf dem naiven Tokenisierungsansatz: Präfixsuche
- Präfixsuche auf dem Suffix-Tokenisierungsansatz: Infixsuche

Es sollte in Betracht gezogen werden, dass die in diesem Kapitel beschriebene Volltextindexierung einen Informationsverlust mit sich bringen kann, etwa durch die Eigenschaften der definiten Verschlüsselung bei der Gleichheitssuche, aber auch durch die Verknüpfung von Volltextinstanzen mit Indexwerten über die M:N-Relation (siehe auch Kapitel 7.4). Aus diesem Grund sollten die in Kapitel 10 vorgestellten Maßnahmen gegen Informationsverlust auch hier Anwendung finden.

8.3.2 Vergleich

Bezüglich der in diesem Kapitel beschriebenen Suchtechniken bietet sich ein Vergleich mit verschiedenen aus der Literatur bekannten Verfahren an, wobei insbesondere die *keyword search*-basierten Verfahren zu nennen sind (siehe etwa [Song2000], [Boneh2004], [Abdalla2005] und [Boyen2007]). Bezüglich einer Menge von verschlüsselten *Dokumenten* (entsprechen den in diesem Kapitel erwähnten Volltextinstanzen) ist dort eine Menge von ebenfalls verschlüsselten *keywords* vorhanden, von der jeweils Teilmengen mit den einzelnen Dokumenten assoziiert sind. Dies ist vergleichbar mit der Gleichheitssuche auf dem naiven Tokenisierungsansatz aus Kapitel 8.2.2.1, der nach exakten Übereinstimmungen ganzer Wörter sucht. In [Li2010] wird ein keyword search-Ansatz präsentiert, der ein fuzzy matching ermöglicht, so dass es der Kombination von Gleichheitssuche und *n*-Gramm- und Stemming-Tokenisierungsansatz gleicht. Was dagegen fehlt, sind Techniken zur Präfix- Postfix- und Infixsuche, wie sie in den Kapiteln 8.2.2.3, 8.2.3.1 und 8.2.3.3 vorgestellt wurden. [Wang2014] beschreibt ein Verfahren für die Suche nach mehreren eingegebenen keywords, das sich leicht auf die hier beschriebenen Techniken übertragen lässt. In einem anderen Ansatz in [Cao2014] wird ein Ranking auf den Treffern der multi-keyword search eingeführt. Dies ist eine sinnvolle Erweiterung, die der verschlüsselten Volltext-Indizierung aus diesem Kapitel ebenfalls hinzugefügt werden kann, etwa bei der Anwendung des *n*-Gramm-Tokenisierungsansatzes.

9 Performanz-Optimierung des cTree-Indexes

In diesem Kapitel werden verschiedene Maßnahmen vorgestellt, die der Basisimplementierung des cTree-Indexes (siehe Kapitel 6) zur Performanzsteigerung hinzugefügt werden können. Diese Maßnahmen sind unterschiedlichen Typs: die zunächst vorgestellten sind naheliegend und werden mehr der Vollständigkeit halber erwähnt, wie etwa der gezielte Einsatz konventioneller Datenbankindexe und redundant gespeicherter Information. Im späteren Verlauf des Kapitels werden dann ausgereiftere Techniken zur Performanzsteigerung wie *Pipelining* (siehe Kapitel 9.6) und *Caching* (siehe Kapitel 9.7) vorgestellt.

9.1 Herkömmliche Datenbankindexe

Eine naheliegende Maßnahme ist die Definition konventioneller Datenbankindexe auf der cTree-Datenbankrelation ix^{cTree}. Diese seien so gewählt, dass die in Kapitel 6 beschriebenen Zugriffsoperationen auf ix^{cTree} optimal beschleunigt werden.

Gemäß der Theorie von Datenbankindexen[75] empfiehlt sich eine Analyse, welche Arten von Datenzugriffen auf den betreffenden Attributextensionen im Zuge der Zugriffsoperationen durchgeführt werden, um eine Entscheidung über Art und Einsatzort der Indexe treffen zu können. Ist etwa abzusehen, dass auf einer Attributextension vorrangig Intervallsuchen durchgeführt werden, empfiehlt sich ein B-bzw. B⁺-Baum-Index[76]. Wenn dagegen auf der Extension eines Attributs ausschließlich punktuelle Suchen ausgeführt werden soll, sollte der Index als Hash-Index implementiert werden.[77]

So übernimmt das Surrogatattribut seq als Kandidatenschlüssel die Rolle der Tupelidentifikation in ix^{cTree}. Mit $parent$, $leftChild$ und $rightChild$ verweisen drei Fremdschlüsselattribute per Selbstreferenz auf seq, so dass zahlreiche punktuelle Suchen darauf stattfinden. Folgerichtig ist ein Hash-Index auf seq empfehlenswert. Das Attribut $sortNumber$ ist ebenfalls häufig Teil von Zugriffspfaden (der „finale SELECT“, siehe Kapitel 6.8.3.4) jedoch nie als Bestandteil einer punktuellen, sondern stets von Intervallsuchen, so dass sich hier ein B- oder B⁺-Baum-Index auf $sortNumber$ empfiehlt.

9.2 Redundante Attribute

In der cTree-Datenstruktur wird redundante Information eingesetzt, um die Operationen auf dem Index zusätzlich zu beschleunigen.

9.2.1 Die Baumstruktur konstituierende Fremdschlüsselattribute

Das Fremdschlüssel-Attribut $parent$, das per Selbstreferenz den jeweiligen Elternknoten eines Indexknotens in der Extension von ix^{cTree} referenziert, konstituiert eine Anordnung der Tupel in Form eines gerichteten Graphen. Per Konvention handele es sich dabei um eine binäre Baumstruktur, im Folgenden T genannt. Jedes cTree-Tupel repräsentiert dabei einen Knoten von T. $parent$ wird dabei ausschließlich serverseitig bei der Zusammenstellung von Teilbäumen τ_k von T mit einer maximalen Höhe h zugegriffen. Suchanfragen auf $parent$ sind somit häufig, weshalb sich ein Index darauf empfiehlt (einem Hash-Index, denn es handelt sich ausschließlich um punktuelle Suchen).

---

[75] siehe [Kemper2011], S. 213-239.
[76] siehe [Kemper2011], S. 208-222.
[77] siehe [Kemper2011], S. 222-226.

Ermittelte Teilbäume τ_k werden an den Client gesendet und dort traversiert, wobei vom Wurzelknoten v_k (d. h.: $v_k \in ix^{cTree}: \pi_{seq}(v_k) = k$) ausgehend bei jedem Knoten v_n des durchwanderten Pfades eine Entscheidung getroffen werden muss, ob der nächste Traversierungsschritt auf dem linken oder dem rechten Kindknoten von v_n durchgeführt werden soll. Würde bezüglich des Auffindens des Kindknotens nur die in $parent$ gespeicherte Information zur Verfügung stehen, müsste folgendermaßen vorgegangen werden:

1. Ermittle in τ_k die zwei Knoten v_i und v_j mit $\pi_{parent}(v_i) = \pi_{parent}(v_j) = n$.
2. Ermittele anhand der $sortNumber$-Attributwerte $\pi_{sortNumber}(v_i)$ und $\pi_{sortNumber}(v_j)$, welcher der beiden Kindknoten der linke und welcher der rechte Nachkomme von v_k ist.
3. Verschiebe den Fokus für den nächsten Traversierungsschritt auf den richtigen Kindknoten.

Wenn ix^{cTree} die beiden zusätzlichen, redundanten Attribute $leftChild$ und $rightChild$ enthält, in deren Exension der linke und rechte Kindknoten jedes Tupels direkt referenziert ist, kann beim Traversierungsschritt das obige Verfahren vermieden und der Fokus des nächsten Traversierungsschritts direkt auf den richtigen Kindknoten gesetzt werden.

9.2.2 Angabe der Baumhöhe

Angesichts dessen, dass für die praktische Implementierung der binären Baumstruktur ein AVL-Baum gewählt wurde, ist die Information, welche Höhe der jeweilige Teilbaum unter einem beliebigen Knoten v von T hat, bei der Balancierung von T erforderlich. Sie ist implizit im Baum enthalten; dennoch ist es von Vorteil, sie in jedem Knoten zu materialisieren, um sie bei Balancierungsoperationen nicht immer neu berechnen zu müssen. Also wird ix^{cTree} ein Attribut $height$ hinzugefügt, das diese Information beinhaltet. Bei Änderungsoperationen auf ix^{cTree} müssen $height$-Werteinstanzen wenn nötig aktualisiert werden.

9.2.3 Explizite Kindknotentypangabe

Ein neues Tupel in ix^{cTree} wird bezüglich der (AVL-)Baumdarstellung des Index als neues Blatt v_{new} in T eingefügt. Anschließend ist es für die Aufrechterhaltung der AVL-Balanciertheit von T erforderlich, dass der Pfad von v_{new} bis zur Wurzel v_{root} von T durchlaufen wird, um etwaige, durch das Einfügen von v_{new} entstandene, Dysbalancen zu beheben. Ist der Algorithmus bei v_{root} angekommen, ist T wieder balanciert. Da sich das „Rückwärts-Traversieren“ auf jedem Knoten v_i des Pfades stets des $parent$-Attributwerts bedient, ist auf jedem Knoten zunächst unbekannt, ob v_i das linke oder das rechte Kind seines Elternknotens ist. Diese Information wird jedoch beim Balancieren des Baumes sowie bei der Ermittlung von Änderungspfaden (siehe Kapitel 9.7.3) benötigt.

Anstelle eines Algorithmus, der über $parent$ zum Elternknoten von v_i navigieren muss, um dessen Kindknotentyp zu ermitteln, bietet es sich an, letzteren in v_i selbst zu materialisieren. D. h., ix^{cTree} wird ein neues Attribut $isLeftChild$ vom Datentyp $boolean$ hinzugefügt. So liegt die Information zum Kindknotentyp von v_i stets direkt vor, anstatt dass erst ein Zugriff auf die Kindknotenreferenzen des Vaterknotens Aufschluss über diese Information gibt. Auch hier gilt, dass $isLeftChild$-Attributwerte bei Änderungsoperationen gegebenenfalls aktualisiert werden müssen.

9.3 Subtree-Retrieval

Dieses Unterkapitel beschäftigt sich mit der Performanzsteigerung bei der Traversierung von T zum Auffinden von Tupeln bzw. Einfügepositionen (der „Lookup"; siehe Kapitel 6.4.2). T habe die Höhe H. Ohne Beschränkung der Allgemeinheit gelte im Folgenden, dass T ausgehend von v_{root} zu einem Knoten v_i auf der Blattebene traversiert werden soll, was einem Traversierungspfad der Länge H entspreche. Alle Knoten auf diesem Traversierungspfad müssen zu diesem Zweck zum Client transferiert, entschlüsselt und bewertet werden.

In einem ersten, naiven Lösungsansatz sendet der Server den gesamten Baum T an den Client, also die gesamte Extension $X(ix^{cTree})$, was oft ein großes Datenvolumen darstellt. Ein zweiter naiver Lösungsansatz sieht vor, jeden Knoten einzeln zum Client zu übertragen, welcher ihn entschlüsselt, bewertet und seinen Nachfolger vom Server anfordert. Auf diese Weise sind nur minimale Datenmengen zu übertragen, jedoch ergeben sich H Client-Server-Roundtrips, welche üblicherweise sehr teuer und in ihrer Anzahl zu minimieren sind.

Um gleichzeitig ein hinreichend geringes Datentransfervolumen zu gewährleisten und die Anzahl der Client-Server-Roundtrips niedrig zu halten, empfiehlt es sich, innerhalb eines Roundtrips anstatt einzelner Knoten gezielt größere strukturierte Teilmengen von T zum Client zu transferieren. Hierfür bieten sich Teilbäume τ_k von T der maximalen Höhe h an, also diejenige Menge von Knoten, für die v_k mit maximal h Transitionen über die $parent$-Verbindung erreichbar ist. Ein solcher Teilbaum, der einen Teil des Traversierungspfades enthält, kann auf dem Client entschlüsselt und traversiert werden.

Bei einer maximalen Teilbaumhöhe h sind anstatt H nur noch $\left\lceil \frac{H}{h} \right\rceil$ Roundtrips zur Traversierung von T nötig, was für eine Maximierung von h spricht. Gleichzeitig werden bei jedem Roundtrip bis zu $2^h - 1$ Knoten zum Server transferiert, wovon lediglich die h Knoten des Traversierungspfades benötigt werden, so dass für jeden Teilbaum bis zu $2^h - 1 - h$ Knoten umsonst transferiert werden. Dieser Wert wächst exponentiell mit h, was für dessen Minimierung spricht. Es manifestiert sich also ein Trade-off bezüglich der Wahl von h, so dass es mit Sorgfalt gewählt werden sollte. Im praktischen Anwendungsteil dieser Dissertation befasst sich das Unterkapitel 11.4 mit den Auswirkungen einer Variation von h.

9.4 AVL-Baum als Baum-Datenstruktur

Wie bereits mehrfach erwähnt, wurde für eine der beiden Darstellungen der linearen Ordnung in der cTree-Indexdatenstruktur der AVL-Baum gewählt. Er bietet den Vorteil einer guten Balance zwischen niedrigem Verwaltungsaufwand, insbesondere bei Rebalancierungsmaßnahmen, und Performanz. Weiterhin lässt sich die in Kapitel 9.3 beschriebene Teilbaumhöhe hier leicht variieren. Dies kann nötig werden , wenn die Roundtrip-Anzahl bzw. zu transferierende Datenmengen flexibel an sich ändernde Umgebungsparameter wie z. B. Netzwerklatenz oder Bandbreite angepasst werden müssen. Noch dazu ist dies komplett von der Clientseite aus steuerbar.

9.5 Definite Verschlüsselung

Sind die im cTree-Index gespeicherten Daten definit verschlüsselt, so dass gilt:

$$a = b \Leftrightarrow a' = Enc_{det}(a) = Enc_{det}(b) = b', \qquad \textbf{(F59)}$$

dann lässt sich diese Eigenschaft nutzen, um die Einfügung neuer Tupel unter bestimmten Voraussetzungen zu beschleunigen: Sei a' ein beliebig verschlüsseltes Attribut einer Relation r, auf dem ein cTree-Index definiert sei. Letzterer sei in der Relation $ix_{r.a'}^{cTree}$ gespeichert. Enthält die entschlüsselte Extension von a' viele Duplikate, dann ist es beim Einfügen eines neuen Tupels t in r wahrscheinlich, dass $x = Dec(\pi_{a'}(t))$ bereits in der entschlüsselten Extension von a' enthalten ist, bzw. dass $X\left(Dec\left(\pi_{ixData}(ix_{r.a'}^{cTree})\right)\right)$ bereits x, bzw. seine Normalisierung $f(x)$, enthält. In diesem Fall entfällt ein erneutes Einfügen von x in $ix_{r.a'}^{cTree}$, da der cTree-Index keine Duplikate enthalten soll.

Sei x ein entschlüsselter Indexwert. Sei weiterhin $ChkEx(x)$ ein Algorithmus, der die Existenz von x in $X\left(Dec\left(\pi_{ixData}(ix_{r.a'}^{cTree})\right)\right)$ effizient prüfen kann. Es gelte

$$CheckEx(x) \coloneqq \begin{cases} true, falls\ x \in X\left(Dec\left(\pi_{ixData}(ix_{r.a'}^{cTree})\right)\right) \\ false, sonst \end{cases} \quad \textbf{(F60)}$$

Seien $c_{INSERT}(x)$ und $c_{ChkEx}(x)$ Kostenfunktionen für die Einfüge- und die Prüfoperation auf Vorhandensein von x in $X\left(Dec\left(\pi_{ixData}(ix_{r.a'}^{cTree})\right)\right)$. Weiterhin werde die Einfügeoperation $INSERT(x)$ nur im Falle von $ChkEx(x) = true$ ausgeführt, was mit einer mittleren Wahrscheinlichkeit von

$$P = Prob\left[x \notin X\left(Dec\left(\pi_{ixData}(ix_{r.a'}^{cTree})\right)\right)\right] \quad \textbf{(F61)}$$

für beliebige x der Fall sei. Dann ist der Einsatz von $ChkEx$ genau dann sinnvoll, wenn gilt:

$$c_{INSERT}(x) > c_{ChkEx}(x) + P \cdot c_{INSERT}(x) \quad \textbf{(F62)}$$

Der Einsatz von $ChkEx$ lohnt sich also genau dann, wenn die konstanten Kosten von $Lookup$ die eingesparten Kosten von obsoleten Ausführungen von $INSERT$ unterschreiten. Bei hohen Kosten für $INSERT$ und vergleichsweise niedrigen Kosten für $ChkEx$ kann die Ungleichung schon für niedrige Werte für P erfüllt sein.[78]

Ein Weg, $ChkEx$ mit geringem Aufwand zu implementieren, ist es,

- die Extension von $\pi_{ixData}(ix_{r.a'}^{cTree})$ defninit zu verschlüsseln,
- $ixData$, das den Datentyp $varbinary$ hat, einem Hash-Index zu versehen und
- bei $Lookup$ folgendermaßen vorzugehen:
  1. ***(Client)*** Verschlüssele x zu $x' = Enc_{det}(x)$. Sende x' an den Server.
  2. ***(Server)*** Führe eine Gleichheitssuche $\sigma_{ixData=x'}(ix_{r.a'}^{cTree})$ aus. Definiere $CheckEx$ damit folgendermaßen:

$$CheckEx(x) \coloneqq \begin{cases} true, falls\ \sigma_{ixData=x'}(ix_{r.a'}^{cTree}) \neq \emptyset \\ false, sonst \end{cases} \quad \textbf{(F63)}$$

[78] Bei einer realen Implementierung von $CheckEx$ zeigte sich, dass $c_{CheckEx}$ erheblich kleiner war als c_{INSERT} und sich der Einsatz von $CheckEx$ schon für kleine Werte von P lohnte.

9.6 Pipelining

Dieses Unterkapitel befasst sich mit Performanzsteigerung durch den Einsatz von Parallelisierung. Da es sich bei dem hier beschriebenen Vorgehen jedoch nicht um eine reine Parallelisierung handelt, ist „Pipelining" der passendere Titel des Unterkapitels.

9.6.1 Pipelining beim Lesezugriff

Im Folgenden sei r eine Relation, in der die unverschlüsselten, linear geordneten Attribute a_1 und a_2 jeweils mit einem konventionellen Datenbankindex versehen seien. r enthalte weiterhin zwei linear geordnete, aber beliebig verschlüsselte Attribute a_1' und a_2', die jeweils mit einem cTree-Index versehen seien. Die folgende Selektion

$$s \subseteq r \text{ mit } s \coloneqq \sigma_{x_1 \leq a_1 \leq y_1 \wedge x_2 \leq a_2 \leq y_2}(r) \qquad \textbf{(F64)}$$

beschreibt eine Anfrage an r, die zwei Intervallsuchen nach $[x_1, y_1]$ und $[x_2, y_2]$ beinhaltet. Diese wird durch den folgenden SQL-String ausgedrückt:

```
SELECT *
FROM r
WHERE a1 BETWEEN x1 AND y1
  AND a2 BETWEEN x2 AND y2
```
(Q41)

Übertragen auf die verschlüsselt indizierten Attribute a_1' und a_2' von r führt die Ausführung einer analogen Anfrage zu jeweils einer Traversierung des dem indexierten Attribut zugeordneten cTree-Indexes für die Intervallgrenzen x_1, y_1, x_2 und y_2 (siehe Kapitel 6.8). Das Ergebnis jeder Traversierung ist ein Indextupel, welches durch seinen $sortNumber$-Attributwert sn_{x_i} bzw. sn_{x_i} repräsentiert wird. Es müssen also vier Index-Traversierungen vor der finalen Datenbankanfrage (siehe Kapitel 6.8.3.4) ausgeführt werden, welche einem SQL-String dieser Form entspricht:

```
SELECT (…)
FROM r
  JOIN ix_r.a1 AS t1 ON (…)
  JOIN ix_r.a2 AS t2 ON (…)
WHERE t1.sortNumber BETWEEN sn_x1 AND sn_y1
  AND t2.sortNumber BETWEEN sn_x2 AND sn_y2
```
(Q42)

Die finale Datenbankanfrage ist offensichtlich abhängig von den vorangegangenen Index-Traversierungen, kann also erst ausgeführt werden, wenn diese terminiert haben. Die Traversierungen sind dagegen voneinander unabhängig; dies gilt auch für eine höhere Anzahl von Traversierungen als vier, etwa für den Fall, dass das Selektionskriterium einer SQL-Anfrage mehr als zwei Bereichssuchen enthält: bei n Intervallsuchen im Selektionskriterium bedeutet dies bis zu $2n$ Traversierungen[79], von denen jede $\left\lceil \frac{H}{h} \right\rceil$ Client-Server-Roundtrips benötigt, also sind insgesamt bis zu $2n \cdot \left\lceil \frac{H}{h} \right\rceil$ Roundtrips nötig.

Anstatt diese Traversierungs-Roundtrips alle nacheinander auszuführen, können sie parallelisiert ausgeführt werden. Dabei ist zu erwähnen, dass in einer praktischen Implementierung die jeweiligen Client-Anteile der Traversierungsoperationen zwar voll parallelisiert ausgeführt werden können, der Datenbankserver seine Anteile dagegen in einem einzigen

[79] Eine Intervallsuche nach einem Werten im Intervall $]-\infty, y]$ würde nur eine cTree-Traversierung erfordern, so dass man bei n Intervallsuchen nicht zwingend von $2n$ Traversierungen sprechen kann.

Worker-Thread nacheinander abarbeitet. Deshalb ist „Pipelining“ der passendere Begriff für den Titel dieses Unterkapitels.

Diese Einschränkung wirkt sich in der Praxis jedoch wenig aus, da der Datenbankserver in einem Informationssystem, das cTree-Indexe verwendet, bei weitem nicht der Engpass des Systems ist. Implementierungen haben gezeigt, dass der Datenbankserver leicht zahlreiche Anfragen für cTree-Traversierungsoperationen bedienen kann, ohne der Gesamtbearbeitungszeit einer Anfrage viel zusätzliche Latenz hinzuzufügen[80]. Weiterhin hat sich gezeigt, dass eine Anfrage auf cTree-indizierten Relationen, die viele Intervallsuchen beinhaltet, nur unwesentlich länger braucht als eine analoge Anfrage mit nur einer Intervallsuche.

9.6.2 Pipelining beim Schreibzugriff

Wenn einem cTree-Index neue Elemente hinzugefügt werden sollen, ist ebenfalls Parallelisierung bzw. Pipelining zur Performanzsteigerung anwendbar. Wenn einem cTree-Index $n > 1$ Elemente hinzugefügt werden sollen, müssen auch der Baumdarstellung T der linearen Ordnung der Indexelemente neue Knoten hinzugefügt werden.

9.6.2.1 Pipelining bei der Ermittlung der Einfügeposition

Dies geschieht auf der Blattebene von T; hier werden diejenigen Blattknoten lokalisiert, an die die neuen Knoten als Kindknoten anzufügen sind, inklusive der Information, ob der neue Knoten als linkes oder rechtes Kind seines Elternknoten einzufügen ist (siehe Kapitel 6.4). Da es sich bei der Ermittlung dieser Einfügepositionen ausschließlich um lesende Zugriffe auf T handelt, kann dieser Schritt ebenfalls parallel, bzw. im Pipelining-Modus (siehe Kapitel 9.6.1), ausgeführt werden.

Nachdem alle Einfügepositionen ermittelt worden sind, können die neuen Knoten abschließend als neue Blätter mit der anschließenden Rebalancierung des Baums eingefügt werden (siehe Kapitel 6.5). Dieser Schritt muss vom Datenbankserver ebenfalls im “Pipelined”-Modus ausgeführt werden, um mögliche Bauminkonsistenzen zu vermeiden.

Falls alle n neuen Knoten unterschiedliche designierte Elternknoten besitzen, ist dieser Vorgang leicht implementierbar: Die Sequenz von n einzufügenden Knoten kann vom Server in beliebiger Reihenfolge abgearbeitet werden.

9.6.2.2 Konkurrierende Kindknoten

Haben jedoch $m > 1$ der neuen Knoten die gleiche designierte Einfügeposition (siehe Abb. 8), dann stehen diese neuen Knoten miteinander im Konflikt. In diesem Fall muss den Regeln zur Benutzung des AVL-Baums (siehe Kapitel 3.4) eine Ergänzung hinzugefügt werden:

Nach dieser Ergänzung werde ein neuer Knoten, der auf der Blattebene des Baums als linkes Kind seines Elternknotens hinzugefügt werden soll,

- entweder genau so eingefügt, wenn diese Einfügeposition von keinem weiteren, neu einzufügenden Knoten zum selben Zeitpunkt beansprucht wird[81],
- oder anderenfalls als rechter Kindknoten des größten Elements des linken Teilbaums des designierten Elternknotens eingefügt.

[80] siehe Kapitel 11.5.2.

[81] Hierbei handelt es sich also um die gängige, unmodifizierte Benutzung von AVL-Bäumen.

Alle anderen Schritte zur Verwaltung der AVL-Baum-Datenstruktur bleiben bestehen, inklusive der Rebalancierung des Baums.

Wenn der Server alle m im Konflikt stehenden neuen Knoten nun in einer bestimmten Reihenfolge einfügt, nämlich aufsteigend sortiert, so dass er mit dem kleinsten Element anfängt und nach jeder Einfügung eine Balancierung des Baumes vornimmt, wird die Konsistenz des AVL-Baums während und nach allen Einfügevorgängen aufrechterhalten.

Im symmetrischen Fall, d. h., bei einer Menge von $m > 1$ Knoten, die um die Position als rechtes Kind des selben Elternknotens konkurrieren, wird analog zur oben beschriebenen Vorgehensweise verfahren, wobei lediglich alle Schritte ins Gegenteil verkehrt werden: Die neuen Knoten werden absteigend sortiert, so dass mit dem größten Element begonnen wird, und ein neuer Knoten wird entweder als rechtes Kind des Elternknotens eingefügt oder als linkes Kind des kleinsten Elements des rechten Teilbaums des Elternknotens.

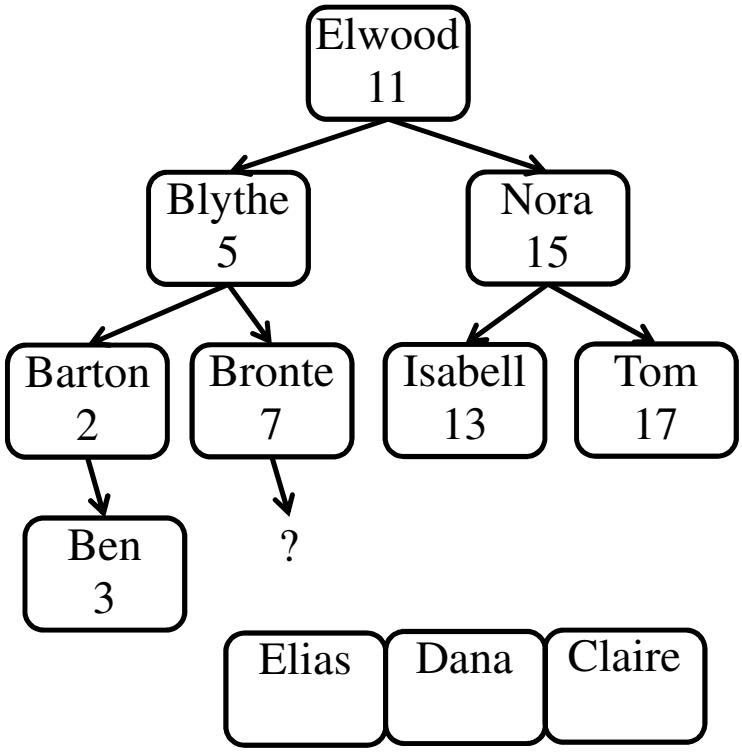

Abb. 8: Konkurrenz um die Einfügeposition als rechtes Kind desselben Elternknotens.

9.7 Caching-Strategien

Caching spielt im Zusammenhang mit die Performanz steigernden Maßnahmen eine wichtige Rolle. Grundsätzlich ist damit das Zwischenspeichern von für die Verarbeitung benötigten Daten in einem schnellen Speicher gemeint, die aus einem langsameren Speicher abgerufen worden sind. Bei ihrer nächsten Anforderung kann dann auf ein erneutes Anfragen verzichtet und stattdessen der schnellere Speicher zugegriffen werden[82].

Eine andere Form des Cachings ist das *read-ahead*-Caching, bei dem Daten, für die absehbar ist, dass sie demnächst benötigt werden, zwischengespeichert werden. Dieser Fall wird hier

[82] siehe [Patterson2011], S. C-11-C-12.

nicht berücksichtigt; stattdessen konzentriert sich dieses Unterkapitel auf den wiederholten Zugriff auf die selben Elemente einer Datenbankextension.

Es wird zwischen Caching bei Schreib- bzw. Lesezugriffen und zwischen client- und serverseitigem Caching unterschieden. Die folgenden Unterkapitel widmen sich den sich daraus ergebenden vier Ausprägungsformen von Caching bei der Verwendung des cTree-Index.

9.7.1 Serverseitiges Caching

Nach gängigen Implementierungsansätzen bietet sich für ein Informationssystem, das den cTree-Index nutzt, eine 3-Schicht-Architektur an, bestehend aus Client, Datenbankserver und Middleware. Letztere beinhalte ihrerseits einen Application und/oder Webserver. Der Application Server bietet sich als Stelle für die Implementierung serverseitigen Cachings an, da der Zugriff darauf kürzere Antwortzeiten ermöglicht als ein Zugriff auf die Datenbank. Zudem bietet diese Vorgehensweise Vorteile bezüglich Skalierung.

9.7.1.1 Naiver Ansatz

In einem ersten naiven Caching-Ansatz speichert der Application Server eine abgesetzte Suchanfrage nach Tupeln von r, deren entschlüsselter Attributwert von a' im Intervall $[x, y]$ liegt, inklusive ihrer Ergebnismengen $r_{[x,y]} \subseteq r$ als Cache-Eintrag $\chi_{[x,y]}$ auf dem Server.

Beispielsweise setze ein Client die Intervallsuchanfrage nach allen Werten eines durch den cTree-Index $ix_{r.a'}^{cTree}$ indizierten Attributs $r.a'$ ab, die im geschlossenen Intervall $[x, y]$ liegen.[83] Eine solche Anfrage führt gemäß Kapitel 6.8 zu zwei Traversierungen der Baumdarstellung T von $ix_{r.a'}^{cTree}$, die als Ergebnis die $sortNumber$-Attributwerte sn_x und sn_y liefern. Es folgt eine Selektion der Tupelmenge $ix_{[x,y]} \subseteq ix_{r.a'}^{cTree}$ aller Indexelemente, deren $sortNumber$-Attributwert zwischen sn_x und sn_y liegt:

$$ix_{[x,y]} \coloneqq \sigma_{x \leq Dec(ixData) \leq y}\left(ix_{r.a'}^{cTree}\right) = \sigma_{sortNumber \in [sn_x, sn_y]}\left(ix_{r.a'}^{cTree}\right) \quad \textbf{(F65)}$$

Die Ergebnismenge $r_{[x,y]} \subseteq r$, die sich aus $ix_{[x,y]}$ ableiten lässt, wird als serverseitiger Cache-Eintrag $\chi_{[x,y]}$ für eine erneute Anfrage nach $[x, y]$ gespeichert.

Als identifizierendes Merkmal $id_{[x,y]}$ für $\chi_{[x,y]}$ zu einer Anfrage nach einem Intervall $[x, y]$ eignen sich die definit verschlüsselten Intervallgrenzen x und y, zusammen mit der Spezifikation der beiden Intervallgrenzen:

$$id_{[a,b]} = \left(Enc_{det}(closed), Enc_{det}(a), Enc_{det}(b), Enc_{det}(closed)\right) \quad \textbf{(F66)}$$

Ein Cache-Eintrag hat also die Form:

$$\chi_{[x,y]} = (id_{[x,y]}, r_{[x,y]}) \quad \textbf{(F67)}$$

Ein Client, der erneut eine Selektionsanfrage nach allen Tupeln $t \in r$ stellt, deren Indexwert in $[x, y]$ liegt, verschlüsselt die Intervallspezifikation auf die in (F66) angegebene Weise und sendet das entstandene 4-Tupel $id_{[x,y]}$ an den Server. Dieser prüft seinen Cache, ob dort ein Eintrag mit $id_{[x,y]}$ vorhanden ist und sendet im Erfolgsfall dessen Ergebnismenge

[83] Im gesamten Verlauf des Unterkapitels 9.7 wird ohne Beschränkung der Allgemeinheit das geschlossene Intervall $[x, y]$ als Beispiel herangezogen.

$r_{[x,y]}$ im Erfolgsfall an den Client zurück. Findet sich kein Cacheeintrag, muss eine Anfrage an die Datenbank gestellt werden, deren Ergebnis dann dem Cache hinzugefügt wird.

9.7.1.2 Cachen der $sortNumber$-Attributwerte

Die in Kapitel 9.7.1.1 beschriebene grundsätzliche Vorgehensweise hat den Nachteil, dass der Speicherbedarf für die zu cachenden Datenmengen sehr schnell anwachsen kann. Eine kompaktere Darstellung der Daten erscheint wünschenswert, auch wenn dies geringfügige Performanzeinbußen bedeuten sollte.

Hier bietet sich eine Modifikation des naiven Ansatzes an: Der Server speichere zu denjenigen Indextupeln, die die Intervallgrenzen einer Intervallsuche nach $[x, y]$ darstellen, deren $sortNumber$-Attributwerte sn_x und sn_y anstelle der gesamten Ergebnismengen $r_{[x,y]}$. Die in (F66) beschriebene, verschlüsselte Identifikationsspezifikation $id_{[x,y]}$ des Suchintervalls wird beibehalten; lediglich der Cache-Inhalt, den sie identifiziert, ändert sich: Er besteht nun aus den zu x und y korrespondierenden $sortNumber$-Attributwerten sn_x und sn_y, so dass Cache-Einträge nun die folgende Form haben:

$$\chi_{[x,y]} = (id_{[x,y]}, sn_x, sn_y) \quad \textbf{(F68)}$$

Diese hat gegenüber dem naiven Ansatz den Performanz-Nachteil, dass die finale SELECT-Operation, die die eigentliche Ergebnismenge der Suchanfrage zusammenstellt, bei der Abfrage des Cache-Eintrags erneut ausgeführt werden muss. Es lässt sich jedoch zeigen, dass bei einer Intervallsuchanfrage, wie sie in Kapitel 9.7.1.1 beschrieben wurde, der weitaus größte Teil der Antwortzeit für die Bearbeitung der beiden Indexbaum-Traversierungen aufgewendet werden muss und nur ein entsprechend kleiner Teil auf die finale SELECT-Operation. Es lohnt sich sicherlich oft, den Performanz-Nachteil dieser Form des Cachings in Kauf zu nehmen, so lange man auf die teureren cTree-Baumtraversierungen verzichten kann und gleichzeitig die Speicherung der umfangreichen Ergebnismengen vermeidet.

Darüber hinaus bietet sich noch eine Verfeinerung dieses Ansatzes an: Die beiden Indextraversierungen sind nicht voneinander abhängig, so dass auch ihre Ergebniswerte unabhängig voneinander als „Intervallhälften" in verschiedenen Intervallsuchen eingesetzt werden können: Der für x ermittelte $sortNumber$-Attributwert sn_x kann beispielsweise auch in einer Intervallsuche nach allen Werten in $[x, z]$ eingesetzt werden.

Daher ist es von Vorteil, die Cache-Einträge derart zu speichern, dass für ein Intervall $[x, y]$ jeweils ein Cache-Eintrag $\chi_{[x}$ für die obere und ein Eintrag $\chi_{y]}$ für die untere Intervallgrenze gespeichert wird. Cache-Einträge haben dann die Form

$$\chi_{[x} = (id_{[x}, sn_x) \text{ bzw. } \chi_{y]} = (id_{y]}, sn_y) \quad \textbf{(F69)}$$

Weiterhin umfasst das Identifikationsmerkmal $id_{[x}$ für eine Intervallhälfte „$[x$" nicht nur $Enc_{det}(closed)$ und $Enc_{det}(x)$, sondern auch die definit verschlüsselte Information $Enc_{det}(lower)$, dass es sich um eine untere Intervallgrenze handelt. Dies ist deshalb wichtig, weil es sich bei den zu den oberen und unteren Intervallgrenzen ermittelten $sortNumber$-Attributwerten jeweils um auf unterschiedliche Art und Weise ermittelten Approximationen zu den tatsächlichen Intervallgrenzen handelt (siehe Kapitel 6.8). Dementsprechend kann ein $sortNumber$-Attributwert, der für eine untere Intervallgrenze ermittelt wurde, nicht als $sortNumber$-Attributwert für eine obere Intervallgrenze eingesetzt werden.

Somit werden nach einer ersten Suche nach $[x, y]$ die folgenden Cache-Einträge gespeichert:

$$\chi_{[x} = \left(id_{[x}, sn_x\right) = (Enc_{det}(lower), Enc_{det}(closed), Enc_{det}(x), sn_x)$$
$$\chi_{y]} = \left(id_{y]}, sn_y\right) = (Enc_{det}(upper), Enc_{det}(closed), Enc_{det}(y), sn_y) \quad \textbf{(F70)}$$

9.7.1.3 Verwendung des serverseitigen Cachens

Das in 9.7.1.2 beschriebene Caching-Verfahren wird folgendermaßen eingesetzt:

- Ein Client, der eine Intervallsuche nach $[x, y]$ ausführen will, berechnet $id_{[x}$ und $id_{y]}$ und schickt beide Quadrupel an den Server.
- Der Application Server durchsucht den Cache nach Einträgen mit $id_{[x}$ bzw. $id_{y]}$.
- Bei einem Treffer sowohl für $id_{[x}$ als auch für $id_{y]}$ zieht der Server beide resultierenden $sortNumber$-Attributwerte für die finale SELECT-Operation heran.
- Wird für ein Identifikationsmerkmal kein Eintrag gefunden, so führen Client und Server die Indextraversierung wie in 6.8 beschrieben durch. Der Server speichert den daraus ermittelten $sortNumber$-Attributwert als neuen Eintrag mit dem entsprechenden Identifikationsmerkmal im Cache.

9.7.1.4 Löschstrategien

Die Evolution der Extension eines mit Caching versehenen Datenbestandes bedingt, dass Cache-Einträge ungültig werden und gelöscht werden müssen. Im vorliegenden Fall des serverseitigen Cachings kann dies zum einen der Fall sein, wenn dem Index ein Element hinzugefügt wird, das bezüglich einer Intervallgrenze ein besserer Approximationswert ist als der, den ein Cache-Eintrag für diese Intervallgrenze angibt. Zum anderen tritt dies ein, wenn ein Indexeintrag gelöscht wird, der in einem Cacheeintrag als Intervallgrenze referenziert wird. In diesem Fall muss der Cache-Eintrag gelöscht bzw. aktualisiert werden.

Ein naiver Ansatz hierfür ist, alle Cache-Einträge zu löschen, wann immer der Index sich ändert. Diese simple Vorgehensweise produziert zwar konsistente Cache-Zustände, arbeitet aber angesichts vieler unnötiger Löschungen sehr ineffizient. Der folgende simple Ansatz ermöglicht dagegen bereits eine effizientere Cache-Nutzung, die jedoch in ihrer Grundform lediglich für cTree-Indexe funktioniert, die keine Löschungen erlauben:

- Es werde ein $k > 0$ definiert, das eine maximale Präfixlänge definiert.
- Wird ein neuer Wert x zu x' verschlüsselt und in den Index eingefügt, so berechne und verschlüssele der Client darüber hinaus auch die Menge der 1- bis k-stelligen Präfixe von x, so dass die Menge $p_x = \{x'_1, \dots, x'_k\}$ entstehe. Der Client sende p_x zusammen mit x' an den Server.
- Nachdem x' eingefügt und ihm ein $sortNumber$-Attributwert $sn_{x'}$ zugeordnet wurde, prüfe der Server für alle $x'_i \in p_x$, ob bereits Cacheeinträge $\chi_{[x'_i}$ bzw. $\chi_{]x'_i}$ mit einem größeren $sortNumber$-Wert existieren. In diesem Fall werde so vorgegangen, dass
    - im Falle einer geschlossenen Intervallhälfte der $sortNumber$-Wert von $\chi_{[x'_i}$ mit $sn_{x'}$ überschrieben werde und
    - im Falle einer offenen Intervallhälfte der $sortNumber$-Wert von $\chi_{[x'_i}$ mit $sn_{x'}$ überschrieben werde, falls $x'_k \neq x'$.

- Weiterhin führe der Server den zum vorgenannten symmetrischen Schritt für etwaige Cache-Einträge $\chi_{x_i'|}$ bzw. $\chi_{x_i'[}$ durch.[84]

Das oben beschriebene Vorgehen führt zu einem stetig wachsenden Cache, der mit einer gängigen Löschstrategie verknüpft werden kann, etwa *least recently used* (*LRU*), so dass stets diejenigen Einträge gelöscht werden, deren letzte Verwendung am längsten zurückliegt.

9.7.1.5 Zweiter alternativer Ansatz: Serverseitiges Cachen der Teilbäume

Alternativ zum oben genannten Ansatz bietet es sich auf der Serverseite an, beim Abrufen von cTree-Teilbäumen letztere auf der Ebene des Application Servers zu cachen und so DBMS-Zugriffe einzusparen. Performante Lösungen zur Implementierung eines solchen Caches wie beispielsweise Infinispan[85] von Red Hat stehen für diese Aufgabe bereit.

Auch hier ist ein effizientes Löschkonzept erforderlich: Die Evolution der cTree-Extension verändert die Anordnung der Indexelemente in der Baumdarstellung und kann damit einzelne oder mehrere Cache-Einträge ungültig machen. In Kapitel 9.7.3.2, welches sich mit clientseitigen Caches befasst, wird ein solches Löschkonzept vorgestellt. Es kann leicht auf diese Form des serverseitigen Caches angepasst werden.

9.7.2 Clientseitiges Caching

In diesem Unterkapitel wird der Einsatz von Caching auf der Clientseite betrachtet. Dessen Grundidee ist, so viele überflüssige Teilbaumanfragen und -übertragungen wie möglich zu vermeiden, indem abgerufene Teilbäume, die für cTree-Traversierungen benötigt wurden, im clientseitigen Cache gespeichert werden. Wenn sie bei nachfolgenden Traversierungen des cTree-Index wieder benötigt werden, können sie aus diesem Cache abgerufen und anschließend traversiert werden, anstatt sie vom Server anfragen zu müssen. Hierbei ist zu beachten, dass bei Lese- und Schreiboperationen der gleiche Typ von Teilbäumen übertragen wird. Teilbäume im Cache, die aus Leseoperationen stammen, können also auch bei Schreiboperationen zum Einsatz kommen und umgekehrt.

τ_{root} beispielsweise, also derjenige Teilbaum, der direkt unter dem Wurzelknoten v_{root} von T hängt, wird bei Traversierungen stets benötigt, da jede Traversierung von T bei v_{root} startet. Andere clientseitig gespeicherten Teilbäume können ebenfalls eine hohe Wahrscheinlichkeit auf wiederholte Benötigung haben, etwa wenn der Client wiederholt nach nahe zueinander gelegenen Indexelementen sucht bzw. solche Elemente einfügt. Ein Beispiel für letzteres ist eine *bulk insert*-Operation, also eine große Anzahl von Einfügeoperationen, mit einem hohen Grad an Lokalität. Wenn sie etwa sortiert erfolgt, wählt sie immer wieder einen ähnlichen Pfad durch den binären Baum, weshalb eine hohe Wahrscheinlichkeit besteht, dass oft wiederholte Anfragen nach den selben Teilbäumen entstehen.

In der praktischen Implementierung dieses Ansatzes wird ein Teilbaum τ_i mit Höhe h und Wurzel v_i nach seinem Erhalt im Client als Cache-Eintrag χ_i gespeichert. χ_i enthalte die $k \leq 2^h - 1$ Knoten des Teilbaums τ_i und verwende die Knotennummer i seines Wurzelkno-

[84] Zwar werden bei diesem Ansatz nur verschlüsselte Präfixe x_i' zum Server geschickt, jedoch stellt die Korrelation von x' und p_x während des Einfüge- und Cache-Aktualisierungsschrittes einen Informationsverlust dar, den sich ein Angreifer zu Nutze machen kann. Um diesen zu vermeiden, müsste der Ansatz noch verfeinert werden, worauf an dieser Stelle aus Platzgründen verzichtet wird.
[85] siehe [URL-Infinispan].

tens als Identifikationsmerkmal. Es hat sich darüber hinaus als nützlich erwiesen, denjenigen Cache-Eintrag, der zu τ_{root} gehört, als solchen in einem $boolean$-Attribut $isRoot$ zu kennzeichnen. Damit hat ein Cache-Eintrag χ_i den folgenden Aufbau:

$$\chi_i = (i, nodes_i, isRoot), \text{ mit}$$

$$nodes_i = \{(seq_{i,j}, leftChild_{i,j}, rightChild_{i,j}, ixData_{i,j}) | 1 \leq j \leq k\} \quad \textbf{(F71)}$$

Ein Knoten bestehe clientseitig dabei, wie oben angegeben, aus seiner Nummer seq, den Nummern seines linken und rechten Kindknotens $leftChild$ und $rightChild$ und seinem verschlüsselten Inhalt $ixData$, welcher durch sein entschlüsseltes Pendant zu ersetzen ist. Dies kann so erfolgen, dass alle $ixData$-Instanzen in τ_i beim Anlegen von χ_i entschlüsselt werden, unabhängig davon, ob sie bei einer Traversierung benötigt werden oder nicht. Eine Alternative ist, jede $ixData$-Instanz erst bei Bedarf zu entschlüsseln, also wenn ihr Klartext während einer Traversierung von τ_i benötigt wird. Bei jeder erneuten Traversierung von τ_i müssen bereits entschlüsselte $ixData$-Instanzen nicht nochmals entschlüsselt werden.

Wird ein Teilbaum τ_i benötigt, prüft der Client zunächst, ob er einen entsprechenden Cache-Eintrag χ_i besitzt. Falls dem so ist, ruft der Client τ_i aus dem Cache ab; falls nicht, fordert er ihn vom Server an.

9.7.3 Löschstrategien für clientseitiges Caching

Für einen korrekten und die Performanz steigernden Einsatz von clientseitigem Caching ist eine effiziente Löschstrategie erforderlich. Sie sollte einerseits sicherstellen, dass der Cache korrekt ist, also keine Teilbaum-Einträge enthält, die durch Veränderung der cTree-Extension auf dem Server inkonsistent geworden sind. Andererseits sollen durch die Strategie so wenige unnötige Löschungen von Cache-Einträgen verursacht werden wie möglich.

Ohne Beschränkung der Allgemeinheit wird in diesem Unterkapitel zur Beschreibung der Löschstrategie eine ausschließlich wachsende Indexextension betrachtet. Die Löschstrategie lässt sich jedoch auch derart modifizieren, dass sie auf cTree-Indexe angewendet werden kann, die auch Löschungen erlauben.

9.7.3.1 Naiver Ansatz

Eine naive Löschstrategie sieht vor, dass jede Indexmodifikation den gesamten Cache lösche: Wird dem cTree-Index ein Element hinzugefügt, werden alle mit dem Server verbundenen Clients darüber benachrichtigt, woraufhin sie jeweils ihren lokalen Cache löschen.

9.7.3.2 Rotationspunkt-Ansatz

Der in Kapitel 9.7.3.1 genannte Ansatz mag jederzeit korrekte Cacheinhalte garantieren, verursacht jedoch zahlreiche unnötige Löschungen, wie im Folgenden gezeigt wird.

Ein neues Element v_{new}, das dem cTree-Index hinzugefügt wird, kann bewirken, dass der AVL-Baum T, der die lineare Ordnung der indexierten Elemente enthält, rebalanciert werden muss. Eine Rebalancierung wird so durchgeführt, dass der Server beim neu eingefügten Blattknoten v_{new} über die $parent$-Verknüpfung den Traversierungspfad durch T von v_{new} bis zu v_{root} entlang zurück läuft: dieser Vorgang wird im Folgenden als *Rücktraversierung* bezeichnet. Bei jedem Knoten v_i auf diesem Pfad wird dessen Balanciertheit geprüft, also ob sich die Höhe der beiden Teilbäume unter v_i um maximal eins unterscheidet.

Im Falle einer Dysbalance führt der Server, ohne Mithilfe des Clients, unterschiedliche Rebalancierungsoperationen (*Rotationen*, siehe Kapitel 3.4) aus, je nach Typ der Dysbalance (siehe Kapitel 3.4). Der Knoten, der die Rotation ausgelöst hat, werde im Folgenden *Rotationspunkt* genannt. Eine Rotation verändert die Positionen, Kind-Eltern-Relationen und Teilbaumhöhen mehrerer Knoten in der Nachbarschaft des Rotationspunkts.

Für AVL-Bäume, die nur Einfügeoperationen erlauben, ist bei einer Rücktraversierung höchstens eine Rotation erforderlich. Die Rücktraversierung kann also abgebrochen werden, sobald eine Rotation aufgetreten ist.[86] Darüber hinaus ist es eine intrinsische Eigenschaft des AVL-Baums, dass die Balance aller Knoten, die außerhalb des (Rück-)Traversierungspfads liegen, nicht beeinträchtigt wird.

Das Prinzip der Rotationen auf dem Rücktraversierungspfad ist nun mit der Konsistenz der clientseitig gecacheten Teilbäume in Einklang zu bringen. Abb. 9 zeigt beispielhaft eine Reihe von Cache-Einträgen χ_i, die mit Teilbäumen τ_i korrespondieren, welche zusammen einen cTree-Index-Baum T darstellen. In Abb. 9 sind die Cache-Einträge χ_{1741}, χ_{2824}, χ_{10}, χ_{1053}, χ_{916} und χ_{2549} zu sehen (somit ist v_{1741} der Wurzelknoten v_{root} von T). Die Teilbäume sind derart miteinander verknüpft, dass die Wurzel eines Teilbaums von der Blattebene eines anderen Teilbaums als Kindknoten referenziert wird, so wie etwa v_{916} von der Blattebene von τ_{2824} aus.

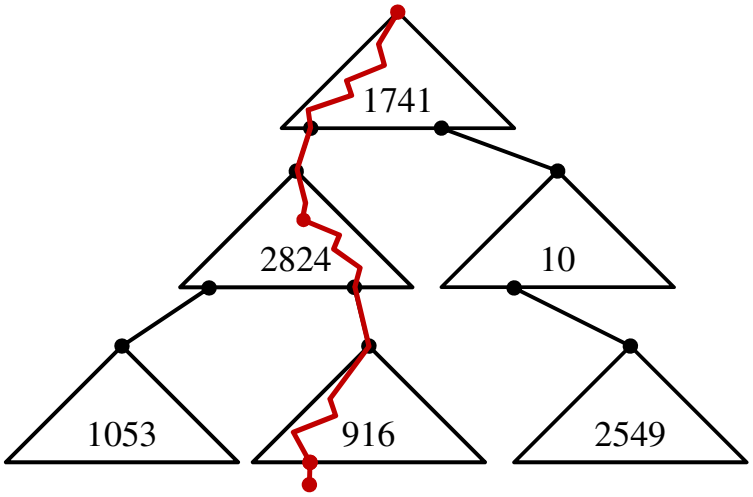

Abb. 9: Traversierungspfad nach Einfügung mit Rotation innerhalb von χ_{2824}.

Abb. 9 zeigt einen Traversierungspfad durch die Gesamtheit der verknüpften Teilbäume, den man durchlaufen muss, um einen neuen Knoten v_{new} auf der Blattebene von τ_{916} einzufügen. Bei der anschließenden Rebalancierung und der damit einhergehenden Rücktraversierung zu v_{root} durchläuft der Rücktraversierungspfad lediglich die Teilbäume τ_{916}, τ_{2824} und τ_{1741}. Die anderen Teilbäume τ_{1053}, τ_{2549} und τ_{10} werden nicht von der Rücktraversierung berührt und können daher unangetastet im Cache verbleiben.

Nimmt man an, dass die Einfügung von v_{new} genau eine Rotation in τ_{2824} verursacht, beeinträchtigt diese zunächst lediglich τ_{2824}, bzw. den zugehörigen Cache-Eintrag χ_{2824} in einem

[86] Für AVL-Bäume, die auch Löschungen erlauben, gilt dies hingegen nicht, was die Löschoperation etwas verkompliziert, aber nicht prinzipiell verändert. Deshalb wird an dieser Stelle lediglich der einfachere Fall berücksichtigt, in dem ausschließlich Einfügeoperationen in den AVL-Baum erlaubt sind.

Client-Cache. χ_{2824} muss aus diesem Cache gelöscht werden, da er nicht mehr mit dem Zustand von T konsistent ist. Die anderen Cache-Einträge auf dem Pfad, χ_{1741} und χ_{916}, bleiben von der Rebalancierung unangetastet, und man könnte annehmen, dass sie daher im clientseitigen Cache verbleiben können.

9.7.3.3 Vermeidung „toter" Cache-Einträge

Tatsächlich ist aber die Löschung weiterer Cache-Einträge erforderlich: Eine Rotation in einem Teilbaum τ_{i_0} stört zwar nicht die Konsistenz seiner (transitiven) Kind-Teilbäume $\tau_{i_1}, \ldots, \tau_{i_n}$; jedoch kann sie verändern, auf welcher Ebene von T sich deren jeweilige Wurzelknoten $v_{i_1}, \ldots, v_{i_n}$ befinden, was diese Teilbäume und die damit korrespondierenden Cache-Einträge $\chi_{i_1}, \ldots, \chi_{i_n}$ unerreichbar und damit nutzlos, in Sonderfällen sogar fehlerhaft werden ließe, wie anhand des in Abb. 9 illustrierten Beispiels verdeutlicht wird. Dies wird im Folgenden anhand eines Beispiels verdeutlicht:

Sei $\gamma_{\mathrm{T}}(v_i)$ eine Funktion, die die Ebene eines Knotens v_i im AVL-Baum T angibt. Im Beispiel in Abb. 9 befinden sich unter der Annahme einer fixen Teilbaumhöhe $h = 5$ die Wurzelknoten v_{1053} und v_{916} der Teilbäume τ_{1053} und τ_{916} vor der Rotation in τ_{2824} jeweils auf der 11. Ebene von T, d. h., $\gamma_{\mathrm{T}}(v_{1053}) = \gamma_{\mathrm{T}}(v_{916}) = 11$. Durch die Rotation können sich $\gamma_{\mathrm{T}}(v_{1053})$ und $\gamma_{\mathrm{T}}(v_{916})$ jedoch zu 10 und 12 ändern, bzw. zu 12 und 10. Dann sind v_{1053} und v_{916} bei einer Baumtraversierung nicht mehr erreichbar, denn bei einer Teilbaumhöhe $h = 5$ müssen die Funktionswerte $\gamma_{\mathrm{T}}(v_i)$ aller Knoten v_i, die als Wurzelknoten für Cache-Einträge χ_i fungieren, die Gleichung

$$\gamma_{\mathrm{T}}(v_i) = 1 + k \cdot h, k \in \mathbb{N} \qquad \textbf{(F72)}$$

erfüllen. Anders ausgedrückt müssen sich bei $h = 5$ die Wurzelknoten von clientseitig gecacheten Teilbäumen, die bei einer Traversierung von T herangezogen werden sollen, auf den Baumebenen 1, 6, 11, 16, etc. aufhalten. Anderenfalls werden sie stets übergangen, verbleiben als unerreichbare Einträge im Cache und werden aufgrund der Evolution der Baumextension wahrscheinlich irgendwann inkonsistent zum tatsächlichen Zustand von T.[87]

Aus diesem Grund würden in einer alternativen Löschstrategie im Fall eines durch Rotation ungültig gewordenen Cache-Eintrags χ_i alle weiteren Cache-Einträge $\chi_{j_1}, \ldots, \chi_{j_n}$ ebenfalls als ungültig deklariert, die zu den transitiven Kind-Teilbäumen von τ_i gehören. Im in Abb. 9 angegebenen Beispiel beträfe dies neben χ_{2824} die Cache-Einträge χ_{1053} und χ_{916}. Diejenigen Teilbäume, für die der Rotationspunkt weder im direkten noch in einem transitiven Elternbaum liegt, könnten dagegen im Cache verbleiben.

9.7.3.4 Benachrichtigung anderer Clients

In einem Verbund von mehreren Clientrechnern $C_1, \ldots, C_n$, die alle auf der Extension eines zu einem cTree-Index gehörenden AVL-Baums T mit einer einheitlichen Teilbaumhöhe h arbeiten, müssen die Clients über Änderungen auf T benachrichtigt werden. Führt ein Client C_k eine Änderung an T durch, durch die ein Cache-Eintrag χ_{i_0} ungültig wird (und mit ihm entsprechend zu Kapitel 9.7.3.3 weitere Einträge $\chi_{i_1}, \ldots, \chi_{i_n}$), so müssen alle anderen Clients hiervon benachrichtigt werden und diese Einträge aus ihren eigenen Caches löschen.

---

[87] Es ist sogar der Fall möglich, dass der Wurzelknoten eines solchen unerreichbaren Cache-Eintrags χ_i nach einer kompensierenden Rotation wieder erreichbar wird, mittlerweile aber inkonsistent geworden ist und deshalb fehlerhafte Traversierungsergebnisse auf dem Client produziert.

9.8 Transaktionales Konzept

9.8.1 Motivation

Das Standard-Anwendungsszenario für die in dieser Dissertation eingeführten Indexstrukturen ist ein in einer Dreischicht-Architektur betriebenes, webbasiertes Informationssystem. Eine Vielzahl von Clients greift über das Internet auf den Applikationsserver und dieser wiederum auf das DBMS zu, wobei letztere beiden in einem entfernten Rechenzentrum gehostet werden. Weitere Aspekte dieses Standard-Anwendungsszenarios seien die Eigenschaft, dass von Clients erheblich mehr lesende als schreibende Zugriffe an den Server abgesetzt werden, sowie dass Client-Verbindungen oft ungeplant terminiert werden, beispielsweise durch Verbindungsabbrüche beim Einsatz mobiler Clients. Weiterhin sollen hohe Anforderungen an die Integrität der Daten gestellt werden, so dass jede Änderung an einem Datum bei jedem anschließenden Zugriff darauf unverzüglich reflektiert werde. In einer Testimplementierung (siehe Kapitel 11) dieser Architektur kommunizieren HTML5/Javascript-Clients über stehende Websocket-Verbindungen mit einem JBoss-Middleware-Server, der mit einem PostgreSQL-DBMS kommuniziert.

Um den Verfügbarkeitsanforderungen eines Informationssystems mit einer solchen Architektur gerecht zu werden, ist es ratsam, keine clientgesteuerten Transaktionen zuzulassen, da sie bei einem Verbindungsabbruch schwierig zu terminieren sind. Zusammen mit den oben formulierten Anforderungen an die Integrität der Daten erscheint weder die Verwendung des im Kontext relationaler Datenbanken gängigen ACID-Paradigmas[88] in clientgesteuerten Transaktionen noch die Anwendung des in modernen Informationssystemen oft zum Einsatz kommenden BASE-Prinzips[89] sinnvoll.

Insbesondere in Bezug auf die jeweils aus mehreren Schritten bestehende Traversierung, Manipulation und Balancierung des cTree-Indexes erschienen verschiedene Aspekte der eingesetzten DBMS-Infrastruktur als unzureichend: Das Locking-Konzept des PostgreSQL-DBMS erwies sich als zu grob bei konkurrierenden Clients, die auf die selbe cTree-Indexrelation zugreifen: Entweder wurde hier für den unterlegenen zweier konkurrierender Clients gleich die ganze Extension der Indexrelation gesperrt (der *ACCESS EXCLUSIVE*-Locking-Modus in PostgreSQL-Terminologie), oder ihm wurde der bisherige, nicht mehr gültige Extensionszustand präsentiert (*ACCESS SHARE*)[90]. Letzteres folgt der *Multiversion Concurrency Control*-Ansatz (*MVCC*), der dem PostgreSQL-DBMS zugrunde liegt.

Sowohl ACCESS EXCLUSIVE als auch MVCC erwiesen sich angesichts der Anforderungen an die Verwendung des cTree-Indexes als unzureichend: Sperrung nach ACCESS EXCLUSIVE stellt zwar Zugriffe auf einen stets korrekten Datenbestand sicher, bewirkt aber auch eine signifikante und in großen Teilen unnötige Minderung der Verfügbarkeit und Performanz des Systems. Der MVCC-Ansatz hingegen ermöglicht zwar ein Arbeiten ohne Performanzeinbußen, jedoch ist das Arbeiten auf nicht aktuellen Daten- bzw. Indexdatenbeständen nach den gestellten Integritätsanforderungen nicht akzeptabel.

[88] siehe [Kemper2011], S. 289.
[89] siehe [Edlich2011] , S. 33.
[90] siehe [URL-PostgreSQL].

9.8.2 Versionierung

Um den in Kapitel 9.8.1 beschriebenen Rahmenbedingungen Rechnung zu tragen, wurde für den Einsatz des skizzierten webbasierten Informationssystems ein simples, optimistisches, der lazy-operating-Philosophie folgendes leichtgewichtiges Transaktionsmodell auf Basis von Versionsnummern entwickelt, welches in diesem Unterkapitel vorgestellt wird.

Danach wird jedem eingesetzten cTree-Index ix in der Indexverwaltung (siehe Kapitel 4.2) eine ganzzahlige Versionsnummer k_{ix} zugewiesen, die den gegenwärtigen Zustand von ix dokumentiert. Jeder schreibende und/oder lesende Clientprozess ρ, der im Rahmen einer aus mehreren Schritten bestehenden cTree-Operation auf ix zugreift (d. h., eine Indextraversierung durchführt oder ein neues Indexelement einfügt), ruft als ersten Schritt k_{ix} vom Server ab. Anschließend wird k_{ix} bei jeder folgenden, zur Transaktion gehörenden Serveranfrage mit an den Server geschickt. Dieser Wert repräsentiert den Zustand von ix, von dem der Client ausgeht.

Der Server überprüft zu Beginn einer jeden erhaltenen Clientanfrage, ob das erhaltene k_{ix} noch dem tatsächlichen Zustand von ix, d. h. dem serverseitig gespeicherten k_{ix}, entspricht. Falls nicht, benachrichtigt der Server den Client diesbezüglich, woraufhin dieser seine Operation abbricht und anschließend gegebenenfalls erneut beginnt. Stimmt hingegen das vom Client gesendete k_{ix} bei allen Transaktionsschritten mit seinem serverseitigen Pendant überein, so inkrementiert der Server bei einer schreibenden Transaktion den k_{ix}-Wert, nachdem der letzte Schritt der Transaktion erfolgreich terminiert hat.

In den beiden folgenden Abbildungen wird das oben beschriebene Transaktionsmodell anhand der Beispiele einer einzelnen, ungestört terminierenden Einfügeoperation (siehe Abb. 10) sowie des gescheiterten Einfügens eines neuen Indexelements (siehe Abb. 11) illustriert.

In Abb. 10 beginnt ein Client eine Transaktion, bei der ein cTree-Index ix traversiert wird, um darin die passende Einfügeposition für ein neues Indexelement zu ermitteln. Die Indexversion $k_{ix} = 763$ wird ermittelt und zusammen mit der Knotenmenge an den Server zurückgegeben. Bei der folgenden Anfrage (nach dem nächsten Teilbaum aus dem Index) übermittelt der Client auch das zuvor erhaltene k_{ix}, wie auch bei der finalen Einfügeoperation. Der Server prüft das vom Client erhaltene k_{ix} gegen dessen serverseitig gespeicherten Wert und führt den Einfügevorgang planmäßig aus. Abschließend inkrementiert er k_{ix} und sendet dessen neuen Wert 764 an den Client zurück.

In Abb. 11 beginnen zwei schreibende Clientprozesse ρ_1 (in rot) und ρ_2 (in blau) jeweils eine Transaktion mit der Anfrage des obersten Index-Teilbaums τ_{root} von ix, bei der auch die Indexversion $k_{ix} = 764$ angefragt wird. Bei jedem folgenden Transaktionsschritt senden beide Prozesse ihr erhaltenes k_{ix} zum Server, bis ρ_1 seine Transaktion mit der finalen Einfügeoperation eines Indexelements abschließt, was eine Inkrementierung des serverseitig gespeicherten Wertes von k_{ix} zur Folge hat. Wenn nun ρ_2 ebenfalls versucht, den finalen Schritt seiner Transaktion beim Server anzuweisen, registriert der Server den nun nicht mehr aktuellen Wert für k_{ix}, führt die angewiesene Einfügung des neuen Indexelements mit anschließender Baumbalancierung nicht aus und benachrichtigt den entsprechenden Client diesbezüglich. Der benachrichtigte Clientprozess ρ_2 kann daraufhin beschließen, seine Transaktion erneut zu beginnen.

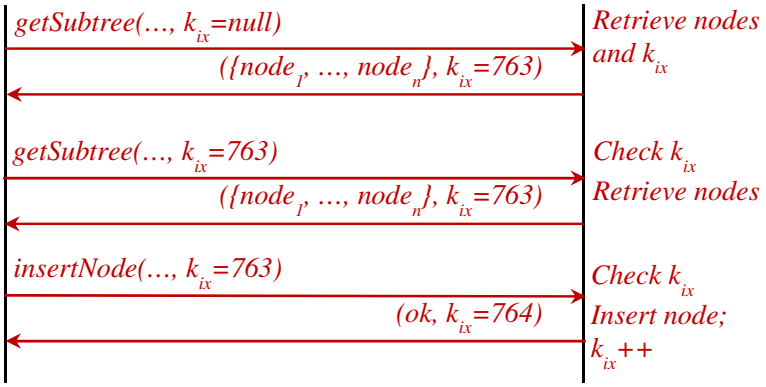

Abb. 10: Client/Server-Kommunikation bei einer einzelnen Transaktionen.

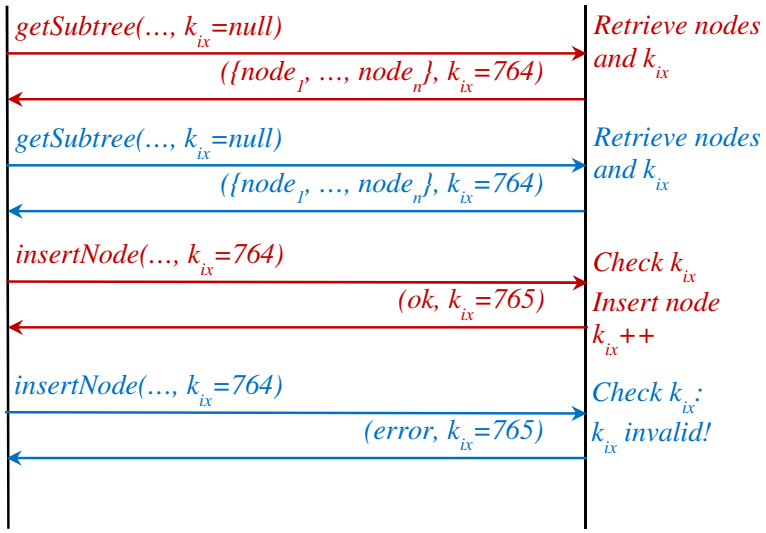

Abb. 11: Client/Server-Kommunikation bei zwei konkurrierenden Transaktionen.

Anmerkung: Um die Eigenschaft der Atomarität einer Transaktion zu erlangen, müssen alle Änderungen, falls nötig, zurückgerollt werden. Bei der Einfügung eines neuen Elements in den Index ist die einzige Operation, die Änderungen auf dem Index verursacht, die letzte Operation der Transaktion (*insertNode* in Abb. 10 und Abb. 11). In der praktischen Implementierung wird diese in einer stored procedure ausgeführt und demnach automatisch zurückgerollt, falls darin ein Fehler auftritt. Somit ist die Eigenschaft der Atomarität erfüllt.

9.8.3 Nebenläufigkeitseffekte bei Bulk Insert-Operationen

Bei sogenannten „Bulk Insert"-Operationen, also Sequenzen von zahlreichen aufeinanderfolgenden Einfügeoperationen in den cTree-Index, weist das in Kapitel 9.8.2 beschriebene Transaktionsmodell eine Eigenschaft auf, die Verfügbarkeitsprobleme für den Index verursachen kann. Abhilfe gegen dieses unerwünschte Verhalten schafft eine zusätzliche Kontrollstruktur, die in diesem Kapitel behandelt wird.

Wie in Kapitel 9.8.2 beschrieben wird jeder auf einen cTree-Index ix zugreifende Clientprozess aufgehoben, dessen Kopie der Indexversionsnummer k_{ix}, die er zu Beginn seiner Transaktion vom Server abgerufen hat, sich bei einem der folgenden Transaktionsschritte als veraltet herausstellt. Da nur schreibende Clientprozesse den Wert von k_{ix} verändern können, bedeutet dies, dass jeder schreibende Prozess jeden lesenden Prozess stets aufheben kann.

Im Falle einer Bulk Insert-Operation eines Clients ist es daher für andere, rein lesende Clientprozesse damit schwer zu terminieren, da sie stets von anderen, schreibenden Prozessen aufgehoben werden. Auch schreibende, mit dem Bulk Insert konkurrierende Clientprozesse werden oft so lange behindert, bis die Bulk Insert-Operation terminiert hat.

Auch wenn dieser Effekt nicht vollständig behoben werden kann, ohne das Transaktionsmodell aus 9.8.2 erheblich zu verkomplizieren, kann er doch deutlich abgemildert werden, indem Clientprozessen die Möglichkeit eingeräumt wird, ein spezielles „override“-Signal an den Server zu senden, mit dem sie die Durchführung des eigenen Anliegens erzwingen können. Stellt ein Clientprozess etwa fest, dass die eigene Transaktion n-mal hintereinander abgewiesen wurde, sendet er das Signal an den Server, der daraufhin ein spezielles Flag setzt, das besagt, dass alle schreibenden Clientprozesse vorübergehend ausgesetzt werden sollen. Nachdem der lesende Clientprozess terminiert hat, erhält der Server ein weiteres Signal, dass dieser das Flag zurücksetzen kann, woraufhin das Informationssystem in seinen normalen Zustand zurückwechselt.

10 Sicherheitsaspekte

Alle in den Kapiteln 5 bis 8 vorgestellten Indexierungstechniken weisen die Charakteristik auf, dass die aus der Indexierung entstehenden Daten verschlüsselt sind, genauso wie die indexierten Daten selbst. Gleichzeitig exponieren die Indexdaten jedoch bei allen Techniken Klartextinformation über sich selbst bzw. über ihre Struktur. Der Server verwendet diese Klartextinformation für den beschleunigten Zugriff auf die Indexdaten, unter Einbeziehung von herkömmlichen Datenbankindexen, so dass er in der Lage ist, Benutzeranfragen auf den indexierten Daten beschleunigt auszuführen.

Diese Preisgabe von unverschlüsselten Teilinformationen der Indexdaten geschieht unter dem Paradigma, dass Klartextinformation stets lediglich über die Struktur der Indexdaten bereitgestellt wird, nicht aber über deren Inhalt[91]. Dennoch ist dieser Aspekt nicht zu vernachlässigen, wie etwa die in [Kamara2015] beschriebenen Angriffe (siehe Kapitel 2) belegen: Auch dort wurden die Angriffe auf definit bzw. OPE-verschlüsselten Daten durchgeführt, durch die große Teile des Klartexts der verschlüsselten Daten inferiert werden konnten. In den folgenden Unterkapiteln werden daher die verschiedenen Typen von preisgegebener Klartextinformation sowie die sich daraus ergebenden Angriffsmöglichkeiten auf die Vertraulichkeit der Indexdaten (und damit auch der indexierten Daten) beschrieben. Schließlich beschreibt ein weiteres Unterkapitel Abwehrstrategien, die den Informationsverlust unter Inkaufnahme von Performanzverlust vermeidet, bzw. erheblich verringert.

10.1 Exponierte Klartextinformation auf der Serverseite

Im Folgenden wird beschrieben, auf welche Art die in den Kapiteln 5 bis 8 vorgestellten Indexierungstechniken Klartextinformation auf dem Server preisgeben. Dazu zählen sowohl bewusst exponierte Information als auch unbeabsichtigter Informationsverlust.

10.1.1 Gleichheitssuche

Bei der Gleichheitssuche auf Attributen mit diskreten numerischen, binären und Zeichenketten-Datentypen (siehe Kapitel 5.3, 5.4 und 5.5) werden entweder die Werte aus der indexierten Datenrelation oder deterministisch davon abgeleitete Funktionswerte definit verschlüsselt, so dass gleiche Eingabewerte in gleichen Kryptotexten resultieren. Befindet sich in der Datenrelation n-mal der gleiche Wert, so enthält die Datenrelation genau so oft den entsprechenden Kryptotext, was sich durch Zählungen auf der Extension der Indexrelation leicht ermitteln lässt. Entsprechend lassen sich charakteristische Häufungen von gleichen Werteinstanzen über die gesamte Extension der Indexrelation feststellen.

Bei der Gleichheitssuche auf Gleitkommazahl-Attributen (siehe Kapitel 5.6.3) wird die Gleichheit zweier Werte x und y entsprechend einer „ε-Definition“ darüber definiert, dass ihr Abstand einen definierten Schwellwert ε unterschreitet: $|x - y| < \varepsilon$. Um dies mit verschlüsselten, auf dem Server auswertbaren Daten realisieren zu können, werden zur Indexierung eines Gleitkommawertes x folgende Werte in der Indexrelation gespeichert:

- Der definit verschlüsselte Wert $x_{low} = \left\lfloor \frac{x}{\varepsilon} \right\rfloor \varepsilon$, also das größte ganzzahlige Vielfache von ε, das kleiner oder gleich x ist.

[91] Dies ist etwa im cTree-Index der Fall, der die Ordnung der indexierten Daten preisgibt, sonst aber nichts (siehe Kapitel 2).

- Der dazu komplementäre, definit verschlüsselte Wert $x_{high} = \left(\left\lfloor \frac{x}{\varepsilon} \right\rfloor + 1\right)\varepsilon$, also das kleinste ganzzahlige Vielfache von ε, das echt größer ist als x. Zusammen mit x_{low} bildet x_{high} das sogenannte ε-Intervall von x.
- Der Wert $x_{rel} = x - \left\lfloor \frac{x}{\varepsilon} \right\rfloor \varepsilon \in [0, \varepsilon[$ im Klartext, also eine Angabe über die Position von x im ε-Intervall $[x_{low}, x_{high}[$.

Auch bei dieser Form von verschlüsselter Indizierung für die Gleichheitssuche lassen sich durch die beiden definit verschlüsselten Werte x_{low} und x_{high} Häufungen von Werten in der Extension der Indexrelation erkennen, die nach Kapitel 5.6.3 als gleich anzusehen sind.

10.1.2 Lineare Ordnung der cTree-Indexstruktur

In der für Intervall- und Präfixsuche eingesetzten cTree-Indexdatenstruktur (siehe Kapitel 6) wird die lineare Ordnung der im Index gespeicherten Werte in zwei verschiedenen Repräsentationen unverschlüsselt auf dem Server nachgebildet.

Die erste Repräsentation ist ein binärer Baum, dessen Knoten die Indextupel bzw. ihre entschlüsselten $ixData$-Attributwerte repräsentieren. Die Knoten referenzieren weitere Knoten als Eltern- und Kindknoten, wobei der $ixData$-Attributwert eines Knotens stets größer ist als der des linken Kindknotens und kleiner als der des rechten Kindknotens.

Die zweite Repräsentation der linearen Ordnung ist eine lineare Liste, bei der die Extension eines numerischen Attributs $sortNumber$ derart mit Werten gefüllt ist, dass sich daraus die gleiche lineare Ordnung der Indexelemente ergibt wie aus den entschlüsselten $ixData$-Attributwerten der Indexelemente.

10.1.3 M:N-Relationen

Die in Kapitel 4.2.4 eingeführten M:N-Relationen verbinden Tupel einer indexierten Datenrelation mit den entsprechenden Tupeln der Indexrelation. Dies gilt sowohl für die Gleichheitssuche wie auch für die Intervall- bzw. Präfixsuche unter Verwendung des cTree-Indexes. Dadurch können zwei oder mehrere unterschiedliche Attributwerteinstanzen der Datenrelation, die hinsichtlich der vorherrschenden Implementierung semantischer Gleichheit den gleichen Wert verschlüsseln, mit dem selben Indextupel verknüpft werden und sind dadurch als semantisch gleich erkennbar.

10.1.4 M:N-Relationen in der „shrinking window"-Erweiterung

In der „shrinking Window"-Erweiterung des cTree-Ansatzes (siehe Kapitel 7) wird die bereits in Kapitel 10.1.3 genannte M:N-Relation um die Klartextattribute $offset$ und $length$ vom Typ $integer$ erweitert, welche, je nach Ausbaustufe, die Wortlängen der Indexwerte unverschlüsselt beinhalten oder zumindest Hinweise darauf geben.

10.1.5 Volltextindex

In der in Kapitel 8 vorgestellten Anwendung der Indexierungsmechanismen aus den Kapiteln 5, 6 und 7 für einen verschlüsselten Volltextindex verknüpft eine M:N-Relation jedes Token einer indizierten Werteinstanz über mindestens ein M:N-Tupel mit Tupeln der Indexrelation. Je nachdem, welche Indexierungsmethode dabei verwendet wurde (siehe Kapitel 8.2), können alle Formen von Informationsverlust auftreten, die in den Unterkapiteln 10.1.1 bis 10.1.4 beschrieben wurden.

Überdies ist ein indexiertes Tupel der Datenrelation über die M:N-Relation mit unterschiedlichen Tupeln der Indexrelation für jedes im indexierten Tupel enthaltene Token verknüpft. Dies enthüllt unter anderem Informationen über die Anzahl der in der Freitext-Werteinstanz enthaltenen Tokens, was am folgenden Beispiel illustriert wird:

Sei in Relation r das Attribut a', welches verschlüsselte Freitext-Zeichenketten enthält, mit einem Volltextindex belegt. Dabei sei die in Kapitel 8.1 behandelte Tokenisierung durch den naiven Ansatz aus Kapitel 8.1.2 implementiert und die Indexierung als Suche auf Gleichheit (siehe Kapitel 8.2.2.1) mit einer simplen Normalisierung nach Kleinbuchstaben der eingegebenen Tokens. Eine M:N-Relation mn verbinde r mit der Indexrelation ix.

Die Werteinstanz $x =$„Man liebt sie oder man hasst sie“ werde verschlüsselt und als Attributwert von a' für ein Tupel $t_{r.1}$ in r eingefügt. Die Tokenisierung und einhergehende Duplikateliminierung von x liefere die fünf Tokens „man“, „liebt, „sie“, „oder“ und „hasst“, welche als Tupel $t_{ix.1}$ bis $t_{ix.5}$ in ix eingefügt werden. Diese Tupel werden über mn mit $t_{r.1}$ verknüpft. Dann wird bezüglich $t_{r.1}$ die Klartextinformation auf dem Server exponiert, dass der Wert von a' in $t_{r.1}$ fünf verschiedene Tokens, also in diesem Fall Wörter, enthält.

Auch wenn die exakte Anzahl von in x enthaltenen Wörtern, nämlich sieben, durch die Duplikateliminierung verschleiert wurde, kann durch eine simple Analyse der Extensionen von Index-, M:N- und Datenrelation bereits eine Annäherung an diese Anzahl erreicht werden.

10.2 Statische Angriffsmöglichkeiten

Durch die im vorangegangenen Unterkapitel beschriebenen Formen von Informationsverlust kann ein Informationssystem Ziel von Angriffen auf die Vertraulichkeit der im System gespeicherten Daten werden. In diesem und dem folgenden Unterkapitel werden einige solche Angriffe beschrieben, bevor in Kapitel 10.4 Gegenmaßnahmen vorgestellt werden, die die genannten Angriffe verhindern oder zumindest signifikant abschwächen.

Dabei wird zwischen statischen Angriffen, die eine Analyse des im System gespeicherten Datenbestandes vornehmen, und dynamischen Angriffen unterschieden, bei welchen das zeitliche Benutzerverhalten analysiert wird. Statische Angriffe werden in diesem Unterkapitel behandelt, bevor sich Unterkapitel 10.3 mit dynamischen Angriffsmöglichkeiten befasst. Der Fokus liegt dabei auf den statischen Angriffen; auch die in Kapitel 10.4 beschriebenen Gegenmaßnahmen beziehen sich in erster Linie auf statische Angriffsszenarien.

10.2.1 Häufigkeitsanalysen

Eine der simpelsten und naheliegendsten Formen der Kryptoanalyse ist die Häufigkeitsanalyse. Dabei nutzt ein Angreifer bekannte statistische Eigenschaften von Klartextelementen wie beispielsweise Buchstaben, Wörtern oder anderem aus, um anhand von ermittelten Häufigkeiten von wiederkehrenden Kryptotextfragmenten, unter Umständen auch sukzessive, Rückschlüsse auf korrespondierende Klartextfragmente zu ziehen.

So permutiere etwa eine monoalphabetische Verschlüsselung die Buchstaben des zugrundeliegenden Alphabets, so dass der in der deutschen Sprache mit 17,4% am häufigsten vorkommende Buchstabe „E“[92] beispielsweise auf „Q“ abgebildet werde, der mit 9,8% zweithäufigste Buchstabe „N“ auf „E“ usw. Ein Angreifer ermittelt in einem längeren Kryptotext

[92] siehe [URL-Frequencies-1].

die relativen Häufigkeiten der jeweiligen Buchstaben und stellt bei den prominenten Vertretern der Verteilung jeweils eine mit hoher Wahrscheinlichkeit korrekte Korrelation her. Weiterhin durchsucht der Angreifer aufbauend auf einer initialen Menge von wahrscheinlich korrekt assoziierten Klartext- und Kryptobuchstaben den Kryptotext nach den häufigsten Trigrammen, welche mit hoher Wahrscheinlichkeit mit den in der deutschen Sprache am häufigsten vorkommenden Wörtern „der", „die" und „und"[93] korrespondieren. Der Angreifer fährt sukzessive fort, bis er die Zuordnungsvorschrift der Permutationsfunktion vervollständigt hat.

Das obige Beispiel besitzt heute keine Relevanz mehr, da solche symmetrischen Verschlüsselungsverfahren längst durch anspruchsvollere Verfahren wie AES ersetzt worden sind. Dennoch illustriert es die Problematik, mit der man sich beim Einsatz von definiter Verschlüsselung, wie dies bei den Indexen für die Suche auf Gleichheit der Fall ist, auseinandersetzen muss. Es sei ein weiteres Beispiel genannt:

Sei $icd10'$ ein beliebig verschlüsseltes Attribut einer Datenrelation $diagnoses$, deren Extension die von einem Arzt gestellten Diagnosen für die von ihm behandelten Patienten enthalte. Die Extension von $icd10'$ enthalte die Kodierungen der gestellten Diagnosen entsprechend des weltweit etablierten und frei verfügbaren ICD10-Standards[94]. $icd10'$ sei weiterhin mit einem Index für Gleichheitssuche belegt, welcher in der Indexrelation $ix^{equality}_{diagnoses.icd10'}$ gespeichert sei. Die Indexrelation sei mit einer M:N-Relation $mn^{equality}_{diagnoses.icd10'}$ mit $diagnoses$ verbunden und enthalte ein definit verschlüsseltes und mit einem konventionellen Datenbankindex schnell durchsuchbares, binäres Attribut $icd10'_{ix}$. Sei weiterhin eine Wahrscheinlichkeitsverteilung darüber bekannt, welche Diagnosen von Allgemeinmedizinern in Form der entsprechenden ICD-10-Codes zu welcher Jahreszeit am häufigsten gestellt werden.

Dann kann sich ein Angreifer die definit verschlüsselten Indexwerte in $ix^{equality}_{diagnoses.icd10'}$ in der Form zu nutze machen, dass sich bei einer genügend großen Extension von $ix^{equality}_{diagnoses.icd10'}$ die der oben genannten Wahrscheinlichkeitsverteilung entsprechenden Häufigkeiten in der Extension herausbilden und der Angreifer somit Tupel identifizieren kann, die mit hoher Wahrscheinlichkeit bestimmten Klartexten zugeordnet werden können. Beispielsweise kann der Angreifer mit dem Wissen, dass der im Winter am häufigsten eingestellte ICD-10-Code „J00" lautet (was für „akute Rhinopharyngitis" steht, also Erkältungsschnupfen), den am häufigsten in $ix^{equality}_{diagnoses.icd10'}$ vorkommenden Indexwert mit einer relativ hohen Wahrscheinlichkeit genau dieser Diagnose zuordnen.

10.2.2 Mehrdimensionale Strukturen durch M:N-Relationen

Die in Kapitel 10.2.1 beschriebene Angriffsmöglichkeit der statistischen Häufigkeitsanalyse lässt sich durch die Verknüpfung von $ix^{equality}_{diagnoses.icd10'}$ mit der Datenrelation $diagnoses$ noch erweitern. Falls die Datenrelation mehrere Attribute beinhaltet, die jeweils mit einem Index für Gleichheitssuche versehen sind, kann die jeweilige in der Indexrelation offenbarte Information über die entsprechende M:N-Relation mit den Tupeln der Datenrelation verknüpft werden, so dass dort alle verschiedenen Informationen zusammengeführt werden können.

[93] siehe [URL-Frequencies-2].

[94] siehe [URL-ICD-10] .

Sollte es im vorgenannten Beispiel etwa möglich sein, dass dem selben Arztbesuch desselben Patienten mehrere Diagnosen zugeordnet werden können, so kann zusätzlich zum öffentlich zugänglichen Wissen, welche Diagnosen zur Winterzeit am häufigsten auftreten, auch das (ebenfalls öffentlich verfügbare) Wissen herangezogen werden, welche Diagnosen in Form von ICD-10-Codes am häufigsten gemeinsam gestellt werden und für statistische Inferenz auf definit verschlüsselten Daten ausgenutzt werden.

10.2.3 Offenlegung von Wortlängen

In Kapitel 10.1.4 wurde beschrieben, wie bei Indexen für Infix- und Postfixsuche zusätzliche Information über die Längen der indizierten Begriffe durch die Indexstrukturen verraten werden kann. Diese Information kann ein Angreifer dem aus dem Einsatz von statistischen Angriffsmethoden resultierenden Erkenntnisgewinn hinzufügen und damit weitere Erkenntnisse bezüglich des Klartextes der verschlüsselten Datenrelation gewinnen. Etwa kann statistische Analyse belegen, dass ein bestimmter, in der Extension von $ix^{equality}_{diagnoses.icd10'}$ enthaltener Kryptotext eine sehr hohe relative Häufigkeit aufweist. Zusammen mit der Information, dass der entsprechende Klartext die Länge 3 hat, liegt die Wahrscheinlichkeit hoch, dass es sich bei diesem Klartext um das Wort, „der", „die" oder „und" handelt.

10.2.4 Offenlegung der linearen Ordnung

Ein naheliegender Angriff, der in der cTree-Indexstruktur durch das Offenlegen der linearen Ordnung der Indexelemente möglich wird, ist beispielsweise eine known-ciphertext-Attacke (sie wird in [Kamara2015] in ähnlicher Form unter der Bezeichnung *sorting attack* ebenfalls genannt): Ein Angreifer kenne für eine Menge von cTree-Indexelementen $K = \{k_1, \dots, k_n\}$ die entschlüsselten Werte für das verschlüsselte Attribut $ixData$:

$$D_K = \{d_1, \dots d_n\} = Dec\big(\pi_{ixData}(K)\big) \quad \textbf{(F73)}$$

Ohne Beschränkung der Allgemeinheit gelte $d_i < d_{i+1}$ für alle $i \in \{1, \dots, n-1\}$. Dann konstituieren die d_i die folgenden beiden Arten von Informationsverlust: Zum einen ist für alle Tupel einer Datenrelation, die per M:N-Relation auf ein k_i abgebildet werden, unvermittelt der Wert des indexierten Attributs ersichtlich. Dies mag bei einem großen Wertebereich und einem im Verhältnis dazu kleinen n nur relativ wenige Datenrelationentupel betreffen. Schwerer wiegt jedoch der andere Fall von Informationsverlust: Die d_i konstituieren eine Partitionierung ihres Wertebereichs, und für jedes Indexelement ix_m lassen sich, abgesehen von Randfällen, leicht zwei Werte d_j und d_{j+1} ermitteln, für die gilt:

$$d_j < Dec\big(\pi_{ixData}(ix_m)\big) < d_{j+1} \quad \textbf{(F74)}$$

Je mehr Elemente K umfasst, desto präziser wird diese Abschätzung, und für manche Anwendungsszenarien kann eine Annäherung an den korrekten Wert eines im Index gespeicherten Wertes bereits einen erheblichen Vertraulichkeitsverlust darstellen.

10.2.5 Informationsgewinn aus Volltextindex

Wie schon in Kapitel 10.1.5 dargelegt kann ein Angreifer die zusammenführende Eigenschaft eines verschlüsselten Volltextindex ausnutzen, der die verschiedenen, in den Kapiteln 5 bis 7 vorgestellten Indexierungstechniken einsetzt. Dabei wird ähnlich zur in Kapitel 10.2.2 beschriebenen Angriffsmöglichkeit die Eigenschaft ausgenutzt, dass mehrere unterschiedliche Indexelemente, von denen jedes einzelne für sich genommen eine Menge an Klartextinfor-

mation preisgibt, bei einem zentralen Element zusammengeführt werden. Die verschiedenen Instanzen von preisgegebener Information können ebenfalls zusammengeführt werden; aus ihrem Verbund kann der Angreifer dann weitere Erkenntnisse schließen.

10.3 Dynamische Angriffsmöglichkeiten

Im Gegensatz zur statischen Analyse, bei der ein Angreifer die Extension des Datenbankschemas eines angegriffenen Informationssystems zu einem bestimmten Zeitpunkt analysiert, beobachtet und analysiert er bei einem dynamischen Angriff die Zugriffe auf die Extension und ihre Veränderung über einen Zeitraum hinweg.

Ein Beispiel dafür ist das Zusammenfallen von schreibenden Zugriffen, also mehreren neuen bzw. aktualisierten Tupeln innerhalb eines kurzen Zeitintervalls. Dies lässt darauf schließen, dass sie zu ein- und derselben Transaktion gehören. Im Falle eines medizinischen Informationssystems, das in einer Arztpraxis eingesetzt wird, lassen sich solche Gruppen von Tupeln etwa mit einiger Wahrscheinlichkeit einem Arztbesuch eines Patienten zuordnen.

Ein weiteres Beispiel für einen dynamischen Angriff ist das zeitliche Zusammenfallen von bestimmten lesenden Zugriffen. Im obigen Beispiel ist es z. B. denkbar, dass standardisierte Abläufe in einer Arztpraxis üblicherweise in lesenden Zugriffen auf eine Menge von bestimmten Datenrelationen resultieren, die dann eine typische Menge von Treffermengen produzieren, welche wiederum jeweils einen typischen Umfang haben. Auf diese Weise ist der Angreifer in der Lage zu erkennen, welcher Vorgang in der Praxis mit einem bestimmten Patienten durchgeführt (beispielsweise das Anlegen einer Röntgenaufnahme), auch wenn er den konkreten Inhalt des Vorgangs nicht kennt.

Kontextinformation, die dem Angreifer zur Verfügung steht, kann der dynamischen Analyse eine noch höhere Mächtigkeit verleihen. Ein Beispiel hierfür ist etwa das Wissen, von welchem Ursprungsrechner, etwa in Form der IP-Adresse, eine Lese- oder Schreiboperation an den Server gelangt. Noch kritischer wird der Angriff, wenn der Angreifer in der Lage sein sollte, eine ermittelte IP-Adresse dem zugehörigen Anschlussinhaber zuzuordnen.

Bei allem Gefahrenpotenzial dynamischer Angriffe befasst sich diese Dissertation nur marginal mit der Abwehr dieser Kategorie von Angriffen. Der Grund dafür ist, dass es sich bei ihr im Gegensatz zu den statischen Angriffen aus Kapitel 10.2 um kein spezifisches Problem der verschlüsselten Indexierungsmethoden handelt. Stattdessen betreffen dynamische Angriffsmethoden alle Informationssysteme, die eine Möglichkeit zur Beobachtung bieten. Deshalb liegt im folgenden Unterkapitel der Fokus auf Abwehrmechanismen gegen die in Kapitel 10.2 vorgestellten statischen Angriffsmethoden, wohingegen Abwehrstrategien gegen dynamische Angriffe nur kurz in Unterkapitel 10.4.8 behandelt werden.

10.4 Gegenmaßnahmen

10.4.1 Skalierbare Vertraulichkeit

Nicht alle in Kapitel 10.2 genannten Formen von Informationsverlust können vollständig beseitigt werden. Es stellt sich das inhärente Problem, dass der Server beschleunigten Zugriff auf verschlüsselte Daten nur genau deshalb liefern kann, weil ihm strukturelle Information über die verschlüsselten Daten zur Verfügung gestellt wird.

Dazu zählen die definit verschlüsselten Attributextensionen für die Suche auf Gleichheit, deren Uniformität der Server ausnutzt, indem ein konventioneller Datenbankindex, vorzugsweise in Form eines Hashindexes, darauf definiert wird. Mit Hilfe dieses Indexes können Kryptotexte anhand eines ebenfalls definit verschlüsselten Suchparameters schnell gefunden werden. Auch die in zwei unterschiedlichen Repräsentationen ausgedrückte lineare Ordnung der Elemente der cTree-Indexdatenstruktur muss im Klartext dargestellt werden, da auch hier konventionelle Datenbankindexe für schnellen Zugriff darauf definiert und benutzt werden. Ohne diese im Klartext exponierte lineare Ordnung ist die Indexdatenstruktur nutzlos.

Dennoch kann der in Kapitel 10.2 und 10.3 beschriebene Informationsverlust durch unterschiedliche Maßnahmen, welche in den folgenden Unterkapiteln beschrieben werden, beseitigt oder zumindest abgeschwächt werden. Struktur und Betriebsmodus der Indexstrukturen werden verändert, so dass das Vertraulichkeitsniveau der verschlüsselten Daten erhöht wird, wenn auch unter Inkaufnahme von Performanzverlusten.

Es ist die Aufgabe eines Informationssystem-Designers, zu entscheiden, ob der Einsatz der Indexstrukturen vertretbar ist oder ob das Schutzniveau der Daten es erfordert, eine modifizierte Version der Indexstrukturen zu benutzen, die den Informationsverlust zulasten der Performanz verringert. In einem extremen Szenario, in dem keinerlei Informationsverlust geduldet werden kann, müsste konsequenterweise auf den Einsatz von Indexstrukturen komplett verzichtet werden, und das Informationssystem könnte nur unter erheblichen Einschränkungen der Benutzbarkeit betrieben werden.

Angesichts der Anwendungsorientiertheit dieser Dissertation bietet sich ein pragmatischer Umgang mit diesem Trade-off an, der im Folgenden als *skalierbare Vertraulichkeit* bezeichnet wird: Zunächst wird eine sorgfältige Abwägung vorgenommen zwischen dem nötigen Schutzniveau von Daten, auch und insbesondere unter Berücksichtigung des im BDSG erwähnten Prinzips der Verhältnismäßigkeit (siehe Kapitel 1.3.1), auf der einen Seite und der Notwendigkeit von beschleunigtem Zugriff darauf auf der anderen Seite. Aufbauend auf dieser Abwägung wird eine angemessene Balance zwischen Vertraulichkeit und Performanz gewählt, so dass Minimalanforderungen an die Performanz bei der unter diesen Umständen maximal erreichbaren Vertraulichkeit erfüllt werden (beziehungsweise: erfüllte Minimalanforderungen an Vertraulichkeit bei der maximal dabei erreichbaren Performanz).

In einem solchen Modell, bei dem sich die Performanz eines Client-Server-Informationssystems mit zunehmender Offenlegung von Struktur auf dem Server immer mehr erhöhen lässt (unter gleichzeitiger Inkaufnahme von graduellem Vertraulichkeitsverlust), bewegt sich skalierbare Vertraulichkeit zwischen den folgenden zwei Extremen:

- Maximale Vertraulichkeit: Dieses Extrem erfordert ein komplett monolithisches System, das in einem Datenblock fester Größe auf dem Server gespeichert ist, verschlüsselt mit einem zufällig gewählten One-Time-Pad mit derselben Bitlänge, wie sie der Datenblock aufweise. Für einen Zugriff muss der gesamte Block vom Server zum Client transferiert werden, wo er entschlüsselt und verarbeitet wird. Anschließend wird ein neues One-Time-Pad gewählt und der Block damit umgeschlüsselt. Ein solches System gibt keine Information über die darin gespeicherte Information preis, abgesehen von einer groben Abschätzung über das im System gespeicherte Datenvolumen. Gleichzeitig ist offensichtlich, dass es hinsichtlich seiner Benutzbarkeit (insbesondere

durch mehrere Benutzer) und Performanz stark eingeschränkt ist. Das System bietet also maximale Vertraulichkeit bei minimaler Performanz.

- Maximale Performanz: Hierfür wird der Stand der Technik bezüglich der Performanz von Client-Server-basierten Informationssystemen angenommen, der mit einem serverseitig voll im Klartext arbeitenden System erreichbar ist. Ein solches System weist somit maximale Performanz und Benutzbarkeit bei minimaler Vertraulichkeit auf.

Im Folgenden werden verschiedene Abwehrmechanismen zur Minimierung von Informationsverlust vorgestellt, die helfen, die Vertraulichkeit eines Informationssystems unter Wahrung des Prinzips der Verhältnismäßigkeit skalierbar zu gestalten.

10.4.2 Verzicht auf definite Verschlüsselung

Ein Verzicht auf definite Verschlüsselung von Attributen bei einer Indexierung für Gleichheitssuche ist möglich. An deren Stelle tritt der Einsatz der cTree-Datenstruktur zur Gleichheitssuche, der bereits in Kapitel 6.8.4 skizziert wurde und der ohne definite Verschlüsselung auskommt, wie am folgenden Beispiel verdeutlicht wird:

Auf einem beliebig verschlüsselten Attribut a' einer Datenrelation r, für das Gleichheitssuche realisiert werden soll, sei ein cTree-Index definiert. Dabei sei die Extension des Attributs $ixData$ der zugehörigen Indexrelation gegebenenfalls entsprechend der für a' definierten Ausprägung von semantischer Gleichheit normalisiert und ebenfalls randomisiert verschlüsselt[95]. Dann funktioniert eine Gleichheitssuche für das Attribut exakt so wie der Lookup-Schritt der cTree-INSERT-Operation, der in Kapitel 6.4.2 beschrieben ist.

Diese Implementierung der Gleichheitssuche bietet eine deutlich niedrigere Performanz als die Implementierung, die definite Verschlüsselung benutzt. Die gilt nicht nur für SELECT-, sondern auch für INSERT-, UPDATE- und DELETE-Anweisungen. Dennoch gibt es keinen prinzipiellen Verlust der Funktionstüchtigkeit, und die Performanz ist immer noch höher als die eines naiven Ansatzes, der die gesamte verschlüsselte Extension des indexierten Attributs auf den Client überträgt, wo sie dann entschlüsselt und durchsucht wird.

Jedoch ist anzumerken, dass bei einer Ersetzung von definit verschlüsselten Indexattributen durch eine cTree-Indexstruktur das Problem von möglichen Häufigkeitsanalysen noch nicht gelöst ist: Bei einer Extension eines indizierten Attributs, die im Klartext auffällige Häufigkeitscharakteristika aufweist, gibt zwar die entsprechende verschlüsselte Extension keine Information mehr preis, aber die M:N-Relation zwischen Index- und Datenrelation verknüpft bestimmte Indextupel häufiger mit Datentupeln als andere. Diese Information kann durch einen Angreifer für eine Häufigkeitsanalyse ausgenutzt werden und liefert den gleichen Erkenntnisgewinn.

10.4.3 Verschlüsselung von Referenzen

Eine Maßnahme, obiger Problematik zu begegnen ist, die Verknüpfung von Index- und Datenrelation durch die M:N-Relation aufzuheben bzw. dem Server unkenntlich zu machen.

---

[95] An dieser Stelle gelten keine Einschränkungen für die Interpretation semantischer Gleichheit. Auch die Gleichheitsinterpretation für Gleitkommazahlen aus Kapitel 5.6.3 lässt sich hiermit realisieren, wenn auch mit einigen kleineren strukturellen Änderungen der cTree-Relation, auf die an dieser Stelle aus Platzgründen jedoch nicht näher eingegangen wird.

10.4.3.1 Verschlüsseln der M:N-Relation

Dies kann erreicht werden, indem mindestens eine der beiden Fremdschlüsselbeziehungen, die von der M:N-Relation ausgehen, verschlüsselt wird, vorzugsweise mit einem randomisierten Verfahren. Es empfiehlt sich, nur eine Fremdschlüsselbeziehung zu verschlüsseln, da dies ausreicht, um dem Server die Verknüpfung zwischen Index- und Datenrelation unkenntlich zu machen unter begrenzten Performanzeinbußen.

Da Zugriffspfade stets von der Indexrelation über die M:N-Relation auf die Datenrelation laufen, liegt es nahe, diejenige Fremdschlüsselbeziehung zu verschlüsseln, die die Datenrelation referenziert. Entsprechend der in dieser Dissertation verwendeten Terminologie wird dadurch das Attribut $refData$ zu $refData'$.

10.4.3.2 SELECT-Operation

Eine beschleunigte Gleichheits- oder Ungleichheitssuche (nach Suchbegriff s) auf eine derart cTree-indizierte Datenrelation r über eine M:N-Relation mn und eine Indexrelation ix geht dann folgendermaßen vonstatten:

- ***(Client/Server)*** Die Suche nach s auf ix folge dem entsprechenden, in 6.4.2 bzw. 6.8 beschriebenen Ablauf und liefere eine gegebenenfalls sortierte Menge von Indextupeln $ix_s \subseteq ix$.
- ***(Server)*** Ermittele durch eine JOIN-Operation über die im unverschlüsselten Attribut $refIndex$ von mn ausgedrückte Fremdschlüsselbeziehung eine Menge von Tupeln $mn_s \subseteq mn$ mit:

  $$mn_s = \pi_{mn.*}\left(mn \bowtie_{mn.refIndex=ix.seq} ix_s\right) \quad \textbf{(F75)}$$

- ***(Server)*** Sende deren verschlüsselte Referenzen $ref_s' = \pi_{refData'}(mn_s)$ auf Tupel der Datenrelation an den Client.
- ***(Client)*** Entschlüssele die Elemente von ref_s' zu einer Menge von Klartextreferenzen $ref_s = Dec(ref_s')$ und sende sie an den Server als Parameter einer Anfrage nach der entsprechenden Menge von Datentupeln.
- ***(Server)*** Selektiere die geforderte Menge von Datentupeln

  $$r_s = \pi_{r.*}\left(ref_s \bowtie_{ref_s.refData=r.seq} r\right) \quad \textbf{(F76)}$$

  und gebe sie dem Client zurück, welcher sie gegebenenfalls entschlüsselt. Es sei angemerkt, dass die im ersten Schritt ermittelte Sortierung der Indextupel über alle genannten Schritte hinweg aufrecht erhalten wird, so dass die finale Ergebnismenge r_s ebenfalls dem Client mit korrekter Sortierung übergeben werden kann.

Dieses modifizierte Vorgehen wirkt sich negativ auf die Performanz der Suchoperation aus: Bisher war die Durchführung einer Suche auf einem cTree-indexierten Attribut einer Datenrelation in die drei Phasen:

1. der Bestimmung der Suchparameter auf dem Client,
2. der cTree-Traversierung und
3. des finalen SELECTs

unterteilt (siehe Kapitel 6.8.2). Phase 3 wird nun in zwei Phasen aufgespalten: den Zugriff auf die Indexrelation und den finalen SELECT auf die Datenrelation. Es kommt also ein wei-

terer Client-Server-Roundtrip hinzu, da die Entschlüsselung der Referenzen nur vom Client geleistet werden kann. Hinzu kommt der Aufwand der Entschlüsselung selbst und der Aufwand für den Transfer der Referenzen vom Server zum Client und zurück.

Da die cTree-Traversierung jedoch nach wie vor den teuersten Teil der ganzen Operation ausmacht, nimmt sich der Performanzverlust moderat aus. In einer experimentellen Implementierung belief er sich auf eine lediglich ca. 15% längere Laufzeit (siehe Kapitel 11).

10.4.3.3 INSERT-, UPDATE- und DELETE-Operation

Die INSERT-Operation bleibt von der oben beschriebenen Modifikation weitgehend unberührt; lediglich beim Einfügen von Tupeln in die M:N-Relation muss der Client die Referenz auf das Datentupel in verschlüsselter Form an den Server senden.

Das Vorgehen bei einer DELETE-Operation auf der Datenrelation ändert sich dagegen folgendermaßen: Der Client führt vor der Löschung eines Tupels t der Datenrelation eine Suche nach t analog zur SELECT-Operation (siehe Kapitel 10.4.3.2) durch, welche ein Indextupel und ein oder mehrere M:N-Tupel liefert. Aus letzteren wird mit Hilfe des Clients das t referenzierende M:N-Tupel identifiziert und gelöscht. Falls es sich dabei um das einzige M:N-Tupel gehandelt hat, kann das singuläre Indextupel ebenfalls gelöscht werden, falls dies anwendungsseitig gewünscht ist.

Die UPDATE-Operation funktioniert analog zur DELETE-Operation mit dem einzigen Unterschied, dass das t verschlüsselt referenzierende M:N-Tupel nicht gelöscht, sondern überschrieben werden muss, so dass es ein anderes Indextupel referenziert, welches dem neuen Zustand des Datentupels entspricht.

10.4.3.4 Häufigkeitsanalysen im modifizierten Fall

Die Entscheidung, die Fremdschlüsselbeziehung zwischen M:N- und Indexrelation im Klartext zu belassen, begünstigt den beschleunigten Zugriff auf Tupel der Datenrelation. Sie bewirkt jedoch auch, dass immer noch Häufigkeitsanalysen auf der Extension der M:N-Relation möglich sind: Für jedes Datentupel t_d, das einem Indextupel t_{ix} zugeordnet wird, enthält die M:N-Relation genau ein Tupel t_{mn} mit $\pi_{refIndex}(t_{mn}) = \pi_{seq}(t_{ix})$. Bei Häufigkeitscharakteristika im Klartext der Datenrelation wie bei der Vergabe von ICD-10-Codes (siehe Kapitel 10.2.1) kann ein Angreifer für einige Indextupel mit relativ hoher Wahrscheinlichkeit abschätzen, welchen Inhalt sie kodieren, wie etwa einen bestimmten ICD-10-Code.

Nicht mehr möglich ist hingegen eine Zuordnung, welche Datentupel welchen Indextupeln zugeordnet sind, so dass die eigentliche produktive Extension eines Datenbankschemas wirksam vor statistischen Angriffen geschützt ist. Die bloße Erkenntnis, dass die Extension der Index- und M:N-Relationen bestimmte Charakteristika herausbilden, die mit öffentlichem Wissen abgleichbar sind, ist für einen Angreifer von geringem Nutzen.

10.4.4 Redundante Speicherung und randomisierte Verschlüsselung

Im vorangegangenen Unterkapitel wurde beschrieben, wie nach der einseitigen Verschlüsselung der M:N-Relation eine statische Häufigkeitsanalyse auf dem Verbund einer cTree-Indexrelation und einer M:N-Relation immer noch möglich ist. Dies lässt sich vollständig eliminieren, indem die Indexrelation nicht mehr eindeutige Werte beinhalten muss; stattdessen werde für jedes indexierte Datentupel ein separates Indextupel in die Indexrelation eingefügt, unabhängig davon, ob der gleiche Wert bereits im Index vorhanden ist oder nicht.

Jedes Indexelement wird darüber hinaus durch Beimischen von randomisierter Zusatzinformation (*Salz*) eindeutig gemacht; hierfür eignet sich die Wahl eines randomisierten Verschlüsselungsverfahrens besonders gut, wie beispielsweise AES-256 im CBC-Modus. Durch die Initialisierung einer jeden Verschlüsselungsoperation mit einem randomisierten IV entstehen Kryptotexte, die nach heutigem Stand der Technik nicht korrelierbar sind, egal, ob sie die gleichen Klartextwerte verschlüsseln, ähnliche oder stark unterschiedliche.

10.4.4.1 Modifikation des Lookup-Schritts

Da der Lookup-Schritt nun nicht mehr terminieren darf, wenn der gesuchte (also der einzufügende) Wert gefunden wurde, muss die in 6.4.2 angegebene Spezifikation angepasst werden: Schritt 6 entfällt dann ersatzlos, und der bisherige Schritt 7 wird modifiziert zu:

(Client) Falls $Dec(v_{current}.ixData) \geq s_f$, setze $v_{current} = v_{current}.leftChild$.

Alternativ dazu kann der bisherige Schritt 7 unverändert bleiben und der bisherige Schritt 8 modifiziert werden zu:

(Client) Falls $Dec(v_{current}.ixData) \leq s_f$, setze $v_{current} = v_{current}.rightChild$.

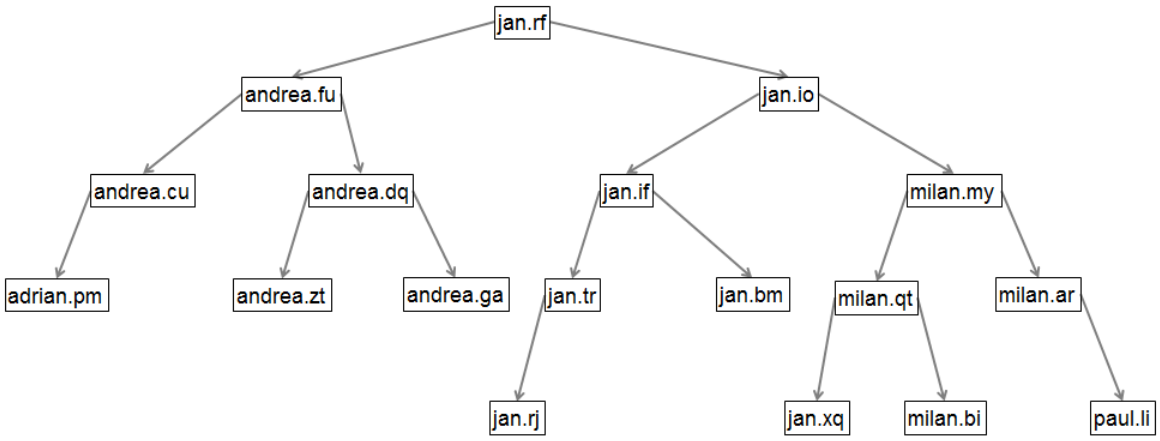

Abb. 12: cTree-Binärbaum mit redundanten, gesalzenen Einträgen.

In Abb. 12 ist ein binärer Baum symbolisch dargestellt, der bei der so modifizierten Indexierung eines Vornamen speichernden Attributs einer Datenrelation angelegt wurde. Der randomisierte IV wird durch zwei zufällig gewählte Zeichen symbolisiert, die an den tatsächlich im Indexelement enthaltenen Wert angehängt werden. Es ist zu beachten, dass innerhalb einer Gruppe von gleichen Werten die sich durch die unterschiedlichen IV ergebende Reihenfolge ignoriert wird; in der Abbildung ist lediglich der Wert vor dem jeweiligen Punkt relevant.

10.4.4.2 Auswirkung auf die Möglichkeiten statistischer Analyse

Bei dieser Betriebsweise können sich praktisch keine Häufigkeitscharakteristika herausbilden, da für jedes Datentupel genau ein Indextupel eingefügt wird. Zudem verhindert der zufällig gewählte IV, dass zwei Indexelemente als den gleichen Klartext verschlüsselnd erkannt werden. Die Wahrscheinlichkeit dafür, dass dies zufällig eintritt, liegt für AES bei $2^{-128} \approx 10^{-38}$, so dass dies für keinen sinnvollen Angriff genutzt werden kann.

In einem Informationssystem, das mit den folgenden drei Modifikationen bezüglich der verschlüsselten Indexstrukturen betrieben wird:

- Verzicht auf definite Verschlüsselung und stattdessen Einsatz der cTree-Indexstruktur sowohl für Gleichheits- als auch für Ungleichheitssuche,
- Verschlüsselung der Referenz der M:N-Relation auf die Datenrelation und
- Redundante Speicherung von randomisierten Indexeinträgen in der cTree-Relation,

werden statische Angriffe auf der Extension des Datenbankschemas für einen Angreifer sehr viel schwieriger und erhöhen somit die Vertraulichkeit der gespeicherten Daten signifikant.

10.4.4.3 Modifikation des cTree-Lookup-Schritts

Die in Kapitel 10.4.4.2 angegebene Modifikation der Verwendung der cTree-Indexstruktur ändert die Ausführung ihres Lookup-Schritts:

- Bei der Ermittlung der Einfügeposition für ein neues Indexelement wird der Lookup-Schritt so abgeändert wie in Kapitel 10.4.4.1 beschrieben.
- Bei einer Gleichheitssuche nach einem Wert s wird diese zu einer Art Intervallsuche modifiziert, in der der Baum so lange traversiert wird, bis das „kleinste" Element gefunden wurde, das s entspricht (in Abb. 12 entspricht dies bei einer Gleichheitssuche nach „jan" dem Wert „jan.rf" an der Wurzel des Baumes). Weiterhin wird der Baum auf analoge Weise ein weiteres Mal traversiert, bis das „größte" Element gefunden wurde, das dem Suchwert entspricht (in Abb. 12 entspricht dies dem Wert „jan.io"). Unter Verwendung des jeweiligen $sortNumber$-Attributwerts der beiden gefundenen Indexelemente wird dann eine Intervallsuche durchgeführt, deren Ergebnis entsprechend Kapitel 10.4.3.2 weiterverwendet wird.
- Zum Auffinden des bestmöglichen Approximationswertes für eine geschlossene Intervallgrenze kann ebenfalls nicht mehr so vorgegangen werden, dass die Traversierung des Baums terminiert, wenn ein entsprechender Wert mit einer exakten Übereinstimmung gefunden wurde (siehe Schritt 7 in Kapitel 6.8.3.2). Stattdessen wird analog zur oben beschriebenen Modifikation der Gleichheitssuche das „kleinste" Element in der Extension der Indexrelation gesucht, das dem gesuchten Intervallgrenzenwert gleicht. Enthält die Extension keinen exakt mit der gesuchten Intervallgrenze übereinstimmenden Wert, so werde verfahren wie bisher.

10.4.4.4 Performanz-Nachteile

Die in Kapitel 10.4.4.2 angegebene Modifikation verschlechtert die Performanz eines verschlüsselten cTree-Index. Dies liegt sowohl der komplizierteren Ausführung des Lookup-Schritts als auch am unter Umständen erheblich gestiegenen Datenvolumen des Indexes.[96]

10.4.5 Zyklische Verschiebung des Attributwertebereichs

Dieses Unterkapitel behandelt eine weitere, zu den in den vorangegangenen Unterkapiteln beschriebenen Abwehrmechanismen alternative Methode, statische Häufigkeitsanalysen auf der verschlüsselten Extension der cTree-Indexrelation zu erschweren. Sie ist anwendbar auf

[96] Zur Abschwächung für letzteres ließe sich eine Zwischenlösung konzipieren, bei der die mehrfache M:N-Referenzierung eines Indexelements in der Form erlaubt wird, dass bis zu beispielsweise $n = 5$ Referenzen von M:N-Tupeln auf ein Indextupel erlaubt sind.

diskret numerischen, cTree-indizierten Attributen, wobei im Folgenden ohne Beschränkung der Allgemeinheit von einem 32 Bit langen Integer-Datentyp mit Vorzeichen ausgegangen wird. Der Wertebereich $DOM = \{MIN, \ldots, MAX\}$ eines Attributs dieses Datentyps hat demnach eine Kardinalität von $|DOM| = 4.294.967.296$, und seine Grenzen sind mit $MIN = -2.147.483.648$ und $MAX = 2.147.483.647$ definiert.

Nach der Abwehrstrategie wird ein Wert $c \in DOM$ zufällig gewählt, der eine Rotation parametrisiert, also eine zyklische Verschiebung von DOM. Sie werde dargestellt durch eine Funktion $rot_c: DOM \to DOM$. c muss allen Benutzern des Informationssystems zugänglich sein, darf aber nicht im Klartext im Server gespeichert werden. Es bietet sich also die verschlüsselte Speicherung von c in einer speziellen Relation auf dem Server an.

Die Rotation werde dann so durchgeführt, dass zu jedem zu indexierenden Wert x der Wert $rot_c(x)$ berechnet wird mit:

$$x_{rot_c} = rot_c(x) \coloneqq \big((x - MIN + c)\ mod\ |DOM|\big) + MIN \qquad \textbf{(F77)}$$

Anstelle von x werde x_{rot_c} in die cTree-Indexdatenstruktur eingefügt; die Invarianten bezüglich der linearen Ordnung werden dabei dennoch aufrechterhalten, so dass die Verteilung der Werte darin über $[MIN, MAX]$ zu einer Pseudo-Verteilung zyklisch rotiere. Die Attributwerte in der indexierten Datenrelation werden unverändert so wie bisher gespeichert.

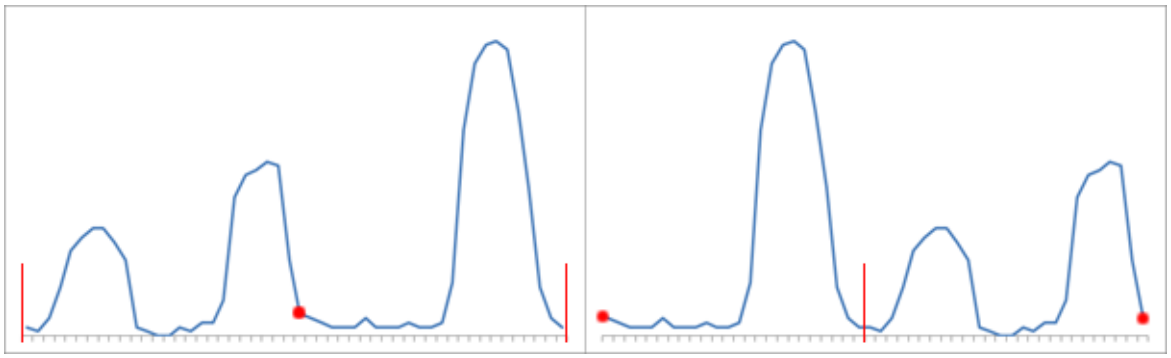

Abb. 13: Werteverteilung vor und nach der zyklischen Verschiebung des Wertebereichs.

In Abb. 13 ist die Modifikation des Wertebereichs grafisch dargestellt: Die Grenzen des Wertebereichs MIN und MAX werden durch rote Linien gekennzeichnet. Im rotierten Wertebereich wird MIN auf $MIN + c$ abgebildet und MAX auf $MIN + c - 1$, so dass die beiden Werte nunmehr unmittelbar nebeneinander liegen. Dagegen wird $MAX - c + 1$ auf MIN abgebildet und $MAX - c$ auf MAX.

Intervallsuchen nach Elementen, die in $[a, b]$ liegen, werden auf der so veränderten Datenstruktur ebenfalls modifiziert, so dass jeweils dasjenige Indexelement (bzw. sein $sortNumber$-Attributwert) ermittelt wird, das größer oder gleich $rot_c(a)$ bzw. kleiner oder gleich $rot_c(b)$ ist. Hier ist nun im WHERE-Kriterium der finalen SELECT-Operation eine Fallunterscheidung durchzuführen:

Falls $sn_{rot(a)} \leq sn_{rot(b)}$:

```
WHERE sortNumber >= sn_rot(a) AND sortNumber <= sn_rot(b)
```
(Q43)

sonst:

```
WHERE sortNumber >= sn_rot(b) OR sortNumber <= sn_rot(a)
```
(Q44)

Bekannte Häufigkeitscharakteristiken auf Klartextdaten sind für Angreifer in diesem Szenario nur erschwert anwendbar, bei einer gleichzeitig praktisch unverminderten Performanz im Vergleich zur unmodifizierten Indexstruktur. Jedoch ist die Gefahr bei weitem nicht so abgeschwächt wie bei den in den Unterkapiteln 10.4.2 bis 10.4.4 beschriebenen Strategien, da die Häufigkeitscharakteristika hier lediglich verfremdet und nicht eliminiert werden. Somit hat diese Abwehrstrategie eher ergänzenden Charakter und bzw. ist eher bei niedrigeren Bedrohungsszenarien zu empfehlen.

10.4.6 Verschleierung der präzisen Wortlängen

Der Informationsverlusts durch die Angabe der Wortlängen in der erweiterten M:N-Relation (siehe Kapitel 10.2.3) lässt sich zumindest abschwächen, indem diese Wortlängen durch gezielten Einsatz von Unpräzision verschleiert werden. An den M:N-Tupeln wird dann nur die ungefähre Wort- bzw. Suffixlänge des Wortes in demjenigen Indextupel annotiert, das sie referenzieren, und bei einer Suche werden alle Tupel, die Elemente der Treffermenge sein könnten, zum Client transferiert. Dieser filtert die Kandidatenliste nach tatsächlichen Treffern. Durch diese Modifikation wird in geringem Maße Overhead generiert durch den überflüssigen Transport von Daten zum Client zuzüglich des Aufwands der Filterung, so dass die Performanz der Datenstruktur sinkt.

10.4.7 k-anonymity

Eine seit Längerem bekannte Strategie, um Häufigkeitsanalysen zu erschweren, ist das in [Sweeney2002] vorgestellte *k-anonymity*-Verfahren. Es kann im Kontext der in dieser Dissertation vorgestellten Indexierungsmethoden in der Form angewendet werden, dass für ein definiertes $k \in \mathbb{N}$ für jedes n-fach referenzierte Indextupel mindestens $k - 1$ weitere Tupel in der Extension der Indexrelation enthalten sind, die ebenfalls n-fach referenziert werden. Falls dies in der tatsächlichen Extension nicht der Fall ist, wird es erreicht, indem zusätzliche Pseudotupel (*Dummytupel*) in den Index, etwa in die cTree-Relation, eingefügt werden, zusammen mit entsprechend vielen Dummy-M:N- und Dummy-Datentupeln. Bei SELECT-Anfragen können Dummytupel mit in die Treffermenge gelangen, so dass sie ebenfalls zum Client transferiert werden, der sie dann herausfiltern muss.

k-anonymity verursacht mitunter erheblichen Overhead in Form von zusätzlichem Datenvolumen, zum einen bezüglich der Persistierung der vergrößerten Datenbankextension auf dem Server als auch bezüglich des über das Netzwerk zu übertragenden Datenvolumens auf den Client. Dennoch ist es ein berechtigtes Konzept, das helfen kann, die Bedrohung durch statistische Analysen zwar nicht zu eliminieren, aber zumindest abzuschwächen.

10.4.8 Abwehrmechanismen gegen dynamische Angriffsmethoden

Wie in Kapitel 10.3 angedeutet, liegt der Schwerpunkt des Kapitels 10.4 auf Abwehrstrategien gegen statische Angriffe, die spezifisch für die in dieser Dissertation vorgestellten Indexstrukturen entwickelt werden können. Gegen dynamische Angriffe bieten sich dagegen eher Standardverfahren als Abwehrstrategien an, die für andere Bereiche, insbesondere bei Rechnernetzen, genau so relevant sind wie für die hier vorgestellten Indexstrukturen. Aus diesem Grund sind sie bereits in anderen Veröffentlichungen gründlich erforscht und beschrieben und nehmen in diesem Kapitel einen geringeren Raum ein.

10.4.8.1 Padding

So ist beispielsweise *Padding* zu nennen, eine Methode, mit der charakteristische Blockgrößen von über das Netzwerk übertragen Nachrichten verschleierbar sind. Dies wird erreicht, indem Datenblöcke bis zu einer definierten Standardgröße mit Zufallsbits aufgefüllt werden, welche dann später wieder entfernt werden. Offensichtlich geht dieses Vorgehen zu Lasten der Übertragungskapazität des Systems, so dass besagte Blockgröße bezüglich des sich hier ergebenden Trade-offs sorgfältig zu wählen ist.[97]

10.4.8.2 Zeitliche Inkohärenz

Eine weitere Abwehrmaßnahme, die ein charakteristisches, zeitliches Zusammenfallen von übers Netzwerk übertragenen Nachrichten verschleiern kann, ist, die Versendung der betreffenden Nachrichten zu verzögern, so dass es erschwert wird, Korrelationen dazwischen herzustellen. Bereits in [Franz1998] ist ein solches Verfahren beschrieben.

Diese Vorgehensweise geht zu Lasten der Reaktionsfähigkeit des Systems, so dass auch hier ein Trade-off existiert und die Höhe des Zeitversatzes ebenfalls mit Sorgfalt zu wählen ist. Weiterhin ist beim bewussten Einsatz von zeitlicher Inkohärenz darauf zu achten, dass diese Inkonsistenzen im Datenbestand bewirken kann. Dies gilt insbesondere für den schreibenden Zugriff, für den sich illustre Beispiele konstruieren lassen wie folgendes: Die zeitlich versetzte Speicherung der folgenden beiden Notizen in einem medizinischen Informationssystem könnte schwerwiegende Folgen haben:

- „Patient bekommt Medikament x verschrieben."
- „Medikament x sollte nicht mit Medikament y kombiniert werden, da Patient einen allergischen Schock erleiden könnte."

10.4.8.3 Anonymisierung von Verbindungsdaten: Onion Routing

Ein Werkzeug zur Anonymisierung von Verbindungsdaten im Internet liefert das *Onion Routing*-Verfahren, das in einer in [Dingledine2004] beschriebenen, fortgeführten Version vom Tool *Tor* (*The Onion Router*)[98] eingesetzt wird. Es basiert auf einer mehrfachen asymmetrischen Verschlüsselung von Nachrichten, wobei jede Verschlüsselung einem Knoten auf einer willkürlich gewählten Route durch ein Netzwerk von speziellen Verbindungsroute zugedacht ist. Dabei ist auch die Route selbst Teil der verschlüsselten Nachricht. Durch weitere Maßnahmen wie ständig wechselnde Routen wird die Sicherheit zusätzlich erhöht.

Trotz einiger Möglichkeiten zur Kompromittierung[99] ist Onion Routing immer noch ein sehr wirkungsvolles Hilfsmittel zur Verwendung anonymisierter Kommunikation im Internet. Es kann als Abwehrmechanismus gegen die Analyse von Verbindungsdaten eingesetzt werden, bei der Client-IP-Adressen, welche konkreten Benutzeridentitäten zugeordnet sind, mit dem Zugriff auf spezifische Tupel des Informationssystems korreliert und daraus Rückschlüsse gezogen werden können.

---

[97] siehe [Ferguson2003], S. 68.
[98] siehe [URL-TOR].
[99] siehe [URL-ICISSP].

11 Performanzmessungen

Die in dieser Dissertation vorgestellten Methoden für Erstellung, Pflege und Einsatz von verschlüsselten Indexstrukturen für den beschleunigten Zugriff auf verschlüsselte Daten wurden in einer beispielhaften Implementierung einer Reihe von Tests unterzogen.

11.1 Testumfang

Dabei wurde nicht jede einzelne in den Kapiteln 5 bis 8 behandelte Variante ausimplementiert und getestet, da dies den sinnvollen Umfang dieses Kapitels und der Dissertation übersteigen würde. Stattdessen konzentrierten sich die Tests auf die beiden Kernverfahren aus den Kapiteln 5 und 6:

- definite Verschlüsselung für die Gleichheitssuche auf verschlüsselten Daten (siehe Kapitel 5) sowie
- ordnungserhaltende Verschlüsselung, bzw. eine Datenstruktur, die auf beliebig verschlüsselten Daten deren Ordnung nachbildet, wodurch Ungleichheitssuchen und damit auch Intervall- und Präfixsuchen auf verschlüsselten Daten ermöglicht werden (siehe Kapitel 6).

Es wurden absolute und relative Messungen durchgeführt, die die Operationen auf verschlüsselten Indexen in ihrer Grundform (*BASIC*) solchen gegenüberstellten, die auf konventionell indexierten Klartext-Datenbankextensionen analoge Operationen durchführten (*CLEARTEXT*). Weiterhin wurde eine entsprechend Kapitel 10.4.3 (einseitig randomisiert verschlüsselte M:N-Relation) und 10.4.4 (redundant gespeicherte Indexelemente, randomisiert verschlüsselt) modifizierte cTree-Variante (*SECURE*) getestet, bei der der bei *BASIC* entstehende Informationsverlust in einem statischen Angriffsszenario vermieden wurde. Die *SECURE*-Variante bot also entsprechend Kapitel 10 zwar einen höheren Grad an Vertraulichkeit, benötigte aber neben zahlreicheren Ver- und Entschlüsselungsoperationen auch einen zusätzlichen Client-Server-Roundtrip und wies dementsprechend höhere Laufzeiten auf.

Alle Tests wurden jeweils auf Datenbankextensionen unterschiedlichen Umfangs durchgeführt, d. h., auf Datenbankrelationen mit 100, 1.000, 10.000 und 100.000 Tupeln (kodiert mit *T_100*, *T_1.000*, *T_10.000* und *T_100.000*).

Orthogonal zum Umfang der Daten dazu wurden die Tests mit variierenden Client-Server-Latenzen von 0 ms (*PING_0*) und 25 ms (*PING_25*) durchgeführt. Die Höhe der vom Client angefragten cTree-Teilbäume, ein Parameter, dessen Einfluss auf die Performanz in Kapitel 11.4 separat analysiert wird, wurde auf 4 festgelegt. Weiterhin wurden die Tests auch auf Beispieldaten mit einem unterschiedlichem Grad an Redundanz ausgeführt, wobei sich dies nur moderat auf die Laufzeiten auswirkte (siehe Kapitel 11.3.4).

Schließlich wurden weitere Tests auf den cTree-Indexen unter Einbeziehung der in Kapitel 9 vorgestellten Maßnahmen zur Performanzsteigerung durchgeführt und mit *BASIC* verglichen (siehe Kapitel 11.5).

Bezüglich der Interpretation der Messergebnisse waren zwar auch komparative Ergebnisse von Belang, gingen also der Frage nach, um wieviel *CLEARTEXT* in den verschiedenen Ausführungsszenarios schneller war als *BASIC* und *SECURE*; noch mehr aber lag der Fokus auf der Frage, ob *BASIC* und wenn möglich auch *SECURE* auf substanziell großen Datenmengen

gute oder zumindest akzeptable Ergebnisse für den operativen Einsatz in einem gängigen Informationssystem liefern können. Somit lag das Hauptinteresse bei der Interpretation der Testergebnisse auf den absoluten Laufzeiten der verschlüsselten Indexvarianten, wobei der in Kapitel 1.4.4 festgelegte Richtwert von 300ms zwischen Absetzen der Anfrage und Vorliegen des entschlüsselten Ergebnisses nicht unterschritten werden sollte.

11.2 Implementierung

Alle Tests wurden in einer 3-Schicht-Client-Server-Architektur ausgeführt, wie sie in einem zeitgemäßen, webbasierten Informationssystem verwendet würde, bestehend aus den folgenden Komponenten:

- Client: Es kam ein HTML5-Client zum Einsatz, bei dem die Ver- und Entschlüsselungsoperationen (unter Verwendung von AES-256-CBC) in JavaScript durchgeführt wurden. Der Client kommunizierte mit dem Server über WebSockets und verschickte und empfing Daten darüber als CBOR-serialisierte Nachrichten. Der Client lief im Browser Google Chrome 45 auf dem Betriebssystem Windows 7 Professional 64 Bit, Service Pack 1. Als Hardware fungierte ein Notebook vom Typ Thinkpad T530 mit einem Prozessor vom Typ Intel® Core™ i7-3720QM mit 2,6 GHz und 12 GB RAM.
- Middleware (Application Server): Der Application Server war mit relativ wenigen Aufgaben ausgestattet und hatte in dieser Architektur lediglich die Aufgabe der CBOR-Serialisierung und -Deserialisierung von Daten für die Kommunikation mit dem Client sowie die entsprechende Kommunikation mit dem DBMS über die JDBC-Datenbankschnittstelle. Zum Einsatz kam der JBoss-Server Wildfly 8.0.1.
- Datenbank: Als DBMS fungierte PostgreSQL 9.3, auf dem neben konventionell indexierten Datenbankrelationen einige spezielle Datenbankprozeduren für das optimierte Retrieval von cTree-Teilbäumen sowie für die cTree-Balancierung (siehe Kapitel 6.4.2) zum Einsatz kamen.

11.3 Basisoperationen auf verschlüsselten und Klartextdaten

11.3.1 Einfügeoperation

Es wurde die durchschnittliche Einfügedauer für ein einzelnes Tupel in eine cTree-indizierte Datenbankrelation in den Testvariationen T_100 bis T_100.000 ermittelt. Alle Tests wurden sowohl unter dem $PING\_0$- als auch dem $PING\_25$-Szenario durchgeführt. Wie in Kapitel 11.1 erläutert, wurde dieser Test sowohl auf einer konventionell indexierten Klartextrelation ($CLEARTEXT$) durchgeführt als auch auf einer Datenbankrelation mit verschlüsselter Extension, die mit einem cTree-Index in seiner Basisausführung versehen war ($BASIC$). Weiterhin wurde der Test auch für eine Datenrelation durchgeführt, deren verschlüsselte Extension mit einem redundanten, entsprechend Kapitel 10.4.4 modifizierten cTree indexiert war ($SECURE$).

Die in Abb. 14 dargestellten Ergebnisse zeigen, dass die $CLEARTEXT$-Variante erwartungsgemäß in allen Testvariationen signifikant schneller lief als $BASIC$ und $SECURE$, wobei sich der Unterschied zu diesen cTree-Varianten sich selbst in den ungünstigsten Varianten noch akzeptabel ausnahm. So lief $CLEARTEXT$ unter $PING\_0$ und $T\_100.000$ etwa 10-mal so schnell wie $SECURE$, während $CLEARTEXT$ unter $PING\_25$ und T_100 weniger als 5-mal so schnell war wie $SECURE$ und weniger als 3-mal so schnell wie $BASIC$.

Weiterhin fiel auf, dass die $CLEARTEXT$-Laufzeiten zwischen T_100 und $T\_100.000$ sowohl für $PING\_0$ als auch für $PING\_25$ kaum voneinander abwichen, während sie unter $BASIC$ und $SECURE$ fast linear anstiegen (dies lässt sich mit der nach $O(\log n)$ ansteigenden Tiefe der cTree-Baumdarstellung erklären).

Während sich $BASIC$ und $SECURE$ unter $PING\_0$ um kaum mehr als 15% unterschieden, wirkte sich der in $SECURE$ zusätzliche, aufgrund der einseitig verschlüsselten M:N-Relation notwendige Client-Server-Roundtrip unter $PING\_25$ stärker aus, so dass $SECURE$ hier eine ca. 45% längere Ausführungszeit aufwies.

Die maximale Dauer eines Einfügevorgangs für $PING\_25$ lag für $BASIC$ und $SECURE$ bei 187,7 bzw. 214,8 ms. Beide Latenzen liegen unterhalb dem Limit von 300 ms und sind somit tolerierbar.

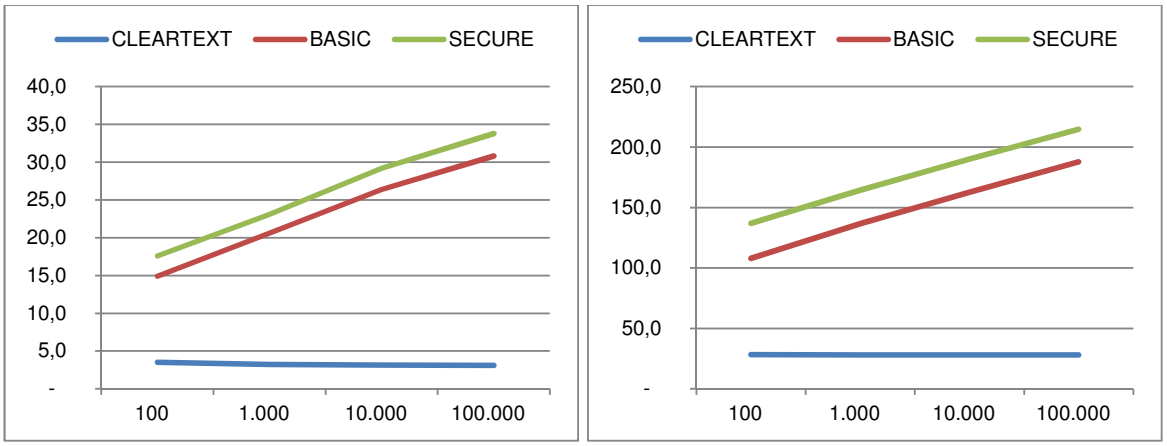

Abb. 14: Einfügeoperation bei 0 bzw. 25ms Latenz.

11.3.2 Gleichheitssuche

Bei den Tests zur Gleichheitssuche wurde in allen Implementierungsszenarios $CLEARTEXT$, $BASIC$ und $SECURE$ der gleiche Suchbegriff s gesucht, zu dem in der Extension der durchsuchten Relation jeweils genau 10 Tupel gefunden werden konnten. Im Falle von BASIC wurde ein konventionell indexiertes Attribut als Index eingesetzt, das den entsprechend Kapitel 5 normalisierten und definit verschlüsselten Attributwert des indexierten Attributs enthielt. Im Falle von $SECURE$ wurde dagegen wieder auf den bereits in Kapitel 11.1 genannten, redundant gespeichert und randomisiert verschlüsselten cTree gesetzt, der über eine einseitige, randomisiert verschlüsselte M:N-Relation die Datenrelation referenzierte.

Die in Abb. 15 dargestellten Ergebnisse zeigen, dass hier im Gegensatz zur Einfügung und zur Präfixsuche die Performance von $CLEARTEXT$ und $BASIC$ nah beieinander lag, also $BASIC$ annähernd so schnell lief wie $CLEARTEXT$, während $SECURE$ bei steigendem Datenvolumen eine ähnliche Charakteristik wie in Kapitel 11.3.1 aufwies.

Unter $PING\_0$ wirkte sich die aufgrund von zusätzlichen Normalisierungs- und Verschlüsselungsoperationen aufwändigere Verarbeitung noch deutlich aus, so dass $CLEARTEXT$ noch knapp 3-mal so schnell war wie $BASIC$. Unter $PING\_25$ fiel dieser Performanznachteil weniger ins Gewicht, so dass $CLEARTEXT$ bei T_100 lediglich 58% schneller war als $BASIC$ und 27% schneller bei $T\_100.000$.

Da *SECURE* die im Vergleich zum in *CLEARTEXT* und *BASIC* eingesetzten, konventionellen Datenbankindex die bereits im vorangegangenen Unterkapitel beschriebene, vergleichsweise weniger performante Technik des modifizierten cTree-Indexes einsetzte, war der Performanz-Nachteil hier höher: Unter *PING*_0 war *CLEARTEXT* zwischen 9- und 17-mal schneller als *SECURE*, und unter *PING*_25 6- bis 8,5-mal schneller. Dagegen war *BASIC* unter *PING*_0 2,5- bis 4-mal schneller als *SECURE* und unter *PING*_25 4- bis 6,5-mal so schnell.

Angesichts der absoluten Antwortzeiten von 33ms bis 60ms unter *PING*_0 bzw. 176ms bis 310ms unter *PING*_25 bleibt jedoch auch hier festzuhalten, dass *SECURE* eine akzeptable Performanz für den Betrieb eines Informationssystems lieferte.

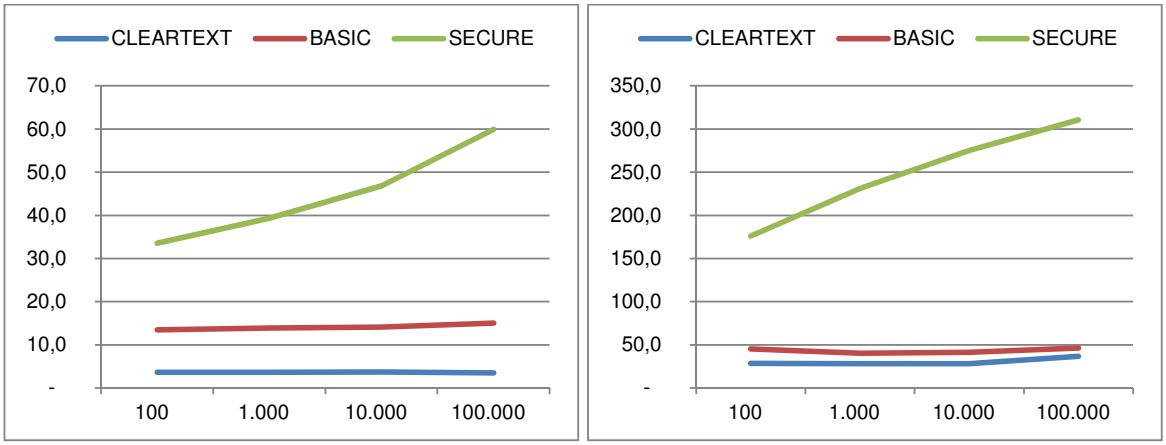

Abb. 15: Gleichheitssuche bei 0 bzw. 25ms Latenz.

11.3.3 Präfixsuche

Auch bei der Präfixsuche wurde in allen Testszenarios stets nach dem gleichen Suchbegriff gesucht, zu dem in der Datenbankextension stets eine Treffermenge von 10 Tupeln zu finden war. Neben der konventionellen Präfixsuche in *CLEARTEXT* wurde in *BASIC* die Präfixsuche angewendet wie in Kapitel 6.8 beschrieben. In *SECURE* wurde die gleiche Datenstruktur eingesetzt, die bereits in den Kapiteln 11.3.1 und 11.3.2 beschrieben wurde.

Die Ergebnisse der Tests unter *PING*_0 und *PING*_25 und *T*_100 bis *T*_100.000 sind in Abb. 16 dargestellt. Es fiel auf, dass die Laufzeiten von *CLEARTEXT* über *T*_100 bis *T*_100.000 nun nicht mehr konstant blieben, sondern wie die von *BASIC* und SECURE linear anstiegen (linear zur exponentiell wachsenden Datenmenge, also eigentlich logarithmisch), was damit erklärt werden kann, dass für Ungleichheitssuchen auch in *CLEARTEXT* baumbasierte Indexstrukturen mit logarithmisch wachsender Höhe zum Einsatz kommen.

Dennoch war *CLEARTEXT* auch hier erwartungsgemäß signifikant schneller als *BASIC* und *SECURE*: Unter *PING*_0 war *CLEARTEXT* etwa 3- bis 8-mal schneller als *BASIC* und 4- bis 10-mal schneller als *SECURE*, und unter *PING*_25 konstant etwa 5-mal schneller als *BASIC* und etwa 6-mal schneller als *SECURE*.

Weiterhin ist zu erwähnen, dass *BASIC* unter *PING*_0 lediglich 16% bis 18% schneller lief als *SECURE* und unter *PING*_25 16% bis 30% schneller, was einen beinah zu vernachlässigenden Performanzunterschied bedeutet.

Die absoluten Laufzeiten lagen im schlechtesten Fall, also bei $PING\_25$ und $T\_100.000$ bei 211ms für $BASIC$ und 306ms für $SECURE$ (gegenüber 47ms für $CLEARTEXT$), so dass auch hier das Echtzeitkriterium sogar für $SECURE$ als gerade noch erfüllt angesehen werden kann.

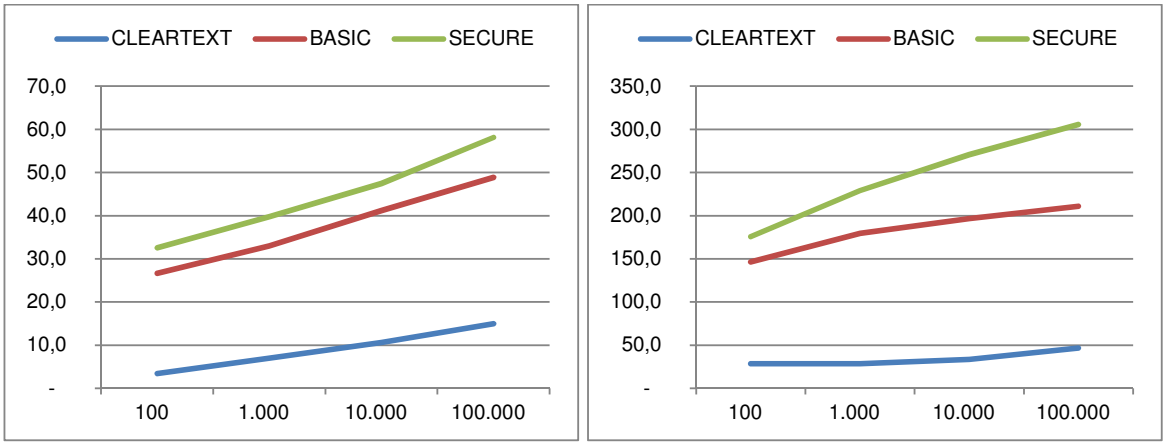

Abb. 16: Präfixsuche bei 0 bzw. 25ms Latenz.

11.3.4 *DISTINCT* und *REDUNDANT*

Neben den Variationen der Tests über Tupelanzahl und Client-Server-Latenz wurde bei den Tests mit der Beschaffenheit des Datenmaterials noch eine weitere Dimension eingeführt, über die die Tests variiert und ausgeführt wurden: Da unter der $BASIC$-Betriebsweise mehrere Tupel der Datenrelation auf dasselbe Indextupel verweisen, bewirkt ein einzufügender Datenbestand mit viel Redundanz ein geringeres Datenvolumen des Indexes, also eine geringere Baumhöhe und damit kürzere Antwortzeiten bei der Präfixsuche als bei einem Datenbestand mit ausschließlich eindeutigen Tupeln. Es sollte getestet werden, ob sich ein hoher Grad von Redundanz in signifikanten Laufzeitunterschieden in der Präfixsuche unter $BASIC$ auswirkt.

Dementsprechend wurden für die Testreihen zwei Basisdatenbestände herangezogen, für die die oben genannten Testreihen jeweils ausgeführt wurden. Der erste ($DISTINCT$) bestand ausschließlich aus eindeutigen Werten, während der andere ($REDUNDANT$) mit etwa 50% einen hohen Grad an Redundanz aufwies. Jedoch bewirkte dies lediglich eine Halbierung der Tupelanzahl des cTree-Indexes, was eine um eine Ebene niedrigere Baumdarstellung bedeutete. Die Ergebnisse der Präfixsuche unter BASIC wichen dementsprechend erkennbar, aber mit 15-20% dennoch nur moderat voneinander ab. Dies zeigt, dass ein gängiger Grad an Redundanz sich zwar bemerkbar macht, eine signifikante Auswirkung auf die Performanz sich aber erst bei hohen Graden einstellt, etwa bei einem großen Umfang der indexierten Datenrelation.

11.4 Variierende Teilbaumhöhe

Der wichtigste (und teuerste) Teilschritt bei der Verwendung des cTree-Indexes ist das Abrufen von Teilbäumen vom Server durch den Client, bei dem die Höhe h des Teilbaums wie in Kapitel 9.3 beschrieben parametrisiert werden kann. Die Größe des zum Client zu transportierenden Datenvolumens hängt exponentiell von h ab: Der Teilbaum enthält bis zu $2^h - 1$ Tupel, von denen für dessen clientseitige Traversierung lediglich h gebraucht werden, so dass stets $2^h - (h + 1)$ Tupel unnötigerweise transferiert werden. Diese Anzahl sollte durch Minimierung von h minimiert werden.

Gleichzeitig werden zur Traversierung der gesamten Baumdarstellung mit einer Gesamtbaumhöhe H bis zu $k = \left\lceil \frac{H}{h} \right\rceil$ aufeinanderfolgende Teilbäume benötigt, die jeweils einen Client-Server-Roundtrip benötigen. Die daraus resultierende Anzahl von k Roundtrips sollte durch eine Maximierung von h minimiert werden.

Somit manifestiert sich ein Trade-off für die Wahl von h, dessen optimale Lösung von Übertragungskapazität und Latenz der Netzwerkverbindung mitbeeinflusst wird. Eine Testreihe, die für $PING\_0$ und $PING\_25$ unter $T\_10.000$ in der $BASIC$-Ausführungsform eine von 1 bis 10 variierende Teilbaumhöhe (H_1 bis H_10) verwendete, ermittelte zum einen die durchschnittliche Dauer der Einfüge- und cTree-Indexierungsoperation eines Tupels und zum anderen die durchschnittliche Dauer einer Präfixsuche.

Die Ergebnisse der Testreihe sind in Abb. 17 dargestellt. Unter $PING\_0$ liegt die optimale Teilbaumhöhe für Einfügeoperation und Präfixsuche bei 5. Danach beginnt sich das Datenvolumen der übertragenen Teilbäume negativ auszuwirken. Unter $PING\_25$ steigt der Wert für die optimale Teilbaumhöhe aufgrund der höheren Roundtrip-Kosten für Einfügeoperation und Präfixsuche auf 7. Man kann davon ausgehen, dass dieser Wert bei einer noch höheren Latenz weiter steigt.

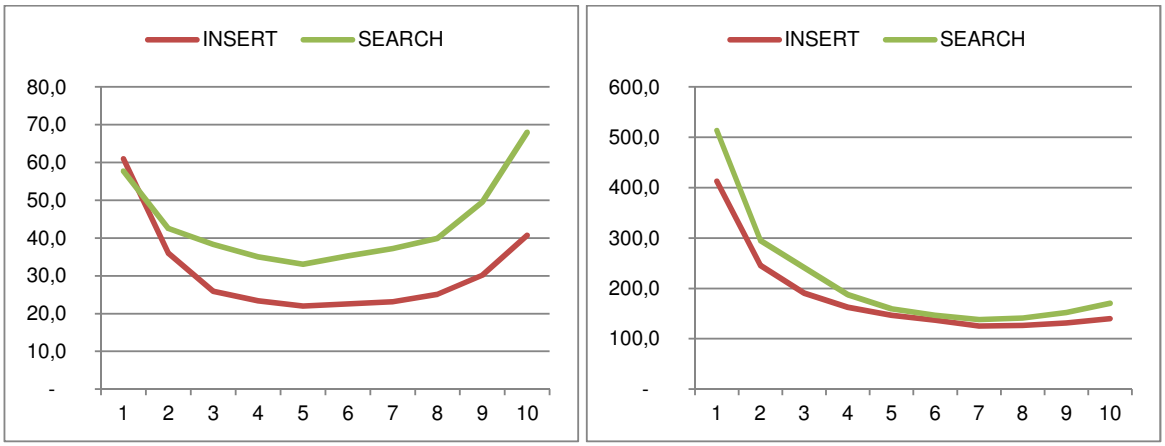

Abb. 17: Einfügeoperation und Präfixsuche bei variierender Teilbaumhöhe und 0 bzw. 25 ms Latenz.

11.5 Maßnahmen zur Performanzsteigerung

Nach der komparativen Analyse der Indexstrukturen in Kapitel 11.3 und der Analyse der Auswirkung von variierender cTree-Teilbaumhöhe in Kapitel 11.4 befasst sich dieses Unterkapitel mit den Auswirkungen der in den Kapiteln 9.6 und 9.7 eingeführten Maßnahmen zur Performanzsteigerung auf dem cTree-Index, also Pipelining und Caching.

Zum einen wurde in einer Testreihe die Präfixsuche im $BASIC$-Modus sowohl mit serverseitigem Caching als auch mit Pipelining für den Lesezugriff beschleunigt. Zum anderen wurde eine massiv wiederholte Einfügeoperation ($BULK\ INSERT$) auf einem redundanten Datenbestand[100] unter dem $PING\_25$-Szenario ausgeführt und mit clientseitigem Caching und Pipelining für den Schreibzugriff beschleunigt.

[100] Es handelte sich um 19.000 Vor- und Nachnamenskombinationen, bezogen von: siehe [URL-Name-DB].

11.5.1 Lesezugriff: Serverseitiges Caching und Pipelining

11.5.1.1 Serverseitiges Caching

Mit Hilfe des serverseitigen Cachings, bei dem Baumtraversierungsergebnisse auf der Middleware gespeichert werden (siehe Kapitel 9.7.1), ließ sich die Anzahl der Client-Server-Roundtrips bei der Präfixsuche auf zwei Roundtrips reduzieren: Bei der ersten Anforderung eines Teilbaums fragte die Middleware vor der Weiterleitung der Anfrage an das DBMS den Server-Cache an und gab im Falle eines Treffers anstelle des Teilbaums das Traversierungsergebnis an den Client zurück, der damit sofort den finalen SELECT generieren und an den Server senden konnte.

11.5.1.2 Pipelining

Durch eine asnychrone Implementierung der Präfixsuche im Client konnte die cTree-Traversierung zur Ermittlung des korrekten Indextupels für die obere und die untere Intervallgrenze quasi parallel angestoßen und mit ihren Zwischenergebnissen verarbeitet werden. Wie bereits in Kapitel 9.6.1 beschrieben muss man allerdings angesichts eines singulären Worker Threads im DBMS allerdings anstelle von echter Parallelität eher von Pipelining sprechen. In den Tests zeigte sich jedoch, dass das DBMS bei weitem nicht der Engpass der Architektur war, so dass dieser Aspekt die Performanzsteigerung nur wenig beeinträchtigte.

Pipelining war also aufgrund der asynchronen Implementierung bereits implizit vorhanden. Auch komplexere Suchanfragen wie etwa solche, die in der Selektionsbedingung mehr als eine Intervallsuche definieren, würden die Suche nur geringfügig verlangsamen, da auch sie im Pipelining-Modus ausgeführt würden.

11.5.1.3 Ergebnisse

Es zeigte sich im Test unter dem *PING*_25-Szenario, dass sich die durchschnittliche Dauer der Präfixsuche von etwa 150ms bis 200ms auf konstant etwa 70ms absenken ließ.

11.5.2 Schreibzugriff: Clientseitiges Caching und Pipelining

Für den Test der bulk insert-Operation wurde der clientseitige Cache aus Kapitel 9.7.2 eingesetzt, bei dem einmal abgerufene Teilbäume lokal auf dem Client in einem schnellen Speicher zwischengespeichert werden. Aufgrund von häufigen Rebalancierungsoperationen der cTree-Baumdarstellung, die durch die zahlreichen Einfügeoperationen auf dem cTree-Index zustande kamen, herrschte im clientseitigen Cache eine hohe Fluktuation, was seinen positiven Einfluss auf die Performanz verringerte. Dennoch profitierte diese spürbar vom Einsatz des Caches.

Gleichzeitig wurde der bulk insert in der Form durchgeführt, dass stets für 8 Elemente im Pipelining-Modus die designierte Einfügeposition in der cTree-Baumdarstellung ermittelt wurde. Anschließend wurden die 8 Elemente entsprechend Kapitel 9.6.2 in einer sortierten Reihenfolge dem DBMS zur Einfügung in die Indexrelation übergeben.

Die Testergebnisse sind in Tabelle 2 dargestellt. Durch den Caching-Einsatz verringerte sich die Einfügedauer mit gut 20% nur geringfügig. Dies ist, wie oben bereits erwähnt, darauf zurückzuführen, dass durch die ständige Evolution der Indexbaumextension Cache-Einträge schnell ungültig wurden und neu geladen werden mussten. Durch die acht quasi-parallelen

Einfügeoperationen verringerte sich die durchschnittliche Einfügedauer dagegen fast linear auf ein Achtel der Zeit, die eine sequenzielle clientseitige Verarbeitung benötigt hätte.

	Single	Pipelined (8)
Ohne Cache	290ms	38ms
Mit Cache	230ms	35ms

Tabelle 2: Effekte von Pipelining und clientseitigem Caching auf BULK INSERT.

Da pro Cache-Treffer ein Client-Server-Roundtrip eingespart werden kann, hat das clientseitige Caching neben der geringeren Ausführungszeit auch einen positiven Effekt auf die Anzahl von Client-Server-Roundtrips und auf das zu transferierende Datenvolumen. Tabelle 3 zeigt, dass die Anzahl der Serveraufrufe um 45% und das transferierte Datenvolumen um 67% verringert werden konnte.

	Ohne Cache	Mit Cache
Serveraufrufe	161.095	89.077
Datenvolumen (MB)	45	15

Tabelle 3: Effekt von clientseitigem Caching auf Serveraufrufe und Datenvolumen.

12 Zusammenfassung

12.1 Motivation

In dieser Dissertation wird Bezug auf Informationssysteme genommen, bei denen Speicherort und Datenverarbeitung (bzw. Teile davon) an externe Dienstleister ausgelagert werden können. Für kommerzielle derartige Systeme gelten hierbei nicht nur die Gesetze des Marktes, sondern auch gesetzliche Bestimmungen, die etwa in der Bundesrepublik Deutschland vom Bundesdatenschutzgesetz und dem Strafgesetzbuch festgelegt werden. Besonders Letzteres stellt eine hohe Hürde dar, denn in § 203 StGB werden Berufsgruppen (wie beispielsweise Ärzte) genannt, deren Angehörige einen besonderen Status als Geheimnisträger innehaben. Sie haften für die Vertraulichkeit der personenbezogenen Daten, die ihnen anvertraut wurden; in extremen Fällen von Verletzung der Vertraulichkeit können sogar Gefängnisstrafen gegen sie verhängt werden.

In einem solchen Szenario muss die Vertraulichkeit der Daten eines Informationssystems wirksam geschützt werden. Ein gängiges Mittel dafür sind rechtliche Behelfskonstrukte wie etwa Einverständniserklärungen der Eigentümer der Daten (z. B. die Patienten eines Arztes). Dies ist jedoch keine Schutzmaßnahme, sondern lediglich ein Schutzverzicht der Eigentümer. Auch administrative Maßnahmen sowie Strafandrohung für korruptes Rechenzentrumspersonal bei Privilegienmissbrauch lösen das Problem nicht.

Der Schutz der Vertraulichkeit sollte vielmehr durch Maßnahmen sichergestellt werden, die im Hoheitsbereich des Systemanwenders durchgeführt, verwaltet und kontrolliert werden können. Ein naheliegender Ansatz ist daher die Verschlüsselung der Daten, bei der alle zugehörigen Schlüssel nur dem Anwender bekannt sind und Daten die nicht vertrauenswürdige Serverseite nur in verschlüsselter Form erreichen. Erst nach dem Abrufen vom Server und dem Wiedereintritt in den Client werden sie wieder in ihre unverschlüsselte Form überführt, so dass es auf der Serverseite keine Möglichkeit gibt, die Daten in Klartextform einzusehen.

Da die herkömmlichen Methoden der beschleunigten serverseitigen Suche auf verschlüsselten Daten nicht einsetzbar sind, werden alternative Methoden benötigt. Für diese Aufgabe gilt das Paradigma, dass alle Indexstrukturen, die dafür auf dem Server angelegt werden, dem selben Schutzniveau unterliegen wie die eigentlichen Daten. Somit müssen auch alle serverseitigen Indexstrukturen verschlüsselt werden.

In dieser Dissertation werden Methoden und Techniken zur Erstellung und Verwendung von verschlüsselten Indexstrukturen vorgestellt, um verschiedene Suchtypen auf den Daten eines verschlüsselten Informationssystems beschleunigt durchführen zu können. Dabei werden auch die Stärken konventioneller Datenbankindexe ausgenutzt.

Angesichts des hohen Praxisbezugs der Dissertation gilt es, den primären Fokus auf häufig verwendete Suchtypen zu legen. Somit werden die Gleichheitssuche, Intervallsuche, Präfixsuche (bzw. in verallgemeinerter Form Infixsuche) und Volltextsuche für die Erstellung verschlüsselter Indexstrukturen mit maximiertem Schutz der Vertraulichkeit und maximierter Performanz ausgewählt.

12.2 Indexe für die Suche auf Gleichheit

Beim ersten untersuchten Suchtyp, der Gleichheitssuche, wird festgestellt, dass Gleichheit je nach zugrunde liegendem Datentyp unterschiedlich interpretiert werden kann, was sich auch auf die jeweilige Implementierung der Gleichheitssuche auswirkt. Entsprechendes ist daher auch für die Suche auf verschlüsselten Daten erforderlich: Für unterschiedliche Datentypen werden unterschiedliche Vorgehensweisen entwickelt und präsentiert; allen gemein ist jedoch die Ausnutzung der Eigenschaften von deterministischer bzw. definiter Verschlüsselung.

12.2.1 Bitweise Gleichheit auf diskreten numerischen Datentypen

Bei diskret numerischen Datentypen wie *integer* und *decimal* sowie beim booleschen Datentyp *bit* bzw. *boolean* erweist sich die Implementierung semantischer Gleichheit als einfach: Zwei Werte werden als gleich angesehen, wenn ihre Binärdarstellungen bitweise übereinstimmen; also gilt dies auch für die Kryptotexte zweier solcher definit verschlüsselter Werte. Dies lässt sich ausnutzen, indem eine Gleichheitssuche auf einem definit verschlüsselten Attribut a' eines dieser Datentypen derart ausgeführt wird, dass der Suchbegriff s ebenfalls definit zu s' verschlüsselt und in der Extension von a' unter Verwendung konventioneller Datenbankindexe gesucht wird.

12.2.2 Bitweise Gleichheit auf binären Datentypen

Die Definition von Gleichheit auf binären Datentypen unterscheidet sich zunächst nicht von der in Kapitel 12.2.1 gegebenen. Jedoch können Instanzen dieser Datentypen in der Praxis oft groß und bitweise Vergleiche darauf teuer werden, weshalb es sich für eine praktische Implementierung anbietet, die Gleichheitssuche auf den definit verschlüsselten Hashwerten dieser Werteinstanzen durchzuführen. Auch hier konnte die Suche von konventionellen Datenbankindexen profitieren, unter einer in den meisten Fällen vernachlässigbar geringen Gefahr von false positives durch Kollisionen von Hashwerten nichtgleicher Klartextwerte.

12.2.3 Normalisierung von zeichenkettenbasierten Datentypen

Attribute mit zeichenkettenbasierten Datentypen können eine Sortiervorschrift (Collation) für ihre Werteinstanzen haben. Solche Collations, z. B. die Norm DIN-5007-2, müssen nachgebildet werden, um auch bei diesen Datentypen die Vorteile definiter Verschlüsselung ausnutzen zu können. Es wurde eine Abbildungsfunktion f eingeführt, für die postuliert wurde, dass sie alle Eigenschaften der jeweiligen Collation nachbildet, und die auf alle Werteinstanzen angewendet wurde. Deren Funktionswerte wurden anschließend analog zu den Kapiteln 12.2.1 und 12.2.2 definit verschlüsselt und konventionell indexiert zusammen mit den Originalwerten gespeichert. Gleichheitssuchen wurden dann auf diesen verschlüsselten Funktionswerten ausgeführt.

12.2.4 Schwellwertunterschreitung bei Gleitkommazahlen

Schon im Klartextbereich ist auf Gleitkommazahl-Datentypen nur eine numerische Approximation an die Gleichheitsrelation möglich, etwa in Form der ε-Methode, nach der zwei Werte als gleich angesehen werden, wenn ihr Abstand geringer ist als ein definierter Wert ε: x „=“ $y \Leftrightarrow |x - y| < \varepsilon$. Es wurde eine verschlüsselte Variante der ε-Methode entwickelt, bei der die Vorteile definiter Verschlüsselung ausgenutzt werden können. Danach wird für einen Gleitkommazahlwert x auch das folgende Tripel berechnet und gespeichert:

$$(x_{low}, x_{high}, x_{rel}) = \left(Enc_{det}\left(\left\lfloor\frac{x}{\varepsilon}\right\rfloor \varepsilon\right), Enc_{det}\left(\left(\left\lfloor\frac{x}{\varepsilon}\right\rfloor + 1\right)\varepsilon\right), x - \left\lfloor\frac{x}{\varepsilon}\right\rfloor \varepsilon\right) \quad \textbf{(F78)}$$

Die Extensionen dieser zusätzlichen Werte können konventionell indexiert und damit schnell durchsucht werden. Die ε-Methode kann dann folgendermaßen auf verschlüsselten Daten nachgebildet werden:

$$(x'_{low} = y'_{low}) \vee (x'_{low} = y'_{high} \wedge x_{rel} < y_{rel}) \vee (x'_{high} = y'_{low} \wedge x_{perc} > y_{rel})$$

$$\Leftrightarrow x_{low} = y_{low} \vee (x_{low} = y_{high} \wedge x_{rel} < y_{rel}) \vee (x_{high} = y_{low} \wedge x_{rel} > y_{rel})$$

$$\Leftrightarrow |x - y| < \varepsilon \quad \textbf{(F79)}$$

12.3 Indexe für die Suche auf Ungleichheit – der cTree-Index

Neben der Gleichheitssuche ist die Ungleichheits- bzw. Intervallsuche einer der am häufigsten angewendeten Suchtypen. Eine besondere Anwendungsform der Intervallsuche ist darüber hinaus die Präfixsuche auf lexikografisch geordneten Daten, welche ihrerseits einen oft verwendeten Suchtyp darstellt: Jede Präfixsuche lässt sich in eine Intervallsuche umformen und somit mit deren Mitteln ausführen.

Um die Intervallsuche auf dem Server zu unterstützen, wird eine serverseitige Offenlegung der linearen Ordnung der indexierten Elemente in Kauf genommen, wobei der Einsatz der hierfür grundsätzlich in Frage kommenden OPE-Verschlüsselungsverfahren vermieden wird. Deshalb wird mit der cTree-Indexierung ein verschlüsseltes Verfahren zur Indexierung linear geordneter Daten entwickelt, das hinsichtlich der Verschlüsselung seiner Indexelemente den Einsatz beliebiger Verfahren gewährt.

12.3.1 Die cTree-Datenstruktur

Kern des Verfahrens ist die cTree-Datenstruktur, die die Indexelemente in verschlüsselter Form in einer konventionellen Datenbankrelation speichert. Sie besitzt die folgenden Eigenschaften:

- Jedes Indexelement entspricht einem Tupel der Indexrelation. Die Tupel der indexierten Datenrelation referenzieren über eine Fremdschlüsselbeziehung die Tupel der Indexrelation.
- Der eigentliche Indexwert kann über ein beliebiges Verfahren verschlüsselt werden. Es eignen sich sowohl definite als auch randomisierte Verfahren.
- Über selbstreferenzierende Fremdschlüsselattribute werden die Tupel der cTree-Relation in einer binären Baumdarstellung angeordnet, die die lineare Ordnung der Elemente nachbildet. In einer praktischen Implementierung wurde hierfür der AVL-Baum gewählt.
- Die cTree-Relation ist mit einem weiteren Attribut $sortNumber$ ausgestattet, dessen Extension ebenfalls die lineare Ordnung der Indexwerte nachbildet. Dabei geben die Elemente der Extension abgesehen von der Ordnung keine Information über die Indexwerte preis.

12.3.2 Einfügeoperation

Bei der Einfügung eines neuen Elements in den cTree-Index muss das Element in die beiden Repräsentationen der linearen Ordnung der Indexelemente einsortiert werden. Dazu wird zunächst die binäre Baumdarstellung traversiert, bei der Wurzel des Baumes startend. Jedes Element des Traversierungspfades muss zum Client transferiert und dort entschlüsselt werden. Der Client vergleicht den Wert des Traversierungselements mit dem des einzufügenden Elements und entscheidet, ob er zum linken oder zum rechten Nachfolgerknoten wechselt. Dies wird wiederholt, bis entweder die Einfügeposition gefunden oder festgestellt wurde, dass der einzufügende Wert bereits im Index enthalten ist.

Da jede Iteration auf dem Traversierungspfad einen Client-Server-Roundtrip bedeutet, können auch größere Teilmengen des Baumes, also Teilbäume, zum Client transferiert und dort durchlaufen werden, um Roundtrips einzusparen.

Anschließend wird ein Wert für das $sortNumber$-Attribut für das neue Element derart gewählt, dass es sich auch korrekt in die strikt lineare Nachbildung der linearen Ordnung der Indexelemente einfügt. Nachdem die nun vollständige Einfügeoperation durchgeführt worden ist, wird der binäre Baum gegebenenfalls balanciert, so dass dessen AVL-Bedingung wieder erfüllt ist.

12.3.3 Intervallsuche

Eine Suche auf dem cTree-Index nach allen Werten, die einem Intervall liegen, besteht zunächst aus zwei separaten Suchvorgängen nach denjenigen Indexelementen, die die obere bzw. untere Intervallgrenze repräsentieren. Ein solcher Suchvorgang ähnelt der Einfügeoperation; jedoch wird dort nicht auf Gleichheit bzw. nach der korrekten Einfügestelle gesucht, sondern nach der bestmöglichen Approximation für den angegebenen Suchwert. D. h., dem kleinsten Element in der Indexextension, das größer oder gleich der unteren Intervallgrenze ist und dem größten Element kleiner oder gleich der oberen Intervallgrenze.

Die beiden Intervallgrenzen werden anschließend verwendet, um die eigentliche Treffermenge der Intervallsuche zusammenzustellen, indem eine Datenbankanfrage alle Indextupel selektiert, die zwischen den Intervallgrenzen liegen. Als Selektionskriterium der Anfrage wird die zweite, strikt lineare Darstellung der linearen Ordnung im cTree herangezogen, die im Attribut $sortNumber$ gespeichert ist; sie profitiert von konventionellen Datenbankindexen, so dass sie beschleunigt ausgeführt werden kann.

12.3.4 Präfixsuche

Auf der lexikografisch geordneten Menge aller Wörter $w \in \Sigma^*$ über dem Alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ lässt sich ein regulärer Ausdruck $w\Sigma^*$, der eine Präfixsuche definiert, zu einem halboffenen Intervall $[w, inc(w)[$ umformen. Nach der Umformung lässt sich eine Intervallsuche wie in Kapitel 12.3.3 beschrieben auf dem cTree-Index anwenden.

12.4 Die Shrinking Window-Erweiterung

In Kapitel 7 wird gezeigt, dass sich die cTree-Indexdatenstruktur modifizieren lässt, so dass damit nicht nur Präfix-, sondern auch flexibel parametrisierbare Infixsuchen auf lexikografisch geordneten Daten möglich sind. Dazu sind Änderungen an der Struktur nötig:

- Anstatt der Originalwerte der indexierten Datenrelation speichert der cTree-Index alle Suffixe dieser Werte.
- Die M:N-Relation, die die cTree-Indexrelation mit der indexierten Datenrelation verbindet, speichert in jedem Tupel zwei zusätzliche Werte:
    - an welcher Stelle der in der Indexrelation referenzierte Suffix von seinem Originalwert „abgeschnitten“ worden ist und
    - die Länge des Originalwerts.

Zusammen mit einigen kleineren Änderungen in den DML- und DRL-Operationen INSERT, UPDATE, DELETE und SELECT lassen sich mit dieser modifizierten Indexstruktur Infixsuchen nach einem Suchterm s auf verschlüsselten Daten durchführen. Dabei kann die Anzahl von Zeichen vor bzw. nach s entweder beliebig groß oder explizit angegeben sein, so dass Suchen entsprechend den folgenden regulären Ausdrücken durchführbar sind:

- (Gleichheitssuche: s)
- Präfixsuche: $s\Sigma^n$, $s\Sigma^*$
- Postfixsuche: $\Sigma^n s$, $\Sigma^* s$
- Infixsuche: $\Sigma^m s\Sigma^n$, $\Sigma^n s\Sigma^*$, $\Sigma^* s\Sigma^n$, $\Sigma^* s\Sigma^*$

Durch die Kombination mehrerer der oben genannten Suchtypen sind darüber hinaus weitere Suchtypen möglich, die komplexeren regulären Ausdrücken entsprechen.

12.5 Volltext-Indexe auf verschlüsselten Daten

Mit dem verschlüsselten Volltextindex wird in Kapitel 8 ein weiterer Indextyp für die beschleunigte Suche auf verschlüsselten Daten vorgestellt, wobei es sich hierbei nicht um eine weitere Indexierungstechnik handelt, sondern eher um eine Anwendung der in den vorangegangenen Kapiteln 5 bis 7 vorgestellten Techniken. Die Volltextindexierung eines Eingabewertes wird dabei in zwei Phasen unterteilt: zuerst dessen Zerlegung in Tokens bzw. Sub-Tokens und anschließend deren Indexierung. Für beide Phasen stehen unterschiedliche Implementierungsansätze zur Verfügung, die in Kapitel 8 untersucht werden.

12.5.1 Token-Zerlegung

Die Tokenisierung eines Eingabe-Freitextes wird in zwei Stufen unterteilt: Danach wird der Eingabewert in einer ersten Stufe nach einer simplen Methode wie beispielsweise der Whitespace-Tokenisierung in Tokens zerlegt. Auf letztere wird dann in einer zweiten Stufe die eigentliche Tokenisierung in die sogenannten Sub-Tokens angewendet, wofür sich die folgenden Implementierungsansätze anbieten:

- Naiver Ansatz: Die Ergebnisse der ersten Tokenisierungsstufe werden ohne weitere Zerlegung als Eingabewerte für die Indexierung in der zweiten Stufe verwendet.
- n-Gramme: Zu jedem Token werden alle n-stelligen Fragmente ermittelt, um sie in der zweiten Phase zu indexieren.
- Suffixe: Zu jedem Token werden alle Suffixe ermittelt, um sie in der zweiten Phase zu indexieren.

- Stemming: Auf die Tokens werden höher entwickelte Verfahren angewendet, die ein Token, oft eine morphologische Variante eines Wortes, auf seinen Wortstamm abbilden.

12.5.2 Indexierung

Bezüglich der Indexierung wird in Kapitel 8 zwischen den beiden Indexierungstechniken aus Kapitel 5 und 6 unterschieden: der Indexierung für die Suche auf Gleichheit und der für die Intervall- bzw. Präfixsuche. Jeder in Kapitel 12.5.1 genannte Tokenisierungsansatz wird hinsichtlich seiner Eignung für die jeweilige Indexierungstechnik bewertet.

Dabei erweist sich die Indexierung für die Suche auf Gleichheit als mit allen vier in Kapitel 12.5.1 genannten Tokenisierungsansätzen kombinierbar. Für die Indexierung für die Präfixsuche eignen sich dagegen lediglich der naive und der Suffix-Ansatz uneingeschränkt; der Stemming-Ansatz zeigt sich nur eingeschränkt und der n-Gramm-Ansatz überhaupt nicht für die Präfixsuche einsetzbar.

12.6 Performanz-Optimierung des cTree-Indexes

Für die Basis-Ausführungsform der cTree-Indexdatenstruktur werden in Kapitel 9 eine Reihe von Maßnahmen zur Performanzsteigerung vorgestellt.

12.6.1 Triviale Maßnahmen

Diese beinhalteten naheliegende Maßnahmen wie die Definition konventioneller Datenbankindexe auf geeigneten Attributen der cTree-Indexrelation und gezielten Einsatz redundanter Attribute und materialisierter Information in der cTree-Datenstruktur.

12.6.2 Weiterführende Maßnahmen

Bei der Implementierung der binären Baumdatenstruktur des cTree-Indexes fällt die Wahl auf den AVL-Baum, da er eine gute Balance zwischen niedrigem Verwaltungsaufwand und Performanz bietet.

Bezüglich der Höhe h der beim cTree-Zugriff abgerufenen Teilbäume wird das Trade-off „Minimale Anzahl von Client-Server-Roundtrips vs. Minimale Anzahl von unnötig transferierten Knoten“ untersucht. Es gibt demnach für h einen optimalen Wert, der von äußeren Umständen wie Client-Server-Latenz und Bandbreite beeinflusst wird. In der cTree-Implementierung in Kapitel 11 wird beispielhaft ein optimaler Wert für h empirisch ermittelt.

Eine weitere Maßnahme zur Verbesserung der Performanz ist das Ausnutzen definiter Verschlüsselung im cTree-Index. Hier kann beim Einfügen neuer Elemente eine Prüfung (die auf der in Kapitel 5 behandelten Gleichheitssuche basiert) auf der definit verschlüsselten Indexextension erfolgen, ob sich der Kryptotext des neuen Wertes bereits im Index befindet. Falls dem so ist, kann eine erneute Einfügung des Wertes entfallen.

12.6.3 Pipelining

Ein wichtiger Aspekt bei der Steigerung der Performanz des cTree-Indexes ist Parallelisierung, wobei beim cTree-Index korrekterweise eher von Pipelining gesprochen werden muss. Hier bieten sich die folgenden beiden Ansätze an:

12.6.3.1 Pipelining beim Lesezugriff

Eine Intervallsuche auf dem cTree-Index führt für die untere und obere Intervallgrenze jeweils eine cTree-Traversierung durch. Bei einer Datenbankanfrage, die n Intervallsuchen beinhaltet, wächst die Anzahl der Traversierungen auf bis zu $2n$ an. Da diese Operationen ausschließlich lesend und unabhängig voneinander sind, können sie parallel an den Server gesendet und dort im Pipelining-Modus abgearbeitet werden. Die Verarbeitung durch das DBMS erfordert den geringsten Anteil an der Gesamtdauer einer Intervallsuche; daher dauert eine Anfrage mit n Intervallsuchen nur unwesentlich länger als eine Anfrage mit nur einer Intervallsuche.

12.6.3.2 Pipelining beim Schreibzugriff

Parallele bzw. Pipelining-Verarbeitung beim Schreibzugriff auf den cTree-Index kann beispielsweise bei großen Mengen von Einfügeoperationen (Bulk Insert) erfolgen. Hier können Kollisionen auftreten, wenn zwei oder mehr neue Knoten als dasselbe Kind desselben Knoten eingefügt werden sollen. Für diesen Fall wird eine Modifikation der AVL-Baum-Algorithmen entwickelt, die hilft, diese Kollisionen aufzulösen und die massenhafte Einfügung in den cTree-Index im Pipelining-Modus zu ermöglichen.

12.6.4 Caching

Sowohl für die Client- als auch für die Serverseite wurden Caching-Verfahren entwickelt, die die Performanz des cTree-Indexes erhöhen.

12.6.4.1 Serverseitiges Caching

Auf der Serverseite bietet es sich in einem ersten naiven Ansatz an, bereits ermittelte Treffermengen zu bestimmten Intervallsuchanfragen zwischenzuspeichern, so dass nachfolgende Anfragen weder die beiden cTree-Traversierungen noch die nachfolgende finale SELECT-Operation durchführen müssen. Da die hierfür anfallenden Datenmengen schnell groß werden können, empfiehlt sich eine alternative, kompaktere Caching-Methode: Anstatt des tatsächlichen Anfrageergebnisses wurde der zur jeweiligen Intervallgrenzen-Tupel gehörende $sortNumber$-Attributwert im Cache gespeichert. Eine Löschstrategie für ungültig gewordene Cache-Einträge vervollständigt den Ansatz für serverseitiges Caching.

12.6.4.2 Clientseitiges Caching

Bei clientseitigem Caching werden Teilbäume, die bei vorherigen Zugriffsoperationen vom Server abgerufen worden sind, im Client bis zur nächsten Verwendung zwischengespeichert. Eine besondere Herausforderung stellt hier die möglichst effiziente Löschung ungültig gewordener Teilbäume dar. Ein naives Verfahren, das bei jeder Änderung des cTree-Indexes die Löschung aller clientseitigen Caches veranlasst, kommt nicht in Frage; stattdessen wird mit dem Rotationspunkt-Ansatz ein Verfahren entwickelt, das nach einer Einfügung in den cTree-Index die Löschung von nur denjenigen Cache-Einträgen veranlasst, die auch tatsächlich durch die Indexänderung ungültig geworden sind.

Dabei wird die Eigenschaft von ausschließlich wachsenden AVL-Bäumen ausgenutzt, dass bei ihrer Rebalancierung nach einer Einfügung maximal eine Rotationsoperation durchgeführt wird. Der Knoten, auf dem die Rotation stattfindet, wird an alle an den Server angeschlossenen Clients gesendet, welche daraufhin die von der Rotation direkt oder indirekt betroffenen und daher ungültig gewordenen Cache-Einträge löschen und alle anderen unberührt im Cache belassen.

12.6.5 Transaktionales Konzept

Ein Informationssystem, das die in dieser Dissertation vorgestellten Indexstrukturen nutzt, ist typischerweise ein webbasiertes System. Es werden deutlich mehr lesende als schreibende Zugriffe auf das System erwartet, und es steht zu erwarten, dass Clientverbindungen zum Server oft ungeplant terminiert werden. Aufgrund letzterem sollten clientgesteuerte Transaktionen vermieden werden. Gleichzeitig wird eine hohe Verfügbarkeit insbesondere für die transaktional gekapselten Zugriffsoperationen auf die verschlüsselten Indexstrukturen gefordert. Deshalb wird in Kapitel 9.8 ein leichtgewichtiges, optimistisches, quasi-transaktionales Konzept eingeführt, das auf Versionsnummern basiert. Dabei wird jeder Indexinstanz eine Versionsnummer zugeordnet, die beim Beginn jeder Indexoperation des Informationssystems abgefragt und zum Schluss dieser Operation inkrementiert wird. Eine nebenläufige Veränderung auf der Indexversion bewirkt, dass die Transaktion zurückgerollt wird. Da dieses Konzept schreibenden Zugriffsoperationen einen Vorteil verschafft, wird lesenden Operationen die Möglichkeit des Setzens eines „override“-Flags eingeräumt, wenn ihr Zugriff zu oft von schreibenden Operationen verhindert wird.

12.7 Sicherheitsaspekte

In Kapitel 10 werden die vorgestellten verschlüsselten Indexstrukturen hinsichtlich preisgegebener Information, daraus resultierender Angriffsmöglichkeiten und Abwehrmaßnahmen dagegen betrachtet.

12.7.1 Informationsverlust und Angriffsmöglichkeiten

Zunächst werden die Formen des Informationsverlusts behandelt, die bei den vorgestellten verschlüsselten Indexstrukturen auftreten können. Dies umfasst die folgenden Aspekte:

- Häufigkeitsinformation durch die Verwendung von definiter Verschlüsselung.
- Offenlegung der linearen Ordnung der Elemente eines cTree-Index.
- Klartextverknüpfungen zwischen Index- und Datentupeln durch die M:N-Relation.
- Hinweise auf die Längen indexierter Worte im „shrinking window“-Betriebsmodus.

Die oben beschriebenen Formen des Informationsverlustes eröffnen verschiedene Angriffsmöglichkeiten, die grob in statische und dynamische Angriffe kategorisiert wurden, wobei der Fokus auf statischen Angriffen liegt. Dabei handelt es sich im Einzelnen um:

- Häufigkeitsanalysen auf definit verschlüsselten Daten sowie auf mehrfach durch die M:N-Relation referenzierten Indextupeln.
- Ableitung der Längen von Zeichenketten im „shrinking window“-Betriebsmodus.
- Ausnutzen der linearen Ordnung im Rahmen von known-ciphertext-Angriffen.

12.7.2 Gegenmaßnahmen

Um die oben genannten Angriffsmöglichkeiten zu vermeiden oder zumindest signifikant abzuschwächen, werden die folgenden Modifikationen der Indexstrukturen eingeführt:

- Verzicht auf definite Verschlüsselung; stattdessen Verwendung der cTree-Datenstruktur für die Gleichheitssuche.

- Randomisierte Verschlüsselung der Fremdschlüsselbeziehungen zwischen M:N- und Datenrelation unter Inkaufnahme zusätzlicher Client-Server-Roundtrips und erhöhtem Datentransfervolumens.
- Redundante Speicherung verschlüsselter Indexelemente zur Vermeidung von Häufigkeitsanalysen auf der cTree-Indexrelation.
- Zyklische Verschiebung des Attributwertebereichs bei numerischen Attributen zur Verschleierung von Häufigkeitscharakteristika der Extension.
- Verschleierung der Wortlängen im „shrinking Window"-Modus.
- *k*-anonymity: Füllen der Datenbank mit Dummy-Werten, um Werteverteilungscharakteristika zu verschleiern.

Weiterhin werden noch gängige Maßnahmen gegen dynamische Angriffe genannt wie Padding von Daten auf eine einheitliche Blockgröße sowie die Einführung zeitlicher Inkohärenz bei ihrem Versand. Abschließend wird Onion Routing zur Anonymisierung von Verbindungsdaten beim Clientzugriff vorgeschlagen.

12.8 Performanzmessungen

In Kapitel 11 werden die Indexstrukturen unterschiedlichen Performanzmessungen unterzogen. In einer ersten Testreihe wird eine Implementierung der Indexstrukturen in ihrer Basisausführung mit einer Variante mit erhöhter Sicherheit, aber geringerer Performanz gemäß Kapitel 10 verglichen. Eine Klartext-Referenzimplementierung komplettiert die Testreihe, in der die Einfügeoperation in den cTree-Index sowie die Gleichheits- und die Präfixsuche unter variierenden Datenvolumina und variierender Client-Server-Latenz getestet werden. Die Ergebnisse der Testreihe zeigen, dass die beiden verschlüsselten Varianten zwar langsamer sind als die Klartextvariante; jedoch zeigen ihre absoluten Laufzeiten gute Performanz, die den in Kapitel 1.4.4 formulierten Anforderungen an ein Informationssystem gerecht wird. Insbesondere gibt es kein Testszenario, in dem die festgesetzte Maximallaufzeit von 300ms weit überschritten wird. Weiterhin zeigt die Variante der Indexstrukturen mit erhöhter Sicherheit erwartungsgemäß eine schwächere Performanz als die Basisimplementierung, doch der Overhead nimmt sich, abgesehen von der Gleichheitssuche, moderat aus.

In einer zweiten Testreihe wird die in Kapitel 9.3 beschriebene Variierung der Teilbaumhöhe anhand der Einfügeoperation in den cTree-Index und der Präfixsuche mit unterschiedlichen Client-Server-Latenzen analysiert. Es zeigt sich, dass sich für beide Operationen bei der größeren Latenz die optimale Teilbaumhöhe deutlich erhöht.

Die dritte Testreihe analysiert die in Kapitel 9 behandelten Maßnahmen zur Steigerung der Performanz. Zuerst wird im Lesezugriff serverseitiges Caching und Pipelining getestet: Hier lässt sich die Dauer der Präfixsuche von 150ms bis 200ms auf konstant etwa 70ms absenken.

Anschließend werden die Auswirkungen von clientseitigem Caching in Verbindung mit Pipelining im massiv ausgeführten Schreibzugriff getestet. Es zeigt sich, dass das clientseitige Caching bei der Ausführungsgeschwindigkeit seine Stärken nicht ganz so sehr ausspielen kann wie bei vorwiegend lesenden Zugriffen, da Cache-Einträge durch die ständige Evolution der Indexbaumextension relativ schnell ungültig werden. Somit kann hier bei der Ausführungszeit lediglich ein Performanzgewinn von 20% erzielt werden. Die Anzahl der Serveraufrufe verringert sich dabei auf etwas mehr als die Hälfte und das transferierte Datenvo-

lumen auf ein Drittel. Dagegen skaliert das Pipelining fast linear, so dass acht quasi-parallele Einfügeoperationen nur wenig mehr Zeit benötigten als eine singuläre Einfügeoperation. Durch eine Kombination von Caching und Pipelining kann die durchschnittliche Ausführungsgeschwindigkeit auf mehr als das Achtfache erhöht werden.

12.9 Beitrag zum Stand der Wissenschaft

Abschließend fasst dieses Unterkapitel den geleisteten Beitrag zum Stand der Wissenschaft zusammen. Die Dissertation ist praxisorientiert; ihr Fokus liegt nicht auf theoretischer Basisarbeit, sondern auf der Entwicklung und Ausarbeitung einer breiten Auswahl verschlüsselter Indexstrukturen, die in webbasierten Informationssystemen einsetzbar sind und den Anforderungen an solche Systeme hinsichtlich Vertraulichkeit, Verfügbarkeit und Integrität, aber auch Performanz, Skalierbarkeit und Mehrbenutzerfähigkeit genügen.

Unter dieser Zielsetzung ist eine Sammlung von Techniken zur verschlüsselten Indexierung ebenfalls verschlüsselter Daten entstanden, die sich zunächst in die folgenden beiden Hauptkategorien einordnen lassen:

- Gleichheitssuche auf Basis definiter Verschlüsselung und
- Ungleichheitssuche auf Basis der eigens eingeführten cTree-Datenstruktur.

Die Sammlung zeichnet sich in beiden Kategorien einerseits durch eine Beschreibung der Basisfunktionalitäten und andererseits durch eine Ausarbeitung auf unterschiedliche Anwendungsszenarien aus. Dabei ist die Basisidee, definite Verschlüsselung zur Implementierung einer Gleichheitssuche zu verwenden, nicht neu; sie taucht in zahlreichen Publikationen wie etwa [Popa2012] auf[101]. In Kapitel 5 dieser Dissertation wird der Ansatz jedoch für die im SQL-92-Standard enthaltenen Basisdatentypen nach einer detaillierten Betrachtung der unterschiedlichen Interpretation von Gleichheit ausdifferenziert.

Analog dazu wird die cTree-Indexdatenstruktur in Kapitel 6 für alle Datentypen eingeführt, die einer linearen Ordnung unterliegen. In [Lehnhardt2014-1] wurde die Basisidee hierzu publiziert. Die grundsätzliche Verwendung beider Indexierungsarten wird anhand der Basisoperationen INSERT, UPDATE, DELETE und SELECT definiert. Weiterhin werden verschiedene Performanztests durchgeführt.

In [Popa2013] wird eine Datenstruktur beschrieben, die dem cTree ähnlich ist, jedoch ohne viel Wert auf ihre praktische Anwendbarkeit zu legen. Stattdessen wird dort die Untersuchung der theoretischen Sicherheit des Ansatzes verfolgt. Die Autoren kommen zu der Erkenntnis, dass ihre Datenstruktur im Gegensatz zu anderen OPE-Verfahren keine Klartextbits verrät (ein Ergebnis, das sich auf den cTree übertragen lässt). Der Einsatz der Datenstruktur in einem verschlüsselten Informationssystem sowie sich daraus ergebende Probleme werden in [Popa2013] kaum betrachtet. In Kapitel 6 dieser Dissertation werden dagegen neben der detaillierten Beschreibung von Aufbau, Funktionsweise und den oben genannten Basisoperationen viele Aspekte der cTree-Indexstruktur ausgearbeitet, von denen ihre praktische Einsetzbarkeit profitiert:

- Ihre Anwendbarkeit als Präfixsuche auf lexikografisch geordneten Daten.

[101] siehe [Popa2012], S. 105.

- Die Einbeziehung von spezifischen Normalisierungsfunktionen, um Sortiervorschriften (Collations) auf den lexikografisch geordneten Daten zu implementieren.
- Spezielle Betriebsmodi, die im Praxisbetrieb von Bedeutung sein können:
    - Einsatz definiter Verschlüsselung zur Beschleunigung des Lookup-Schritts.
    - Anwendung der cTree-Struktur zur Implementierung der Gleichheitssuche.
    - Erweiterung der Struktur zur Implementierung der Infixsuche.

Die Kombination aller vorgenannten Indexierungstechniken auf entsprechend tokenisierte Freitexte erlaubt die Implementierung eines verschlüsselten Volltextindexes; dies ist neben den beiden oben genannten Hauptkategorien Gleichheits- und Ungleichheitssuche eine weitere Indexkategorie, die in verschlüsselter Form implementier- und einsetzbar ist.

Neben der Kernfunktionalität der Indexstrukturen werden in der Dissertation weitere praxisrelevante Erweiterungen und Verbesserungen, insbesondere der cTree-Datenstruktur, entwickelt. Dies umfasst:

- Synchronisierung von parallelen Indexaktualisierungen,
- ein leichtgewichtiges Transaktionskonzept zur Steuerung des Mehrbenutzerbetriebs, welches auf einem optimistischen Sperrkonzept beruht und
- zahlreiche Maßnahmen zur Steigerung der Performanz des cTree-Indexes.

Diese Techniken sind in [Lehnhardt2014-2] und [Lehnhardt2016-1] publiziert.

Bezüglich der Informationssicherheit und ihrer üblichen Schutzziele Vertraulichkeit, Integrität und Verfügbarkeit konzentriert sich die Dissertation lediglich auf Ersteres; für die anderen beiden wird auf Standardlösungen verwiesen.

Alle für diese Dissertation entwickelten Indexstrukturen offenbaren Teile der Klartextinformation der indexierten Daten. Damit dieser Informationsverlust die praktische Einsetzbarkeit nicht beeinträchtigt, enthält die Dissertation in Kapitel 11 eine Reihe von Modifikationen der Indexstrukturen, die den Informationsverlust eliminieren bzw. signifikant verringern. Somit haben auch diese Modifikationen einen Anteil am wissenschaftlichen Beitrag der Dissertation.

Die Modifikationen zur Minimierung des Informationsverlusts gehen auf Kosten der Performanz, so dass sich das Trade-off „Vertraulichkeit vs Performanz“ offenbart. Dieses Trade-off schränkt die praktische Einsetzbarkeit der Indexstrukturen jedoch nicht ein: In einen pragmatischen Ansatz mit dem Namen „Skalierbare Vertraulichkeit“ wird einem Informationssystem-Entwickler die Möglichkeit gegeben, diejenige Balance zwischen Vertraulichkeit und Performanz flexibel zu wählen, die für sein entwickeltes Informationssystem angemessen ist. Anwendungsszenarien mit einem hohen Bedarf an Schutz der im Informationssystem gespeicherten Daten können diesen erhalten, wenn auch unter Inkaufnahme von Performanzverlust. Im Gegensatz dazu können andere Anwendungsszenarien, bei denen eine Offenbarung von Teilen der Struktur der indexierten Daten akzeptabel ist, dies um einer hohen Performanz willen in Kauf nehmen.

12.10 Ausblick

Für die Weiterentwicklung der Indexstrukturen bieten sich verschiedene Ansätze an:

Eine naheliegende Erweiterung der cTree-Indexdatenstruktur sind Indexextensionen, in denen nicht nur Einfügeoperationen erlaubt sind, wie dies derzeit der Fall ist, sondern auch Löschoperationen. Dies ist in der in Kapitel 6 vorgestellten Basisausprägung des cTree-Indexes keine große Herausforderung, da lediglich die Löschoperation auf dem AVL-Baum nachgebildet werden muss. Bezüglich des clientseitigen Cachings von Teilbäumen erfordert die durch die Löschungen veränderte Evolution des Baumes eine entsprechende Anpassung der Cache-Löschstrategien, die in Kapitel 9.7.3 behandelt worden sind. Der Aufwand hierfür dürfte gering sein und der cTree-Indexstruktur eine breitere Anwendbarkeit verleihen.

Angesichts der Orientierung an praktischer Anwendbarkeit liegt es nahe, die Indexstrukturen für eine starke Skalierung des Servers vorzubereiten. Dies beinhaltet naheliegende Techniken wie etwa Clustering auf Datenbankebene, wofür sich die Indexstrukturen gut eignen. Aber auch eine weitere Form des serverseitigen Cachings bietet sich an: Analog dazu, dass cTree-Teilbäume clientseitig im Cache zwischengespeichert werden, um Client-Server-Roundtrips einzusparen, können sie auch serverseitig auf der Ebene des Application Servers in Caches abgelegt werden. Dies hätte zum einen den Vorteil, dass Anfragen nach Teilbäumen nicht bis auf Datenbankebene propagiert werden müssten, wo die Teilbäume in recht teuren Arbeitsschritten zusammengestellt werden müssen[102].

Zum anderen existieren für diese Schicht der Systemarchitektur zahlreiche Caching-Lösungen, wie etwa Infinispan[103] von Red Hat, die gut skalieren und sich dabei automatisch synchronisieren, wenn sie sich auf mehrere Knoten verteilen. Infinispan ist ein *Key-Value-Store*, bei dem ein Cache-Eintrag, ebenfalls analog zum clientseitigen Caching, jeweils einen cTree-Teilbaum beinhalten würde, der mit der *seq* des Wurzelknotens identifiziert würde. Bezüglich Löschstrategien für ungültig gewordene Cache-Einträge könnten die Konzepte aus Kapitel 9.7.3, insbesondere die transitiven Löschansätze aus Kapitel 9.7.3.3, gut in den Infinispan-Kontext übertragen werden.

Der in Kapitel 7.3.6 angedachte Ausbau der Indexstrukturen, so dass sie komplexere reguläre Ausdrücke unterstützen, sollte ebenfalls vorangetrieben werden. Jedoch ist dabei zu bedenken, dass gerade die Indexstrukturen in Kapitel 7 besonders unter Informationsverlust zu leiden hatten. Gerade hier sollten so viele Gegenmaßnahmen wie möglich eingesetzt werden, um diesen Informationsverlust einzudämmen.

Eine Erweiterung, die für den Betrieb eines Informationssystems, das die Indexstrukturen nutzt, von Bedeutung sein könnte, ist die Offline-Vorberechnung eines cTree-Index. Beispielsweise bei der Migration großer Datenbestände wäre es eine große Hilfe, den Index nicht inkrementell aufbauen zu müssen, sondern den Datenbestand in sortierter Form im Klartext vorliegen zu haben, die entsprechenden Attributwerte der Indexrelation vorzuberechnen und en bloc im DBMS zu speichern. Dieses Vorgehen ist bereits entwickelt und in [Lehnhardt2016-2] patentiert.

---

[102] In der praktischen Implementierung wurden hierfür *Common Table Expressions* (*CTE*) verwendet, die Rekursion bei der mengenorientierten Datenverarbeitung in relationalen DBMS ermöglichen.
[103] siehe [URL-Infinispan].

12.11 Fazit

In dieser Dissertation werden verschlüsselte Indexstrukturen für die Suche auf verschlüsselten Daten vorgestellt. Der Fokus liegt dabei auf praktische Anwendbarkeit; die unterstützten Suchtypen umfassen daher mit Gleichheits-, Intervall-, Präfix-, Infix- und Volltextsuche die gängigsten Suchtypen in Informationssystemen.

Die Performanz dieser Indexstrukturen lässt sich so weit steigern, dass sie Echtzeitanforderungen standhalten können, wie sich in Testimplementierungen zeigen lässt. Die Implementierung der Indexstrukturen in einem kommerziell eingesetzten System eines Anbieters für medizinische Informationssysteme unterstreichen ihre Praxistauglichkeit.[104]

Weiterhin zeigt sich, dass die Indexstrukturen Klartextinformation über die verschlüsselten Daten auf dem Server exponieren, welche ihre Vertraulichkeit beeinträchtigen kann. Gegenmaßnahmen verringern das Problem jedoch signifikant auf ein für zahlreiche Anwendungen vertretbares Niveau, so dass die Anwendbarkeit der Indexstrukturen nicht beeinträchtigt wird.

[104] siehe [URL-MedicalCloud].

13 Bibliografie

[Abdalla2005] ABDALLA, Michel, et al. Searchable encryption revisited: Consistency properties, relation to anonymous IBE, and extensions. In: *Advances in Cryptology–CRYPTO 2005*. Springer Berlin Heidelberg, 2005. S. 205-222.

[Agrawal2004] AGRAWAL, Rakesh, et al. Order preserving encryption for numeric data. In:*Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 2004. S. 563-574.

[Agrawal2009] AGRAWAL, Divyakant, et al. Database management as a service: Challenges and opportunities. In: *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. IEEE, 2009. S. 1709-1716.

[Ang2014] ANG, George Weilun; WOELFEL, John Harold; WOLOSZYN, Terrence Peter.*System and method of sort-order preserving tokenization*. U.S. Patent Nr. 8,739,265, 2014.

[Arasu2013] ARASU, Arvind, et al. Orthogonal Security with Cipherbase. In: *CIDR*. 2013.

[Bajaj2014] BAJAJ, Sumeet; SION, Radu. TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality. *Knowledge and Data Engineering, IEEE Transactions on*, 2014, 26. Jg., Nr. 3, S. 752-765.

[Bellare2007] BELLARE, Mihir; BOLDYREVA, Alexandra; O'NEILL, Adam. Deterministic and efficiently searchable encryption. In: *Advances in Cryptology-CRYPTO 2007*. Springer Berlin Heidelberg, 2007. S. 535-552.

[Beutelspacher2003] BEUTELSPACHER, Albrecht. Lineare Algebra. Mathematik für Studienanfänger, Vieweg & Sohn Verlag, 2003.

[Bethencourt2006] BETHENCOURT, John; SONG, Dawn; WATERS, Brent. New constructions and practical applications for private stream searching. In: *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 2006. S. 132-139.

[Boldyreva2009] BOLDYREVA, Alexandra, et al. Order-preserving symmetric encryption. In:*Advances in Cryptology-EUROCRYPT 2009*. Springer Berlin Heidelberg, 2009. S. 224-241.

[Boldyreva2011] BOLDYREVA, Alexandra; CHENETTE, Nathan; O'NEILL, Adam. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In: Advances in Cryptology–CRYPTO 2011. Springer Berlin Heidelberg, 2011. S. 578-595.

[Boneh2004] BONEH, Dan, et al. Public key encryption with keyword search. In: *Advances in Cryptology-Eurocrypt 2004*. Springer Berlin Heidelberg, 2004. S. 506-522.

[Boneh2007] BONEH, Dan; WATERS, Brent. Conjunctive, subset, and range queries on encrypted data. In: *Theory of cryptography*. Springer Berlin Heidelberg, 2007. S. 535-554.

[Boyen2007] BOYEN, Xavier; WATERS, Brent. Anonymous hierarchical identity-based encryption (without random oracles). In: *Advances in Cryptology-CRYPTO 2006*. Springer Berlin Heidelberg, 2006. S. 290-307.

[Cao2014] CAO, Ning; WANG, Cong; LI, Ming; REN, Kui; LOU, Wenjing. Privacy-preserving multi-keyword ranked search over encrypted cloud data. In: *IEEE Transactions on parallel and distributed systems*, 2014, 25. Jg., Nr. 1, S. 222-233.

[Cormen2009] CORMEN, Thomas H. *Introduction to algorithms*. MIT press, 2009.

[Danezis2011] DANEZIS, George; LIVSHITS, Benjamin. Towards ensuring client-side computational integrity. In: *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. ACM, 2011. S. 125-130.

[Dingledine2004] DINGLEDINE, Roger; MATHEWSON, Nick; SYVERSON, Paul. Tor: The second-generation onion router. Naval Research Lab Washington DC, 2004.

[Eckert2013] ECKERT, Claudia. IT-Sicherheit: Konzepte-Verfahren-Protokolle. Walter de Gruyter, 2013.

[Edlich2011] EDLICH, Stefan; FRIEDLAND, Achim; HAMPE, Jens; BRAUER, Benjamin. *NoSQL*. Carl Hanser Fachbuchverlag, 2011.

[Ferguson2003] FERGUSON, Niels; SCHNEIER, Bruce. Practical cryptography. John Wiley & Sons, New York, 2003.

[Ferguson2010] FERGUSON, Niels; SCHNEIER, Bruce; KOHNO, Tadayoshi. Cryptography engineering. John Wiley & Sons, New York, 2010.

[Ferreira2012] FERREIRA, Bernardo; DOMINGOS, Henrique. Management and search of private data on storage clouds. In: *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*. ACM, 2012. S. 4.

[Franz1998] FRANZ, E.; GRAUBNER, A.; JERICHOW, A; Pfitzmann, A. Comparison of commitment schemes used in mix-mediated anonymous communication for preventing pool-mode attacks. In: Information Security and Privacy. Springer Berlin Heidelberg, 1998. S. 111-122.

[Furnell2008] FURNELL, Steven. Securing information and communications systems: Principles, technologies, and applications. Artech House, 2008.

[Ge2007] GE, Tingjian; ZDONIK, Stan. Answering aggregation queries in a secure system model. In: *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007. S. 519-530.

[Gentry2009] GENTRY, Chris. Fully Homomorphic Encryption Using ideal Lattices. In: *STOC '09 - ACM symposium on Theory of computing*. ACM, 2009. S. 169-178.

[Goldwasser1984] GOLDWASSER, Shafi; MICALI, Silvio. Probabilistic encryption. Journal of computer and system sciences, 1984, 28. Jg., Nr. 2, S. 270-299.

[Golle2004] GOLLE, Philippe; STADDON, Jessica; WATERS, Brent. Secure conjunctive keyword search over encrypted data. In: *Applied Cryptography and Network Security*. Springer Berlin Heidelberg, 2004. S. 31-45.

[Hacigümüş2002-1] HACIGÜMÜŞ, Hakan; IYER, Bala; MEHROTRA, Sharad. Providing database as a service. In: *Data Engineering, 2002. Proceedings. 18th International Conference on*. IEEE, 2002. S. 29-38.

[Hacigümüş2002-2] HACIGÜMÜŞ, Hakan, et al. Executing SQL over encrypted data in the database-service-provider model. In: *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 2002. S. 216-227.

[Kadhem2010-1] KADHEM, Hasan; AMAGASA, Toshiyuki; KITAGAWA, Hiroyuki. MV-OPES: Multivalued-order preserving encryption scheme: A novel scheme for encrypting integer value to many different values. *IEICE TRANSACTIONS on Information and Systems*, 2010, 93. Jg., Nr. 9, S. 2520-2533.

[Kadhem2010-2] KADHEM, Hasan; AMAGASA, Toshiyuki; KITAGAWA, Hiroyuki. A Secure and Efficient Order Preserving Encryption Scheme for Relational Databases. In:*KMIS*. 2010. S. 25-35.

[Kamara2015] KAMARA, Seny; NAVEED, Muhammad; WRIGHT, Charles. Inference Attacks on Property-Preserving Encrypted Databases. In: Proceedings of the 22nd ACM Conference on Computer and Communications Security – CCS'15. ACM, 2015.

[Kemper2011] KEMPER, Alfons; EICKLER, André. *Datenbanksysteme: Eine Einführung*. Oldenbourg Verlag, 2011.

[Kerschbaum2014] KERSCHBAUM, Florian; SCHRÖPFER, Axel. Optimal average-complexity ideal-security order-preserving encryption. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014. S. 275-286.

[Lehnhardt2014-1] LEHNHARDT, Jan; RHO, Tobias; SPALKA, Adrian; CREMERS, Armin B. Ordered Range Searches on Encrypted Data. In: *Technical Report IAI-TR-2014-03, Computer Science Department III, University of Bonn, ISSN 0944-8535*.

[Lehnhardt2014-2] LEHNHARDT, Jan; RHO, Tobias; SPALKA, Adrian; CREMERS, Armin B. Performance-Optimized Indexes for Inequality Searches on Encrypted Data in Practice. In: *Proceedings of the 1st International Conference on Information Systems Security and Privacy (ICISSP)*, SCITEPRESS, 2015. S. 221-229.

[Lehnhardt2016-1] LEHNHARDT, Jan; RHO, Tobias. Method for querying a database. European Patent Office. EP16204660.1. Anmeldedatum: 23.12.2016.

[Lehnhardt2016-2] LEHNHARDT, Jan; RHO, Tobias. Offline preparation for bulk inserts. European Patent Office. EP16206698.9. Anmeldedatum: 16.12.2016.

[Li2010] LI, Jin; WANG, Qian; WANG, Cong; CAO, Ning; REN, Kui; LOU, Wenjing. Enabling Efficient Fuzzy Keyword Search over Encrypted Data in Cloud Computing. In: *INFOCOM, 2010 Proceedings IEEE*. IEEE, 2010. S. 1-5.

[Liu2012] LIU, Dongxi; WANG, Shenlu. Programmable order-preserving secure index for encrypted database query. In: *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012. S. 502-509.

[Liu2013] LIU, Dongxi; WANG, Shenlu. Nonlinear order preserving index for encrypted database query in service cloud environments. *Concurrency and Computation: Practice and Experience*, 2013, 25. Jg., Nr. 13, S. 1967-1984.

[Narayanan2008] NARAYANAN, Arvind; SHMATIKOV, Vitaly. Robust de-anonymization of large sparse datasets. In: *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 2008. S. 111-125.

[Nielsen1994] NIELSEN, Jakob. Usability engineering. Elsevier, 1994.

[Ostrovsky2005] OSTROVSKY, Rafail; SKEITH III, William E. Private searching on streaming data. In: *Advances in Cryptology–CRYPTO 2005*. Springer Berlin Heidelberg, 2005. S. 223-240.

[Özsoyoglu2003] ÖZSOYOGLU, Gultekin; SINGER, David A.; CHUNG, Sun S. Anti-Tamper Databases: Querying Encrypted Databases. In: *DBSec*. 2003. S. 133-146.

[Pappas2011] PAPPAS, Vasilis, et al. Private search in the real world. In: *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011. S. 83-92.

[Patterson2011] HENNESSY, John L.; PATTERSON, David A. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[Pfaff1998] PFAFF, Ben. An Introduction to Binary Search Trees and Balanced Trees. *Libavl Binary Search Tree Library*, 1998, 1. Jg.

[Popa2012] POPA, Raluca Ada, et al. CryptDB: Processing queries on an encrypted database. *Communications of the ACM*, 2012, 55. Jg., Nr. 9, S. 103-111.

[Popa2013] POPA, Raluca A.; LI, Frank H.; ZELDOVICH, Nickolai. An ideal-security protocol for order-preserving encoding. In: *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013. S. 463-477.

[Schneier2007] SCHNEIER, Bruce. Applied cryptography: protocols, algorithms, and source code in C. john wiley & sons, 2007.

[Seungmin2009] SEUNGMIN, L. E. E., et al. Chaotic order preserving encryption for efficient and secure queries on databases. *IEICE transactions on information and systems*, 2009, 92. Jg., Nr. 11, S. 2207-2217.

[Shi2007] SHI, Elaine, et al. Multi-dimensional range query over encrypted data. In:*Security and Privacy, 2007. SP'07. IEEE Symposium on*. IEEE, 2007. S. 350-364.

[Song2000] 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, 2000. S. 44-55.

[Sweeney2002] SWEENEY, Latanya. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 2002, 10. Jg., Nr. 05, S. 557-570.

[Tu2013] TU, Stephen, et al. Processing analytical queries over encrypted data. In: *Proceedings of the VLDB Endowment*. VLDB Endowment, 2013. S. 289-300.

[URL-BDSG-11] Bundesministerium der Justiz und für Verbraucherschutz. Bundesdatenschutzgesetz (BDSG) - § 11 Erhebung, Verarbeitung oder Nutzung personenbezogener Daten im Auftrag. URL https://www.gesetze-im-internet.de/bdsg_1990/__11.html (abgerufen am 28.06.2017).

[URL-BDSG-4] Bundesministerium der Justiz und für Verbraucherschutz. Bundesdatenschutzgesetz (BDSG) - § 4 Zulässigkeit der Datenerhebung, -verarbeitung und –nutzung. URL https://www.gesetze-im-internet.de/bdsg_1990/__4.html (abgerufen am 28.06.2017).

[URL-MedicalCloud] CGM SE. Auf ins mobile Zeitalter – mit der Medical Cloud. URL https://www.cgm.com/de/ueber_uns_de/medizinische_technologien/med_cloud/auf_ins_mobile_zeitalter_mit_der_medical_cloud.de.jsp (abgerufen am 28.06.2017).

[URL-DIN-5007-1] Deutsches Institut für Normung. Norm DIN-5007-1. URL http://www.din.de/de/mitwirken/normenausschuesse/nia/normen/wdc-beuth:din21:80825245 (abgerufen am 28.06.2017).

[URL-Dropbox] Dropbox. URL https://www.dropbox.com/ (abgerufen am 28.06.2017).

[URL-Frequencies-1] Hebisch, Udo. Häufigkeitstabellen (einzelner Buchstaben in bestimmten Sprachen). URL http://www.mathe.tu-freiberg.de/~hebisch/cafe/kryptographie/haeufigkeitstabellen.html (abgerufen am 28.06.2017).

[URL-Frequencies-2] Bibliographisches Institut GmbH. Die häufigsten Wörter in deutschsprachigen Texten. URL http://www.duden.de/sprachwissen/sprachratgeber/die-haeufigsten-woerter-in-deutschsprachigen-texten (abgerufen am 28.06.2017).

[URL-GoogleDrive] Google Drive. URL https://www.google.com/drive/ (abgerufen am 28.06.2017).

[URL-ICD-10] Deutsches Institut für Medizinische Dokumentation und Information. ICD-10-GM. URL: http://www.dimdi.de/static/de/klassi/icd-10-gm/index.htm (abgerufen am 28.06.2017).

[URL-ICISSP] FORD, Bryan. Hiding in a Panopticon - Grand Challenges in Internet Anonymity. In: ICISSP 2015. URL https://vimeo.com/119321487 (abgerufen am 28.06.2017).

[URL-ICloud] Apple iCloud. URL https://www.icloud.com/ (abgerufen am 28.06.2017).

[URL-IEEE-754] Institute of Electrical and Electronics Engineers. 754-2008 - IEEE Standard for Floating-Point Arithmetic. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4610935 (abgerufen am 28.06.2017).

[URL-Infinispan] Red Hat Infinispan. URL http://infinispan.org/ (abgerufen am 28.06.2017).

[URL-Name-DB] Norvelle, Erik. NameDatabases. URL https://github.com/enorvelle/NameDatabases (abgerufen am 28.06.2017).

[URL-OneDrive] Microsoft OneDrive. URL https://onedrive.live.com (abgerufen am 28.06.2017).

[URL-OwnCloud] ownCloud. URL https://owncloud.org/ (abgerufen am 28.06.2017).

[URL-PostgreSQL] PostgreSQL Global Development Group. PostgreSQL 9.6.3 Documentation. Chapter 13. Concurrency Control. URL https://www.postgresql.org/docs/9.6/static/explicit-locking.html (abgerufen am 28.06.2017).

[URL-Seafile] Seafile. URL https://www.seafile.com/ (abgerufen am 28.06.2017).

[URL-Snowball] Snowball small string processing language. URL http://snowballstem.org/ demo.html (abgerufen am 28.06.2017).

[URL-SQL-92] Digital Equipment Corporation. ISO/IEC 9075:1992, Database Language SQL. URL http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt (abgerufen am 28.06.2017).

[URL-StGB-203] Bundesministerium der Justiz und für Verbraucherschutz. Strafgesetzbuch (StGB) - § 203 Verletzung von Privatgeheimnissen. URL https://www.gesetze-im-internet.de/stgb/__203.html (abgerufen am 28.06.2017).

[URL-StGB-203-2] Bundesministerium der Justiz und für Verbraucherschutz. Gesetz zur Neuregelung des Schutzes von Geheimnissen bei der Mitwirkung Dritter an der Berufsausübung schweigepflichtiger Personen. URL https://www.bmjv.de/SharedDocs/Gesetzgebungsverfahren/DE/Neuregelung_Schutzes_von_Geheimnissen_bei_Mitwirkung_Dritter_an_der_Berufsausuebung_schweigepflichtiger_Personen.html (abgerufen am 28.06.2017).

[URL-TOR] Tor. URL https://www.torproject.org/ (abgerufen am 28.06.2017).

[URL-TPC] Transaction Processing Performance Council. TPC Benchmark™ H. URL http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf (abgerufen am 28.06.2017).

[VanTilborg2014] VAN TILBORG, Henk CA; JAJODIA, Sushil (Hg.). Encyclopedia of cryptography and security. Springer Science & Business Media, 2014.

[Wang2010] WANG, Cong; CAO, Ning; LI, Jin; REN, Kui; LOU, Wenjing. Secure ranked keyword search over encrypted cloud data. In: Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on. IEEE, 2010. S. 253-262.

[Wang2014] WANG, Bing; YU, Shucheng; LOU, Wenjing; HOU, Y. Thomas. Privacy-Preserving Multi-Keyword Fuzzy Search over Encrypted Data in the Cloud. In: *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 2014. S. 2112-2120.

[Xiao2012-1] XIAO, Liangliang; YEN, I.-Ling; HUYNH, Dung T. Extending Order Preserving Encryption for Multi-User Systems. *IACR Cryptology ePrint Archive*, 2012, 2012. Jg., S. 192.

[Xiao2012-2] XIAO, Liangliang; YEN, I.-Ling; HUYNH, D. T. A Note for the Ideal Order-Preserving Encryption Object and Generalized Order-Preserving Encryption. *IACR Cryptology ePrint Archive*, 2012, 2012. Jg., S. 350.

[Yum2012] YUM, Dae Hyun, et al. Order-preserving encryption for non-uniformly distributed plaintexts. In: *Information Security Applications*. Springer Berlin Heidelberg, 2012. S. 84-97.