Otto von Guericke Universität Magdeburg
Fakultät für Informatik

Masterarbeit

Forensisch sicheres Löschen in relationalen Datenbankmanagementsystemen

Verfasser:

Alexander Grebhahn

27. Februar 2012

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake,
Dipl.-Inform. Martin Schäler
Dipl.-Inform. Mario Pukall

Institut für Technische und Betriebliche Informationssysteme (ITI)

Grebhahn, Alexander:
Forensisch sicheres Löschen in relationalen Datenbankmanagementsystemen
Masterarbeit, Otto von Guericke Universität Magdeburg, 2012.

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mir bei der Fertigstellung dieser Arbeit geholfen haben. Ein besonderer Dank gilt dabei Prof. Dr. rer. nat. habil. Gunter Saake, Dipl.-Inform. Martin Schäler und Dipl.-Inform. Mario Pukall, die mich im Laufe dieser Arbeit sehr gut betreut haben. Außerdem bedanke ich mich bei Sebastian Bress und Christoph Mewes, die bei Problemen immer ein offenes Ohr für mich hatten.
Des Weiteren möchte ich mich auch bei Freunden, meiner Freundin und meiner Familie bedanken, die mich während des Bearbeitungszeitraums dieser Arbeit seelisch und moralisch unterstützt haben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

DB	Datenbank
DBMS	Datenbankmanagementsystem
DBS	Datenbanksystem
HSQLDB	HyperSQL DataBase
MVCC	Multiversion Concurrency Control Protokoll
OID	Objekt ID
RDBMS	Relationales Datenbankmanagementsystem
SQL	Structured Query Language

1. Einleitung

Die Speicherung von personenbezogenen Daten hat durch die vermehrte Verwendung von computergestützten Informationssystemen enorme Ausmaße angenommen.
In diesem Zusammenhang versuchen die Gesetzgeber in vielen Ländern die Speicherung und die Weitergabe dieser Informationen gesetzlich zu regeln. So ist zum Beispiel in den Vereinigten Staaten von Amerika die Verwendung von personenbezogenen Gesundheitsinformationen durch die HIPAA [Con96] geregelt. Es existieren auch Vorschriften, die zum Beispiel die Speicherung von IP-Adressen oder Verkehrsdaten festlegen. Unter Verkehrdaten werden in diesem Zusammenhang Daten verstanden, die „bei der Erbringung eines Telekommunikationsdienstes erhoben, verarbeitet und genutzt werden“ [Bun04]. Der Grund für die Speicherung solcher Daten kann unter anderem eine mögliche Verhinderung von terroristischen Straftaten sein. Jedoch ist zu beachten, dass es sich bei diesen Daten um sehr sensible Daten handelt, die nicht missbraucht werden dürfen. Daher ist ihre Verwendung innerhalb der Europäischen Union durch eine Richtlinie festgelegt, welche besagt, dass die Daten nur für einen gewissen Zeitraum gespeichert werden dürfen. Ist dieser Zeitrahmen abgelaufen, müssen sie entweder anonymisiert oder gelöscht werden [UNI02]. Obwohl es nach einem Eckpunktepapier der Deutschen Bundesregierung nicht vorgesehen ist, die Daten jedes Bundesbürgers zu speichern [fJ11], ist trotzdem zu erwarten, dass es sich bei den erfassten Daten um große Datenmengen handelt.
Damit diese enormen Datenmengen überhaupt verarbeitet werden können, bieten sich für ihre Verwaltung und Speicherung Datenbanksysteme an. Im Hinblick auf das forensisch sichere Löschen von Informationen bringen diese Systeme jedoch einige Probleme mit sich. So müssen Datenbanksysteme beispielsweise die Möglichkeit besitzen, Daten im Fehlerfall wiederherstellen zu können [Cod82]. Damit dies durchführbar ist, werden von den Systemen zusätzliche Informationen angelegt. Diese Informationen erschweren jedoch das forensisch sichere Löschen von Daten, da durch sie auch bereits gelöschte Daten wiederhergestellt werden können. Es müssen somit Techniken entwickelt werden, die sowohl ein forensisch sicheres Löschen als auch die Wiederherstellung von Daten im Fehlerfall unterstützen.

1.1 Zielsetzung

In dieser Arbeit sollen Methoden erarbeitet werden, durch die ein forensisch sicheres Löschen von personenbezogenen Daten aus relationalen Datenbankmanagementsystemen durchgeführt werden kann. Dabei muss in erster Hinsicht analysiert werden, welche Mittel notwendig sind, um persistent gespeicherte Informationen forensisch sicher zu löschen. Nachdem dies geschehen ist, soll eine theoretische Betrachtung, des Aufbaus, dieser Systeme erfolgen, um einen generellen Einblick in die Problematik zu geben. In diesem Zusammenhang soll herausgestellt werden, welche zusätzlichen Informationen, neben den Daten selbst, von einem Datenbanksystem (DBS) gespeichert werden und wie es ermöglicht werden kann, Datensätze trotz dieser Informationen forensisch sicher zu löschen. Da auf diesem Gebiet schon Teillösungen existieren, wird im zweiten Teil dieser Arbeit analysiert, in wieweit es möglich ist, diese Lösungen zu generalisieren und auf andere DBSs anzuwenden. Dabei sollen exemplarisch Datenbankmanagementsysteme herangezogen und adaptiert werden.

1.2 Gliederung

Die Arbeit ist wie folgt gegliedert. Im Kapitel 2 dieser Arbeit wird dargelegt, welche unterschiedlichen Methoden existieren, wenn Daten forensisch sicher von einem Speichermedium entfernt werden sollen. Anschließend dazu wird auf den konzeptionellen Aufbau eingegangen, den ein DBSs unterliegt. Dazu werden Architekturen von Datenbankmanagementsystem (DBMS) vorgestellt und es werden die wichtigsten Komponenten der DBS betrachtet. In diesem Zusammenhang wird gezeigt, welche verschiedenen Informationen über einen Datensatz zusätzlich zu diesem Datensatz gespeichert werden.
Im Kapitel 3 werden die DBSs in ihre generellen Bestandteile zerlegt. Es erfolgt daraufhin eine Analyse dieser Bestandteile dahingehend, in wieweit sie verändert werden müssen um ein forensisch sicheres Löschen zu ermöglichen.
Nachdem diese theoretische Betrachtung abgeschlossen ist, wird im Kapitel 4 auf die Speicherverwaltung von einigen ausgewählten DBMS eingegangen. In diesem Zusammenhang wird gezeigt, welche Techniken zum Löschen von Datensätzen aus den Datenbank (DB) verwendet werden. Außerdem wird dargelegt, welche Möglichkeiten bestehen die Datensätze bereits auf Dateiebene zu rekonstruieren.
Im Kapitel 5 erfolgt eine Erweiterung von zwei DBMSs. Ziel dieser Erweiterung ist es, zu verhindern, dass die Werte gelöschter Datensätze auch nach dem Löschen noch innerhalb der Datenbank weiter existieren. Nachfolgend dazu wird in Kapitel 6 evaluiert, welche Veränderung der Ausführungszeit von Transaktionen durch die Implementierung entsteht.
Abschließend erfolgt im Kapitel 7 eine Zusammenfassung der Arbeit und im Kapitel 8 wird gezeigt, was in zukünftigen Arbeiten noch durchgeführt werden kann.

2. Grundlagen

In diesem Kapitel werden allgemeine Grundlagen dargelegt, die notwendig sind, um das Thema und verwendeten Lösungen zu verstehen. Dazu wird gezeigt, welche Möglichkeiten existieren, um Daten nicht rekonstruierbar von einem Speichermedium zu löschen. Danach wird näher auf den Aufbau von Relationales Datenbankmanagementsystem (RDBMS) eingegangen. Dazu werden zwei Architekturen von DBS vorgestellt. Im Anschluss dazu wird näher auf die Komponenten von DBS eingegangen. Es folgt eine Analyse der unterschiedlichen Datenbestände, die von einem DBS angelegt werden, im Hinblick auf die Möglichkeit des Rekonstruierens von Datensätzen. Außerdem werden in diesem Abschnitt Lösungen vorgestellt, durch die gezeigt wird, wie eine Rekonstruktion der Daten aus den Datenbeständen teilweise verhindert werden kann.

2.1 Löschen und Anonymisieren von personenbezogenen Daten

Nach der Datenschutzrichtlinie für elektronische Kommunikation des Europäischen Parlamentes und des Europäischen Rates gibt es zwei Möglichkeiten der Bearbeitung von personenbezogenen Daten, wenn sie nicht mehr gespeichert werden dürfen. Bei diesen Möglichkeiten handelt es sich um das Anonymisieren und um das Löschen der Daten [UNI02]. Dabei müssen die Änderungen an der physischen Repräsentation der Daten auf eine Weise durchgeführt werden, die nicht rückgängig gemacht werden kann. Die Daten dürfen also nicht rekonstruierbar sein. Jedes dieser Verfahren besitzt dabei seine Besonderheiten. Sollen die Daten anonymisiert werden, kann dies zum Beispiel durch das Verschlüsseln der Daten geschehen. Um diese verschlüsselten Daten dann nicht rekonstruierbar zu löschen, muss lediglich der Schlüssel, durch den sie verschlüsselt wurden, gelöscht werden.

Diese Verschlüsselung bringt jedoch einige Probleme mit sich. So muss eine Möglichkeit bestehen, die Granularität der einzeln verschlüsselbaren Blöcke so zu wählen, dass der ganze Datensatz und keine weiteren Daten verschlüsselt werden können. Dies ist

besonders dann eine Herausforderung, wenn Daten unterschiedlicher Länge verschlüsselt werden sollen. Außerdem müssen die Schlüssel, die für die Verschlüsselung benutzt wurden, sicher verwaltet werden [SML07]. Bei der Verwendung der verschlüsselten Daten besteht die Problematik, dass bei jedem Lesen eine Entschlüsselung erfolgen muss. Dies kann je nach Komplexität des Schlüssels viel Zeit in Anspruch nehmen. Es existieren zwar Prozessoren, die eine solche Ver- und Entschlüsselung auf Hardwareebene unterstützen [Gue10, NIS01], jedoch kann nicht davon ausgegangen werden, dass diese hardwareseitige Unterstützung in allen verwendeten Systemen existiert. Außerdem besteht das Problem, dass die Daten nicht ohne Zuhilfenahme des Schlüssels entschlüsselbar sein dürfen. Somit ist diese Methode im Produktiveinsatz zum gegenwärtigen Zeitpunkt nur bedingt praktikabel.

Sollen die personenbezogenen Daten gelöscht werden, ist es nötig sie unter Zuhilfenahme von Mitteln zu löschen, die sicherstellen, dass sie nicht wieder rekonstruiert werden können. Dabei kann die potentielle Rekonstruktion der Daten unter Anwendung von unterschiedlichen Methoden geschehen. So kann dabei Software zum Einsatz kommen, die die freien Bereiche einer Festplatte analysiert. Es können aber auch Elektronenmikroskope verwendet werden, durch die die Restmagnetisierung einzelner Speicherblöcke analysiert wird. Generell reicht es bei dem Löschen nicht aus, die vom Betriebssystem bereitgestellte Löschfunktionalität zu verwenden. Der Grund dafür ist, dass bei einem einfachen Löschen von Dateien auf Betriebssystemebene der Speicherbereich, der von der Datei verwendet wurde, nur als frei verfügbar markiert wird. Da die Werte dieser gelöschten Daten weiterhin, bevor ihr Speicherbereich neu verwendet wurde, auf dem Sekundärspeichermedium gespeichert sind, ist eine Rekonstruktion dieser Daten durch die Nutzung frei verfügbarer Rekonstruktionswerkzeuge wie zum Beispiel Recuva[1] möglich [Tra11].

Wurden die Speicherbereiche einer Datei mit neuen Daten überschrieben, ist eine komplette Rekonstruktion der ursprünglichen Daten nur noch bedingt möglich. Dabei hängt es davon ab, mit welchem Muster die Daten überschrieben wurden. Wurden die Speicherbereiche der Daten einfach mit einer Folge von Nullen überschrieben, kann durch die Restmagnetisierung der Speicherbereiche herausgefunden werden, welche ursprünglichen Daten auf den Speicherbereichen existierten. Hier wird eine Rekonstruktion schwerer, je höher die Datendichte ist. Unter der Datendichte wird in diesem Zusammenhang die Anzahl der Daten verstanden, die auf einem physischen Bereich des Speichermediums gespeichert werden kann. Somit kann ein einfaches Überschreiben mit Nullen als Lösung nicht verwendet werden [Sch07].

Sind die Speicherbereiche einer Datei jedoch mit zufällig generierten Mustern überschrieben worden, ist eine Rekonstruktion der Daten nicht mehr möglich [Wri08]. Es können zwar noch einzelne Bits mit einer Wahrscheinlichkeit von über 90% rekonstruiert werden, jedoch nicht ganze Dateien, selbst wenn sie nur von geringer Größe sind. Es ist zum Beispiel nur noch mit einer Wahrscheinlichkeit von 0.06% möglich, einen 32 Bit langen Bereich korrekt zu rekonstruieren. An dieser Stelle sei angemerkt, dass diese 90% für die Rekonstruktion eines einzelnen Bits für neue Festplatten gelten, bei Festplatten, die unter täglicher Benutzung stehen, ist diese Wahrscheinlichkeit um einiges

[1] http://www.piriform.com/recuva, letzter Zugriff 30.01.2012

geringer und liegt bei etwas über 50%. Diese Wahrscheinlichkeiten sind von Festplatte zu Festplatte unterschiedlich. Die hier genannten Wahrscheinlichkeiten wurden für eine Seagate mit 1GB Speichervolumen ermittelt. Jedoch liegen diese Wahrscheinlichkeiten auch für andere Festplatten in der gleichen Größenordnung [Wri08]. Ein Beispiel dafür, dass mit zufälligen Daten überschriebene Daten nicht wieder rekonstruiert werden können, liefern auch Bauer und Priyantha [BP01]. Von ihnen wurden im Rahmen einer Fallstudie 20 Firmen befragt, die kommerzielle Wiederherstellungssoftware vertreiben, ob sie eine 100KB große mit zufälligen Daten überschriebene Datei wieder herstellen können. Auf diese Frage hin antworteten 19 dieser Firmen, dass dies nicht möglich sei und nur eine Firma war bereit, es zu versuchen. Dabei schätzen sie ihre Chance, die Datei wieder herzustellen, auf 1% ein [BP01]. Das einfache Überschreiben der Speicherbereiche einer Datei mit zufälligen Daten ist somit eine gute Möglichkeit, Daten nicht rekonstruierbar zu löschen.

Soll jedoch verhindert werden, dass auch nur ein einzelnes Bit der Daten wieder rekonstruiert werden kann, ist es notwendig, den Speicherbereich der Daten mehrfach zu überschreiben. Dazu stellte Gutmann [Gut96] 22 Pattern vor, durch deren Anwendung alle Bits einer Datei, unabhängig von der Speicherart auf der Festplatte sicher gelöscht werden können. Diese Pattern wurden mit einer Sequenz um zufällig generierte Muster erweitert. Durch diese zusätzliche Sequenz werden alle Bereiche 35 mal überschrieben [Gut96]. Allerdings sind in den letzten Jahren neue Codierungsstandards eingeführt wurden, deshalb kann diese Technik als veraltet betrachtet werden. So reicht es aufgrund von stochastischen Wahrscheinlichkeitsberechnungen schon aus, 33 Schreibvorgänge mit zufällig generierten Daten durchzuführen [Sch07].

Beim Überschreiben der Speicherbereiche muss immer darauf geachtet werden, dass alle Sektoren des Speichermediums, auf denen die Datei gespeichert worden ist, überschrieben werden. Geschieht dies nicht, entstehen Segmente, aus denen alte, nicht gelöschte Informationen rekonstruiert werden können. Diese nicht überschriebenen Segmente werden nachfolgend, angelehnt an Stahlberg et al. [SML07], als *Slacks* bezeichnet.

Ein generelles Problem des nicht rekonstruierbaren Löschens sind unbewusste Kopien der Daten. So kann es zum Beispiel passieren, dass eine Kopie des Datensatzes ungewollt durch ein Auslagern des Arbeitsspeichers entsteht [Sch07].

Das Löschen von Daten auf Dateisystemebene ist eine oft analysierte Problematik, die nicht Schwerpunkt dieser Arbeit ist. Hier wird sich mit dem Löschen von Datensätzen aus DBS beschäftigt, wobei nicht die Möglichkeit bestehen soll, einen gelöschten Datensatz zu einem späteren Zeitpunkt wieder zu rekonstruieren. Dabei werden in dieser Arbeit Daten als nicht rekonstruierbar betrachtet, wenn ihr Speicherbereich einmal überschrieben wurde. Diese Annahme beruht auf der Aussage, dass durch dieses einmalige Überschreiben eine Möglichkeit zur Rekonstruktion der Daten hinreichend verringert wurde, was in der Praxis meist schon ausreicht [Fox09].

Beim Löschen von Datensätzen aus einer DB werden vom DBS nicht die vom Betriebssystem bereitgestellten Funktionalitäten verwendet. Außerdem ist zu beachten, dass durch das DBS zusätzlich zu den Daten selbst noch Metadaten gespeichert werden. Werden die Daten noch über die Zeit hinweg durch Schreiboperationen verändert, ist

es notwendig, noch weitere Informationen innerhalb eines Logbuches zu speichern, um bei einem Fehlerfall den aktuellen konsistenten Zustand der Daten wieder rekonstruieren zu können. Deshalb können Datenbanken im Hinblick auf forensische Analysen im Gegensatz zu Dateisystemen, die als eindimensionales Konstrukt angesehen werden können, als mehrdimensionales Konstrukt betrachtet werden [Mar09]. Um diese Mehrdimensionalität zu analysieren, wird nachfolgend ein genauerer Einblick in den Aufbau von Relationalen Datenbankmanagementsystemen gegeben.

2.2 Architekturen von DBMS

Datenbanken speichern neben den eigentlichen Werten der Datensätze noch weitere Informationen. Es werden zum Beispiel Indexe, Logs, Materialisierte Sichten und temporäre Relationen angelegt [MLS07]. Soll ein Datensatz aus einem DBS nicht rekonstruierbar gelöscht werden, müssen nicht nur der Datensatz selbst, sondern auch alle zusätzlichen Informationen, die über den Datensatz gespeichert wurden, mit gelöscht werden. Damit diese Informationen nicht auf dem Sekundärspeichermedium verbleiben, ist eine Analyse der Struktur der Datenbanksysteme notwendig. Da DBS aus einem DBMS bestehen, dass auf einer DB arbeitet, muss genau analysiert werden, welche Informationen durch das DBMS zusätzlich gespeichert werden. Daher wird in diesem Abschnitt auf zwei Systemarchitekturen von DBMS eingegangen. Diese werden sowohl in Heuer und Saake [HS00], als auch in Saake et al. [SHS05] benutzt, um einen Einblick in den Aufbau von DBS zu geben. Da jedoch eine Analyse der Struktur nicht ausreicht, wird nachfolgend genauer auf die im DBS existierenden Komponenten eingegangen. Dabei werden besonders die Komponenten betrachtet, durch die Datensätze selbst oder Metadaten dieser Daten auf einem Sekundärspeichermedium gespeichert werden.

2.2.1 ANSI/SPARC Architektur

Bei der ersten vorgestellten Architektur handelt es sich um die ANSI/SPARC Architektur. Durch sie wird ein DBS in drei Ebenen aufgeteilt (siehe Abbildung 2.1). Bei den drei Ebenen handelt es sich um eine externe Ebene, eine konzeptuelle Ebene und eine interne Ebene. Die interne Ebene regelt dabei die Dateiorganisation auf dem Sekundärspeichermedium, die konzeptuelle Ebene die logische Struktur der Daten und die externe Ebene die einzelne Benutzersicht durch eine externe Applikation [Vos94]. Durch diese Einteilung entsteht eine physische Datenunabhängigkeit zwischen der konzeptuellen und der internen Ebene und eine logische Datenunabhängigkeit zwischen der externen und der konzeptuellen Ebene. Generell wird im internen Schema spezifiziert, **welche** Daten, **wie** gespeichert werden [Dat86].

Eine differenzierte Darstellung eines Teils des ANSI/SPARC Frameworks ist in Abbildung 2.2 zu sehen. In ihr werden die drei logischen Ebenen in zwei Bereiche zusammengefasst. Der obere Bereich ist die *Definition* des DBS und der untere Bereich seine *Verwendung* [Dat86]. In der Abbildung sind die aktiven Bestandteile des Frameworks als Quader dargestellt und die Rollen, die durch Personen übernommen werden, als Sechsecke. Zwischen den einzelnen Komponenten des Frameworks existieren wohl definiert

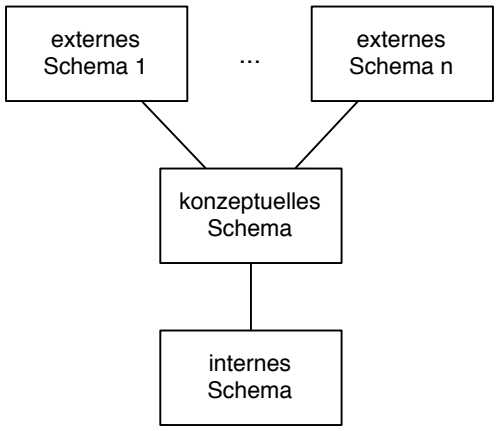

Abbildung 2.1: Drei-Ebenen-Architektur.

Schnittstellen. Durch sie wird eine große Flexibilität gewährleistet. Diese Schnittstellen sind in der Abbildung durch Zahlen gekennzeichnet, wobei die Nummerierung der Schnittellen aus [Dat86] übernommen wurde.

Soll eine Datenbank definiert werden, wird unter Verwendung der Schnittstelle 1 das konzeptuelle Schema der Datenbank definiert. Dieses Schema wird dann in Hinblick auf seine Konsistenz durch den konzeptuellen Schema Prozessor untersucht und durch die Schnittstelle 2 im Metadaten Speicher gespeichert. Außerdem werden durch den Konzeptuellen Schema Prozessor dem Datenbankadministrator und dem Applikationsadministrator über die Schnittstellen 3 die Schemainformationen der Datenbank bereitgestellt. Durch diese Informationen können dann unter der Verwendung der Schnittstellen 4 und 13 der interne beziehungsweise der externe Schema Prozessor definiert werden. Diese können dann die Informationen untersuchen und ihrerseits Informationen über die Schnittstellen 5 und 14 im Metadatenbereich speichern [Dat86].
Nachdem die Datenbank definiert wurde, kann sie durch die Benutzer verwendet werden. Dies kann durch eine Datenmanipulierungssprache, zum Beispiel die Structured Query Language (SQL), über die Schnittstelle 12 geschehen. Führt der Benutzer eine Anfrage aus, so wird sie durch den konzeptuellen externen, den konzeptuellen internen und den internen Speicher Umwandler unter Zuhilfenahme der Schnittstellen 31, 30 und 21 transformiert. Diese Komponenten benutzen dabei Informationen aus dem Metadatenbereich, wobei ihnen diese Informationen durch die Schnittstellen 38, 36 und 34 zur Verfügung gestellt werden [Dat86].
Es werden somit von vielen Bestandteilen des DBS innerhalb eines Metadatenbereichs Informationen gespeichert. Für eine detaillierte Betrachtung von DBS ist es notwendig, nicht nur eine Architektur zu betrachten, die eine anwendungsbezogene Sicht auf die Datenbank vornimmt. Es ist noch nötig zu analysieren, welche Transformationsschritte vom DBS bei der Ausführung einer Anfrage durchgeführt werden. Solch eine Betrachtung wird durch die nachfolgend vorgestellte Architektur vorgenommen.

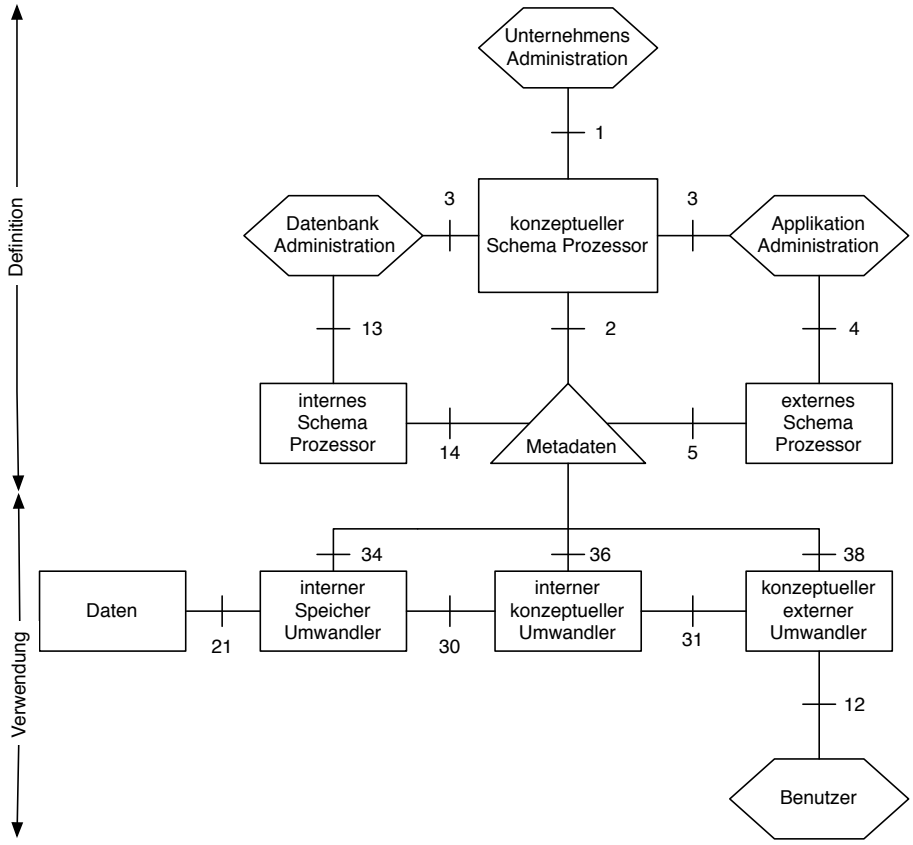

Abbildung 2.2: ANSI/SPARC Framework nach [Dat86].

2.2.2 Fünf-Schichten-Architektur

Bei der zweiten, in [SHS05] und [HS00] vorgestellten Architektur, handelt es sich um die Fünf-Schichten-Architektur. Sie wurde 1973 von Senke präsentiert [Mic73]. In Abbildung 2.3 ist ein DBMS im Hinblick auf die Schichten und die dazwischen liegenden Schnittstellen unterteilt. Bei der Fünf-Schichten-Architektur handelt es sich um eine Modularisierung von DBMS, die mit der in der Praxis verwendeten Aufteilung der Systeme übereinstimmt. Dabei beschreiben die Schichten der Fünf-Schichten-Architektur, welche Verarbeitungsschritte bei einer Anfrage- beziehungsweise Änderungsoperation, die auf dem DBMS ausgeführt werden soll, durchgeführt werden. Dabei nimmt der Abstraktionsgrad von oben nach unten zu [Vos94].

Sollen Operationen auf dem DBS realisiert werden, werden sie durch die mengenorientierte Schnittstelle entgegengenommen. Dabei werden die Operationen üblicherweise unter Zuhilfenahme einer deklarativen Datenmanipulationssprache beschrieben. Ein Beispiel für eine derartige Sprache ist SQL. In einer solchen Anweisung wird auf Tupeln, die in Relationen oder Sichten von Relationen enthalten sind, gearbeitet. Diese Operationen werden dann im Datensystem zum Beispiel durch Zugriffspfadabbildungen in eine Form umgewandelt, die von der satzorientierten Schnittstelle interpretiert werden kann. Im Zugriffssystem wird dann entschieden, mit welcher Zugriffsmethode

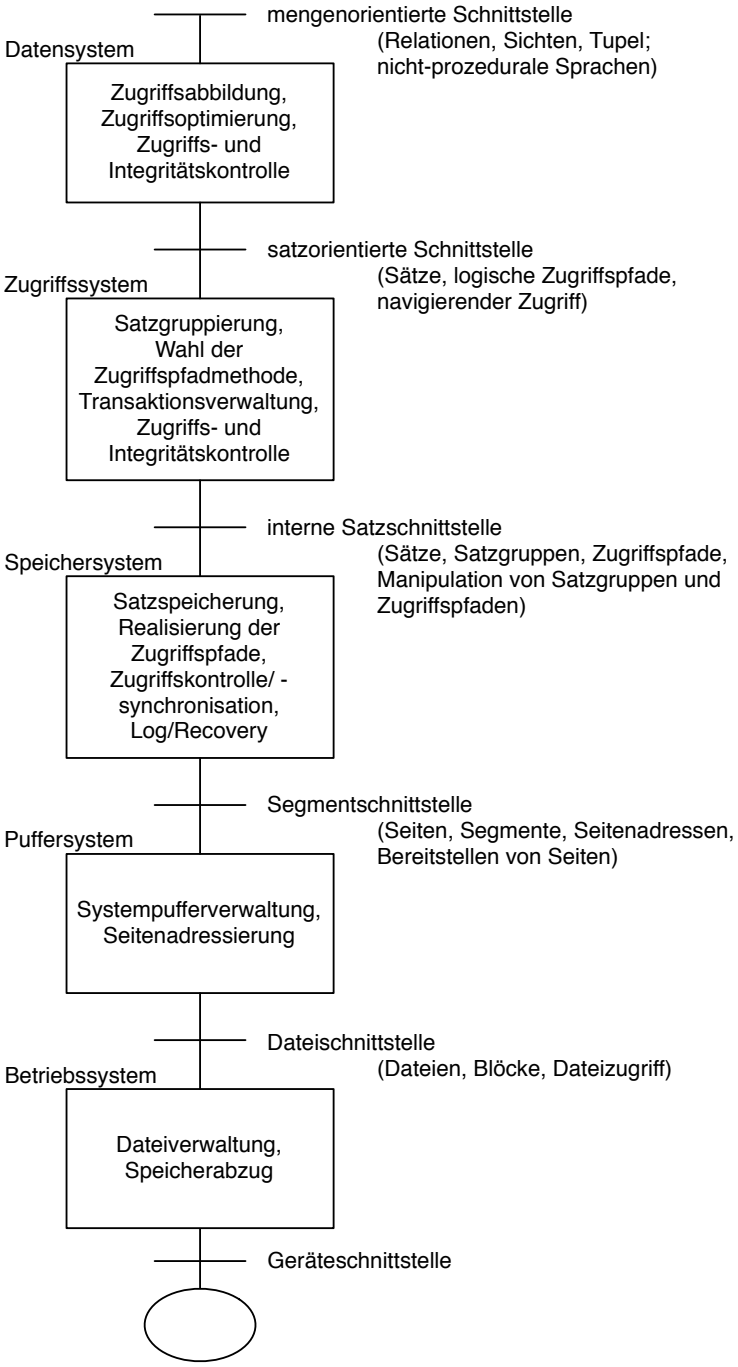

Abbildung 2.3: Fünf-Schichten-Architektur in Anlehnung an [LS87] und mit zusätzlichen Bezeichnungen von [SHS05].

auf die Daten zugegriffen werden soll. Damit dabei keine Fehler bei konkurrierenden Zugriffen von mehreren Benutzern entstehen, wird die Reihenfolge der Zugriffe durch eine Transaktionsverwaltung geregelt. Wurden die Operationen in eine zulässige Reihenfolge überführt, wird nachfolgend im Speichersystem eine Abbildung der Objekte der internen Satzschnittstelle auf den internen Adressraum vollzogen. Damit bei eventuell

auftauchenden Fehlern keine Daten verloren gehen, wird durch eine Log und Recovery Komponente protokolliert, welche Operationen auf der DB durchgeführt wurden. Durch das Puffersystem wird dann auf die einzelnen Seiten zugegriffen, die von der Systempufferverwaltung bereitgestellt werden. Die Bereitstellung der Dateien, in denen die Seiten gespeichert sind, wird dann durch das Betriebssystem geregelt, das dann durch die Geräteschnittstelle auf die Hardware des Systems zugreift [SHS05].

Für eine konkretere Betrachtung von DBS reicht es jedoch nicht aus, ihre Architekturen zu analysieren. Es ist vor allem notwendig, die wichtigsten Komponenten und ihre Funktionsweise zu kennen. Aus diesem Grund wird im folgenden Abschnitt näher auf die Komponenten eines DBS eingegangen.

2.3 Komponenten eines DBS

In Abbildung 2.4 ist ein Überblick über die von Vossen als besonders relevant eingestuften Komponenten eines DBMS gegeben [Vos94]. In dieser Abbildung sind die Komponenten in fünf Schichten eingeordnet, die nicht mit den Schichten der Fünf-Schichten-Architektur von Senke [Mic73] übereinstimmen. Bei den hier verwendeten Schichten handelt es sich um die Ebene der Benutzersprache, der Anfrageverarbeitung, der Zugriffsstrukturen, der Code-Erzeugung, der Synchronisation paralleler Zugriffe und um die Ebene der Speicherverwaltung. Neben diesen fünf Ebenen gibt es drei Datenbestände, auf denen das DBMS arbeitet. Diese drei Bestände sind die Datenbank selbst, das Data Dictionary, in dem die vom DBMS über die Datenbank angelegten Metadaten gespeichert sind und das (interne) Logbuch, dessen Informationen beim Auftreten von Fehlern verwendet werden [Vos94].

Soll eine Operation auf der Datenbank ausgeführt werden, nimmt der *Input/Output Prozessor* die Operationen des Benutzers entgegen. Diese Operationen werden dann vom *Parser* auf ihre syntaktische Korrektheit überprüft. Bei dieser Verarbeitung werden gegebenenfalls Informationen aus dem *Data Dictionary* benötigt. Handelt es sich bei den Kommandos um eingebettete Kommandos, ist eine weitere Verarbeitung durch den *Precompiler* notwendig [Vos94]. Wurde die syntaktische Kontrolle abgeschlossen, wird durch die *Automatisierungskontrolle* überprüft, ob der Benutzer die Berechtigung besitzt, die von ihm angestrebten Operationen durchführen zu dürfen. Dabei werden wiederum Informationen aus dem *Data Dictionary* benötigt.
Sollen durch die Operationen Änderungen auf der Datenbank durchgeführt werden, muss eine *Integritätsprüfung* geschehen. Diese kann oft nur unter Zuhilfenahme von Informationen geschehen, die im *Data Dictionary* gespeichert sind. Bei einfachen Anfrageoperationen entfällt diese Integritätsprüfung. Damit die angestrebten Operationen schnellstmöglich durchgeführt werden, werden durch den *Optimierer* und eine *Zugriffsplanerstellung* mehrere Anfragepläne erstellt, die dann in Hinblick auf die zu erwartende Ausführungszeit verglichen werden. Bei der Abschätzung werden Informationen aus dem *Data Dictionary*, wie zum Beispiel die Größe der Tabellen und Informationen über existierende Indexe, verwendet. Der durch diesen Schritt erzeugte Zugriffsplan wird dann durch die *Code-Erzeugung* in ausführbaren Code umgewandelt.

Da der Benutzer in den meisten Fällen nicht allein auf der Datenbank arbeitet, existiert in vielen DBMS eine *Transaktionsverwaltung*. Sie stellt sicher, dass durch den

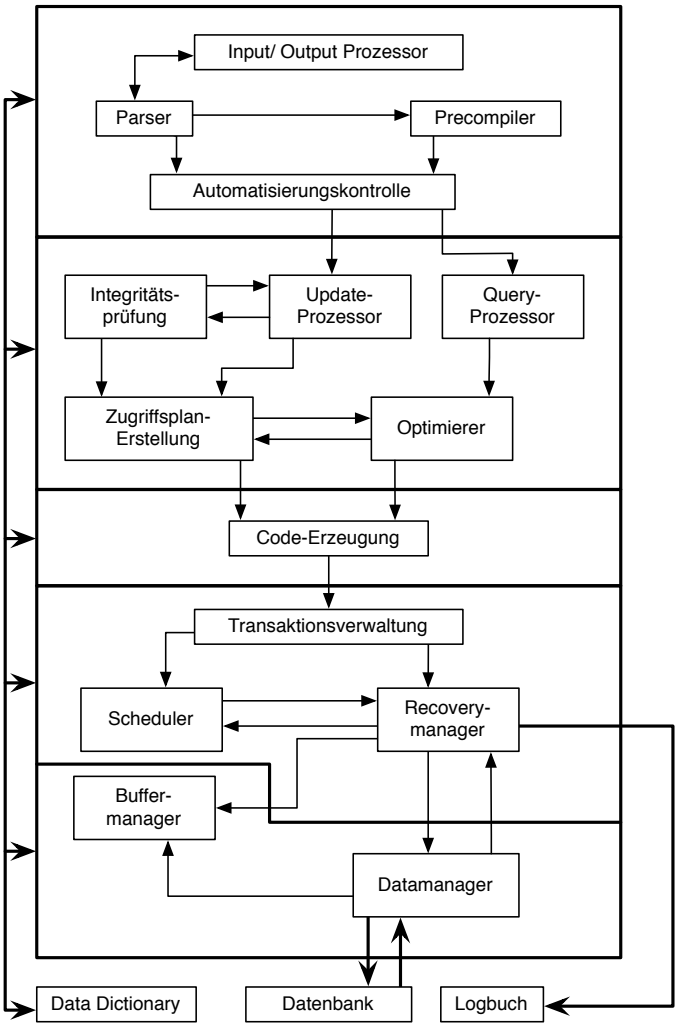

Abbildung 2.4: Wichtige Komponenten eines DBMS nach [Vos94].

Mehrbenutzerbetrieb keine ungewollten Seiteneffekte entstehen, wie zum Beispiel das Verlorengehen von Änderungen. Die *Transaktionsverwaltung* interagiert mit mehreren Komponenten. So werden die Operationen von Transaktionen durch einen *Scheduler* in eine Reihenfolge gebracht, die keine Konflikte verursacht. Wird eine Transaktion aufgrund von Problemen abgebrochen, muss sichergestellt werden, dass Änderungsoperationen, die durch die Transaktion schon durchgeführt wurden, wieder zurückgesetzt werden, damit unter anderem die Atomaritätseigenschaft der Transaktion gewährleistet wird. Damit diese vorhergehenden Zustände wieder rekonstruiert werden können, werden durch den *Recoverymanager* alle Änderungen innerhalb eines *Logbuches* protokolliert. Diese Informationen werden auch bei einem Absturz des Systems verwendet, um die Zustände abgeschlossener Transaktionen wieder zu rekonstruieren. Deshalb ist es notwendig, dieses *Logbuch* auch persistent auf einem Sekundärspeichermedium zu speichern. Nachdem die Protokollierung vollzogen ist, werden unter Zuhilfenahme des *Buffer-* und des *Datamanager* die Operationen auf der DB durchgeführt. Der *Buffer-* und der *Datamanager* dienen zur Verwaltung des Hauptspeichers und zur effizienten

Administration der einzelnen Speicherbereiche der DB, auf denen die Datensätze gespeichert sind [Vos94]. Von diesen Komponenten werden Metadaten benötigt, durch die beschrieben wird, wie die gespeicherten Informationen eines Datensatzes zu interpretieren sind [SHS05].

Es werden also von vielen Komponenten unterschiedlichste Metadaten verwendet, die alle auf dem Sekundärspeichermedium gespeichert werden. Des Weiteren werden durch das Logbuch ältere Versionen der Datensätze oder Informationen gespeichert, aus denen die Datensätze rekonstruiert werden können. Daher müssen diese Datenbestände mit betrachtet werden, wenn Datensätze nicht rekonstruierbar aus einem DBS gelöscht werden sollen.

2.4 Persistente Speicherung

Wie im vorhergehenden Abschnitt beschrieben, speichern unterschiedliche Komponenten eines DBS verschiedenartige Informationen über einen Datensatz auf dem Sekundärspeichermedium. Soll ein Datensatz aus dem DBS forensisch sicher gelöscht werden, ist es notwendig, diese Informationen zu löschen oder dahingehend zu verändern, dass der Datensatz aus ihnen nicht wieder rekonstruiert werden kann.

Die Rekonstruktion von bereits veränderten Werten ist jedoch nicht nur in Fehlerfällen sinnvoll. So kann durch die gespeicherten Informationen herausgefunden werden, wann durch welchen Benutzer Änderungen auf der Datenbank durchgeführt wurden. Diese Informationen können vor allem dann hilfreich sein, wenn sich unberechtigte Personen Zugang zu der DB verschafft haben [Fow08, Mar09], oder wenn versehentlich ungewollte Operationen auf der Datenbank durchgeführt wurden. Da es jedoch notwendig ist, alle Informationen über einen Datensatz zu löschen, wenn der Datensatz forensisch sicher gelöscht werden muss, kann bei einer ungewollten Veränderung der ursprüngliche Datensatz nicht wieder hergestellt werden. Somit muss davon ausgegangen werden, dass nur berechtigte Personen auf dem DBS arbeiten, die auch nur gewollte Operationen auf der DB durchführen.

Zur besseren Übersichtlichkeit werden die vom DBS gespeicherten Informationen, in dieser Arbeit in drei Gruppen eingeteilt. Diese Gruppen sind dabei angelehnt an die drei Datenbestände, die in Abschnitt 2.3 unterhalb den Komponenten des DBMS stehen. Die erste Gruppe bildeen dabei die *Datensätze* selbst, die in der DB gespeichert sind. Der zweite Bereich bilden die *Metadaten*, die vom DBMS zur Verwaltung der Daten genutzt werden und im Data Dictionary verwaltet werden. In der dritten und letzten Gruppe werden die *historisierten Informationen der Datensätze* zusammengefasst. Diese Informationen sind notwendig, um bei eventuellen Fehlerfällen die Datensätze verlustfrei wiederherzustellen zu können. Dabei werden diese Daten in einigen DBS nicht nur im Logbuch gespeichert, sondern auch innerhalb der DB.

2.4.1 Datensätze

Werden Datensätze aus der DB gelöscht und ihre Speicherbereiche nicht überschrieben, entstehen die im Abschnitt 2.1 beschrieben Slacks. Dabei handelt es sich um Speicherbereiche, in denen Daten vorhanden sind, die nicht mehr verwendet werden. Sind die Slacks

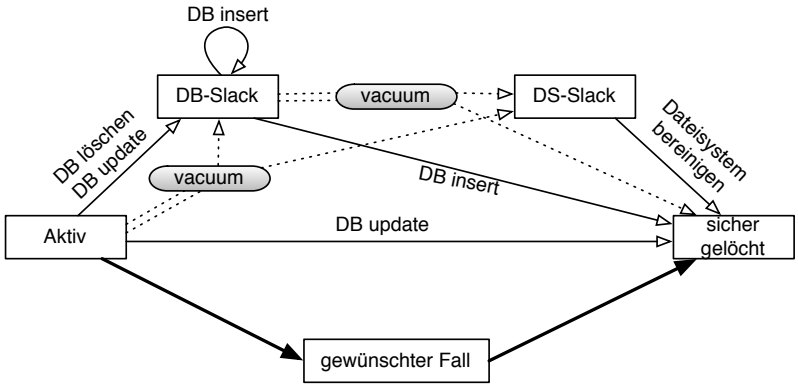

Abbildung 2.5: Zustände, die die Werte eines Datensatzes annehmen können. In Anlehnung an [MLS07] und [SML07].

durch Operationen entstanden, die von DBMS durchgeführt wurden, kann zwischen zwei Arten von Slacks unterschieden werden. Bei den beiden Arten handelt es sich um Datenbank (DB-) und Dateisystem (DS-) Slacks [SML07]. Dabei sind DB-Slacks nicht überschriebene Speicherbereiche, die in den Dateien existieren, die vom DBS gehalten werden. Dagegen sind DS-Slacks alte Datensatzwerte, deren Speicherbereich schon an das Dateisystem gegeben wurde, deren Werte aber noch auf dem Sekundärspeichermedium gespeichert sind. Eine Unterscheidung dieser beiden Arten ist wichtig, weil DB-Slacks nicht mittels Methoden gelöscht werden können, die im Dateisystem implementiert sind [MLS07]. Zur Verwaltung des Speicherbereichs eines DBMS und damit auch indirekt zur Verwaltung existierender DB-Slacks existiert in vielen DBMS ein `VACUUM` Prozess. Datenbankmanagementsysteme, in denen ein solcher Prozess existiert, sind beispielsweise PostgreSQL, MySQL, IBM DB2 und SQLite [SML07].

Wird dieser `VACUUM` Befehl ausgeführt, sind zwei Effekte der Repräsentation der Datensätze auf der Festplatte zu erkennen. Der erste ist, dass die Datensätze reorganisiert werden. Diese Reorganisation ist mit der Defragmentierung auf Speicherebene zu vergleichen. Das bedeutet, dass Datensätze auf Speicherbereiche geschrieben werden, an deren Stelle die Werte eines nicht mehr verwendeten Datensatzes gespeichert waren. Der zweite Effekt taucht auf, wenn die Datensätze innerhalb der DB auf Seiten verwaltet werden. Existiert eine solche Verwaltung, können durch die Umsortierung der Datensätze Seiten entstehen, die nicht mehr benötigt werden. Die Speicherbereiche dieser Seiten werden infolgedessen von dem DBS an das Dateisystem freigegeben. Durch diese Freigabe entstehen DS-Slacks [SML07, Mar09]. Dabei ist zu beachten, dass durch die Umsortierung und die eventuelle Freigabe von Seiten an das Dateisystem auch DB- und DS-Slacks für Datensätze entstehen können, die noch von dem DBS verwendet werden.

Zur Übersicht werden in Abbildung 2.5 die Zustände gezeigt, die die Werte eines Datensatzes durchlaufen können. Wird ein Datensatz erzeugt und vom DBMS verwendet, wird sein Zustand als *Aktiv* bezeichnet. Besitzt der Datensatz diesen Zustand, kann er mittels SQL oder anderer Anfragesprachen von dem DBS verwendet werden. Soll der

Datensatz mittels einer `DELETE` Operation gelöscht oder mit einer `UPDATE` Operation verändert werden, so kommt es in vielen DBS vor, dass der Datensatz oder Teile des Datensatzes innerhalb eines DB-Slacks weiter existieren. So wurde von Stahlberg et al. innerhalb eines Experimentes gezeigt, dass in PostgreSQL 100% der Informationen von gelöschten Datensätzen innerhalb von DB-Slacks weiter existierten, bis die `VACUUM` Operation aufgerufen wird [SML07]. Dieses Verhalten ist durch die Tatsache zu erklären, dass in PostgreSQL für jede veränderte Version eines Datensatzes und für jeden neuen Datensatz ein neuer Speicherbereich zur persistenten Speicherung verwendet wird [Sto87]. Werden bei einer `UPDATE` Operation, die auf einem Datensatz durchgeführt werden soll, die neuen Werte des Datensatzes an die Speicherstellen geschrieben, an denen schon die alten Werte des Datensatzes gespeichert wurden, kann es vorkommen, dass die alten Werte des Datensatzes sicher gelöscht sind. Dies ist jedoch nur dann der Fall, wenn durch die neuen Werte des Datensatzes eine größere oder mindestens gleiche Speicherlänge benötigt wird, die schon von den alten Werten verwendet wurde. Besitzt der neue Datensatz eine geringere Speicherlänge, sind Teile der alten Werte des Datensatzes noch in DB-Slacks vorhanden. Existieren die Werte eines nicht mehr verwendeten Datensatzes in einem DB-Slack weiter, kann es vorkommen, dass diese Werte beim Einfügen eines neuen Datensatzes in die DB überschrieben werden. Dabei kann es auch wieder aufgrund der unterschiedlichen Länge von Datensätzen passieren, dass nur Teile des nicht mehr benötigten Datensatzes überschrieben werden.

Beim Aufruf des `VACUUM` Prozesses, werden die Datensätze reorganisiert. Dabei ist es möglich, dass die Werte eines verwendeten Datensatzes auf die Speicheradressen eines nicht mehr benötigten Datensatzes kopiert werden. Werden die alten Speicheradressen des Datensatzes dabei nicht überschrieben, entsteht im Normalfall ein DB-Slack. Kommt es bei der Umsortierung der Datensätze vor, dass ganze Seiten nicht mehr benötigt werden, tritt der Fall auf, dass die Werte des Datensatzes in einem DS-Slack weiter existieren. Außerdem kann es durch die Verwendung dieses Prozesses vorkommen, dass Datensätze, deren Werte schon in DB-Slacks existieren, durch noch verwendete Datensätze überschrieben und somit sicher gelöscht werden. Es kann jedoch auch der Fall auftauchen, dass der Speicherbereich einer DB Seite auf der ein DB-Slack vorhanden ist, an das Dateisystem übergeben wird und somit DS-Slacks entstehen.

Wird der `VACUUM` Prozesses verwendet, kann demnach nicht sichergestellt werden, dass ein gelöschter Datensatz auch wirklich sicher gelöscht ist. Durch diesen Prozess können sogar von Datensätzen, die noch von der DB verwendet werden, Kopien erzeugt werden, die in DB-Slacks oder auch DS-Slacks weiter existieren.

Stahlberg et al. stellten exemplarisch für eine MySQL Speicher-Engine eine Methode vor, die die Entstehung eines Slacks beim Löschen eines Datensatzes verhindert. In der vorgestellten Methode werden die Speicherbereiche überschrieben, auf denen der nicht mehr benötigte Datensatz gespeichert war. Bei den durchgeführten Evaluierungen der Ausführungszeit wurden von Stablberg et al. sehr kleine Laufzeiteinbußen gemessen. Diese waren so gering, dass in den durchgeführten Tests der Mittelwert der veränderten Methode innerhalb der Standardabweichung der unveränderten Tests lag [SML07]. Es wurden somit keine signifikanten Laufzeiteinbußen gemessen. Das Entstehen von Slacks von Datensätzen kann somit in effizienter Zeit verhindert werden. Die gilt jedoch nur,

wenn durch den `VACUUM` Prozess nicht schon eine Kopie des Datensatzes erzeugt wurde, die innerhalb eines Slacks weiter existiert.

2.4.2 Metadaten

Soll ein Datensatz sicher gelöscht werden, muss nicht nur verhindert werden, dass seine Werte in DB- oder DS-Slacks weiter existieren. Es muss weiterhin noch sichergestellt werden, dass keiner seiner Werte innerhalb von dem durch das DBS angelegte Data Dictionary weiter existiert. Dabei ist zu beachten, dass die Metadaten oft unabhängig von der physischen Repräsentation der Daten in der DB gespeichert sind [Dat04]. Da diese Daten nach einem Absturz des Systems vorhanden sein müssen, werden sie persistent auf dem Sekundärspeichermedium gespeichert. Ein Überblick über diese Metadaten und die Komponenten, von denen sie verwendet werden, ist in Tabelle 2.1 gegeben. Jedoch ist hier ihre Vollständigkeit nicht garantiert, da in einigen DBMS noch spezielle Metadaten gespeichert werden.

Komponente	Metadaten
Automatisierungskontrolle	Informationen über Zugriffsrechte der Benutzer.
Parser	Informationen über die Schemata von Tabellen.
Precompiler	Informationen über integrierte Kommandos.
Integritätsprüfung	Informationen über die Integritätseigenschaften von Tabellen.
Optimierer	Tabellenstatistiken (Verteilung der Daten, Anzahl der Tupel) Informationen über existierende Indexe und Tabellenschemata.
Buffer Manager	Informationen, welche Seiten als letztes geladen wurden und generell Statistiken über geladene Seiten.
Data Manager	Abbildung von gespeicherten Bits zu den Werten der Datensätze. Durch diese Abbildung wird beschrieben, wie die gespeicherten Bits interpretiert werden müssen.

Tabelle 2.1: Komponenten eines DBMS mit den von ihnen verwendeten Metadaten.

Diese Metainformationen müssen jedoch nicht alle betrachtet werden, wenn ein Datensatz sicher gelöscht werden soll. So werden zum Beispiel in den Schemainformationen einer Tabelle keine Informationen über die einzelnen Datensätze der Tabelle gespeichert. Jedoch müssen Tabellenstatistiken und Indexstrukturen betrachtet werden, da in ihnen Informationen über einzelne Datensätze gespeichert werden. Ein Beispiel für die Rekonstruktion von Datensätzen unter Zuhilfenahme von Tabellenstatistiken bietet Fowler [Fow08]. Er beschreibt, dass auf Histogramme in SQL Server mittels SQL Anweisungen direkt zugegriffen werden kann. Somit kann durch eine einfache analytische Anfrage der Unterschied zwischen den aktuell in der Datenbank existierenden Werten und in der Statistik gespeicherten Werten ermittelt werden. Durch diesen Unterschied

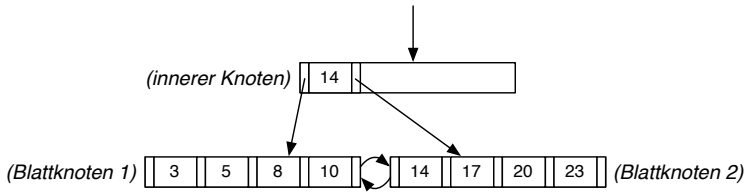

Abbildung 2.6: Aufbau eines B$^+$-Baums.

kann dann herausgefunden werden, welche Datensätze seit dem Erstellungszeitpunkt der Statistik gelöscht beziehungsweise neu hinzugekommen oder verändert worden sind. Außerdem werden in den Histogrammen in SQL Server 200 Beispielwerte als Grenzen der einzelnen Bereiche verwendet, was eine weitere Möglichkeit der Rekonstruktion von Datensätzen bietet.

Weiterhin speichern Indexstrukturen Werte von Datensätzen ab, um schnell auf die Datensätze zugreifen zu können. Soll ein Datensatz gelöscht werden, müssen die über den Datensatz gespeicherten Informationen mit gelöscht werden. Dabei können die Indexe nach verschiedenen Kriterien in unterschiedliche Klassen eingeteilt werden. Bei den Klassen handelt es sich um Primär- und Sekundärindexe, geclusterte und nicht geclusterte Indexe, um dicht und dünn besetzte Indexe und um Ein- oder Mehrkomponenten Indexe [SHS05]. Typische Indexstrukturen für DBMS sind Baumstrukturen wie B-, B$^+$-Bäume, ISAM-Strukturen (Index Sequencial Access Method) und Hash-Strukturen [Vos94].

Stahlberg et al. stellen in [SML07] dar, dass bei einem B$^+$-Baum verschiedene Möglichkeiten der Wiederherstellung von Daten bestehen. Die erste Möglichkeit besteht darin, dass beim Löschen von Datensätzen aus den Blattknoten die Werte des Datensatzes in internen Knoten weiter existieren können. Eine Visualisierung dieser Problematik ist in Abbildung 2.6 gegeben. Die Ursache für dieses Problem ist, dass die Werte von Datensätzen als Entscheidungskriterium für eine Suche innerhalb des Baums verwendet werden. Wird beispielsweise der Datensatz mit dem Wert 17 in dem Baum aus Abbildung 2.6 gesucht, wird zuerst im inneren Knoten (innerer Knoten) abgefragt, in welchem Blattknoten der Datensatz potentiell enthalten ist. Nachdem dies geschehen ist, wird dann, in diesem Beispiel, im Blattknoten 2 auf den Datensatz zugegriffen. Soll der Datensatz mit dem Wert 14 gelöscht werden, ist es notwendig, diesen Wert auch aus dem inneren Knoten zu entfernen. Wird dies nicht gemacht, kann der Wert des Datensatzes durch diese weiterhin gespeicherten Information wieder rekonstruiert werden. Es kann damit nötig sein, den B$^+$-Baum zu reorganisieren, damit aus ihm ein Datensatz nicht rekonstruierbar gelöscht wird.

Zur Lösung dieser Probleme wurde von Stahlberg et al. exemplarisch die B$^+$-Baum Implementierung in MySQL verändert. Dabei wurden von ihnen die `UPDATE`, `DELETE` und `INSERT` Operation des B$^+$-Baumes dahingehend modifiziert, dass durch sie keine Slacks entstehen. Bei einer Evaluierung der entwickelten Methode hat sich herausgestellt, dass keine zusätzliche I/O Operation nötig ist, da nur auf Seiten gearbeitet wird, die schon im Arbeitsspeicher vorliegen. Daher sind durch diese Veränderung keine relevanten Lauf-

zeitveränderungen zu erwarten [SML07]. Als zweites Problem wird angegeben, dass der Aufbau von B^+-Bäumen von der Reihenfolge abhängt, in der die Datensätze in den Baum eingefügt werden. Es handelt sich hier somit um eine historisch abhängige Indexstruktur. Dadurch kann ein Eindringling durch den Aufbau des B^+-Baums Rückschlüsse über die Reihenfolge der Operationen ziehen, die auf dem B^+-Baum durchgeführt worden sind. Dies ist jedoch nur dann möglich, wenn nicht alle Datensätze im Wurzelelement des Baumes gespeichert sind. Dies ist jedoch nur dann der Fall, wenn der Baum wenige Werte umfasst [SML07, MLS07].

Neben den Metadaten, die in vielen DBMS gespeichert werden, gibt es in einigen DBMS noch spezielle Metadaten, durch die zum Beispiel bei Anfragen eine Laufzeitverbesserung erzielt wird. So existiert in MonetDB, einem spaltenorientierten DBMS, eine Komponente, durch die Zwischenergebnisse von Anfragen gespeichert werden. Dabei wird bei jeder Speicherung abgewogen, wie wichtig eine Materialisierung dieses Ergebnisses ist und wie groß ihr Berechnungsaufwand ist [IKNG09]. Da es sich bei diesen Zwischenergebnissen um aggregierte Datensatzwerte handeln kann, können durch sie Werte von Datensätzen wieder rekonstruiert werden. Soll ein Datensatz aus einem MonetDB DBS nicht rekonstruierbar gelöscht werden, müssen die Zwischenergebnisse, an deren Konstruktion der Datensatz beteiligt war, auch mit gelöscht oder dahingehend verändert werden, dass der Datensatz nicht rekonstruiert werden kann.

2.4.3 Historische Daten zur Fehlertoleranz

Neben der Verwendung von Metadaten bieten Logbucheinträge und historische Werte des Datensatzes eine andere Möglichkeit, gelöschte Daten wieder zu rekonstruieren. Generell ist aber eine Protokollierung der Operationen, die auf der DB durchgeführt wurden nötig, um bei Fehlern den aktuellen, konsistenten Zustand der DB wieder herzustellen. Die in einer DB möglichen Fehlerszenarien können in drei Klassen eingeteilt werden [BHG87]:

1. Transaktionsfehler
2. Systemfehler
3. Mediafehler

Transaktionsfehler entstehen, wenn eine Transaktion zum Beispiel durch den Benutzer, das System oder Fehler des Anwendungsprogrammes abgebrochen wird. Sie werden auch als lokale Fehler bezeichnet, da sie auf eine Transaktion begrenzt sind. Bei der zweiten Klasse von Fehlern handelt es sich um Systemfehler. Systemfehler sind Fehler, die das DBS zum Absturz bringen, bei denen jedoch der persistente Speicher der Daten unversehrt bleibt. Ist bei einem Fehler der Sekundärspeicher betroffen, wird dieser Fehler als Mediafehler bezeichnet. Um bei Mediafehlern wenig Wiederherstellungsaufwand betreiben zu müssen, wird von vielen Datenbanken in regelmäßigen Abständen eine Kopie der Datenbank erzeugt [SHS05]. Diese Kopie wird dann sicher verwahrt und bei Mediafehlern wieder eingespielt.

Soll ein Datensatz sicher gelöscht werden, müssen auch seine alten Einträge aus dem Logbuch mit gelöscht werden. Dies muss geschehen, weil der Datensatz durch die im Logbuch gespeicherten Informationen wiederhergestellt werden kann. Geschieht diese Betrachtung nicht, ist es möglich, eigentlich gelöschte Daten wieder herzustellen [Lit07a]. Handelt es sich bei dem Logbuch um ein feingranulares Transaktionslogbuch, kann diese Wiederherstellung auch in effizienter Zeit geschehen [ZFZ+08]. Dabei kann nach Fowler die Wiederherstellung von gelöschten Daten auch automatisiert geschehen. So wird von ihm ein Skript bereitgestellt, mit dem eine solche Wiederherstellung der Daten durchgeführt werden kann [Fow08].

Stahlberg et al. stellen eine Lösung vor, durch die verhindert werden kann, dass ein Datensatz durch Informationen aus Logbuch wiederhergestellt werden kann. Bei der gezeigten Methode werden die Einträge des Logbuchs verschlüsselt. Dabei wird für jeden Speicherbereich ein anderen Schlüssel zum Verschlüsseln verwendet. Soll eine Änderung auf dem Speicherbereich durchgeführt werden, so wird ein *Before* Image des Speicherbereichs angelegt, das mit einem neuen Schlüssel verschlüsselt wird. Dabei wird der neue Schlüssel aus dem alten Schlüssel, anhand einer Hash-Kette, generiert und der alte wird aus dem System entfernt [SML07]. An dieser Stelle sei erwähnt, dass diese Methode nur bei einem physisch protokollierten Logbuch funktioniert.

Neben dem Logbuch und alten Datenbankkopien gibt es in einigen DBMS noch eine zusätzliche Möglichkeit, durch die alte Zustände von gelöschten Datensätzen rekonstruiert werden können. Diese Möglichkeit ist eng mit der Möglichkeit der Rekonstruktion von Datensätzen aus der DB verbunden. Sie existiert, wenn als Transaktionsverwaltung des DBMS das nicht sperrende Multiversion Concurrency Control Protokoll (MVCC) verwendet wird. Dieses Verfahren wird zum Beispiel in PostgreSQL, IBM DB2 und Oracle genutzt. Werden Datensätze innerhalb einer Transaktion verändert, wird eine Kopie des Datensatz angelegt, auf der dann die Veränderungen durchgeführt werden. Konnte diese Transaktion erfolgreich abgeschlossen werden, wird ein interner Zeiger auf den Datensatz derart abgeändert, dass er auf die neue Version des Datensatzes zeigt [WV01]. Da die alte Version des Datensatzes, nachdem sie als veraltet markiert wurde, jedoch nicht überschrieben wird, existieren die Werte des Datensatzes auf Dateiebene weiter, bis ihr Speicherbereich zum Beispiel durch den `VACUUM` Prozess von einem anderen Datensatz überschrieben wird oder in einen DS-Slack übergeht.

2.5 Zusammenfassung

Zu Beginn dieses Kapitels wurde erläutert, wie es möglich ist, personenbezogene Daten auf Dateiebene zu löschen. Dabei wurde herausgearbeitet, welche Möglichkeiten bestehen, bereits gelöschte Daten wieder zu rekonstruieren und wo die Grenzen dieser Rekonstruktionsmöglichkeiten liegen. Werden diese Daten durch ein DBS verwaltet, wird die Aufgabe des nicht rekonstruierbaren Löschens komplexer, da nicht nur die Daten selbst, sondern auch Metadaten dieser Daten mit gelöscht werden müssen. Zur Einführung in diese Problematik wurde auf Architekturen von RDBMS und auf ihre Komponenten eingegangen. Nachfolgend dazu wurde gezeigt, durch welche Arten von Informationen gelöschte Datensätze wieder rekonstruiert werden können. Außerdem

wurden bereits existierenden Lösungen dargelegt, durch die eine Rekonstruktion der Datensätze durch die Informationen die von einzelnen Komponenten gespeichert werden, verhindert werden kann. Des Weiteren wurde am Beispiel von MonetDB gezeigt, dass in einzelnen DBMS besondere Informationen über die Daten gespeichert werden, die auch mit betrachtet werden müssen.

3. Konzepte für ein forensisch sicheres DBS

Der Fokus dieser Arbeit liegt darauf, einen Einblick in die generellen Problematik des forensisch sicheren Löschens von Datensätzen aus Datenbanksystemen zu geben. Diesbezüglich wurde im vorhergehenden Kapitel aufgezeigt, welche Informationen von einem DBS gespeichert werden. Soll ein Datensatz nicht rekonstruierbar aus dem DBS gelöscht werden, muss eine gesamtheitliche Betrachtung dieser Informationen geschehen. Der Grund dafür ist, dass wie schon im Abschnitt 2.4 beschrieben, eine Rekonstruktion der Daten nicht nur durch Werte geschehen kann, die in der Datenbank gespeicherten sind. Die Daten können auch durch Metainformationen und durch Informationen, die im Logbuch gespeichert sind, wiederhergestellt werden. Da diese Informationen von unterschiedlichen Komponenten gespeichert werden, ist es notwendig, jede Komponente dahingehend gesondert zu betrachten, in wieweit von ihr wichtige Informationen gespeichert werden.

In diesem Kapitel werden dazu einige Begriffe definiert. Anschließend werden allgemeine Ansatzpunkte betrachtet, die die DBS liefern. Nachfolgend dazu erfolgt eine nähere Betrachtung der einzelnen Komponenten, durch die Informationen über Datensätze gespeichert werden.

3.1 Begriffserklärung und Vorbedingung

In diesem Abschnitt werden Begriffe definiert, durch die eine differenzierte Betrachtung der Problematik des forensisch sicheren Löschens in Datenbankmanagementsystemen ermöglicht wird.

Arten von Metadaten

Bei den Metadaten, die von einem DBS angelegt werden, kann zwischen zwei Arten von Metadaten unterschieden werden. Dabei handelt es sich um datensatzunabhängige Metadaten und um datensatzabhängige Metadaten.

Definition 3.1 (Datensatzunabhängige Metadaten)**.** *Bei datensatzunabhängigen Metadaten handelt es sich um Metadaten, die die Struktur der Daten beschreiben oder die generelle Informationen über die Daten liefern. Dabei dürfen in ihnen keine Informationen über die eigentlichen Werte der Daten enthalten sein, die zu einem bestimmten Zeitpunkt in der Datenbank vorhanden sind.*

Definition 3.2 (Datensatzabhängige Metadaten)**.** *Datensatzabhängige Metadaten sind Metadaten, in denen Informationen über die Werte von Datensätzen gespeichert sind. Dabei müssen nicht die spezifischen Werte der Datensätze in ihnen gespeichert sein. Es reicht aus, wenn durch sie Rückschlüsse auf die Werte der Datensätze gezogen werden können.*

Bei datensatzunabhängigen Metadaten handelt es sich nach der Definition zum Beispiel um Schemainformationen einer Tabelle oder um die Integritätseigenschaften einer Tabelle. Beispiele für datensatzabhängige Metadaten sind in dem Zusammenhang die von Indexstrukturen oder auch von Tabellenstatistiken angelegten Informationen über die Werte der Datensätze.
Im Hinblick auf diese Einteilung müssen nicht alle Metadaten des DBS näher betrachtet werden, wenn ein Datensatz forensisch sicher gelöscht werden soll. Es reicht aus, die datensatzabhängigen Metadaten des DBS näher zu analysieren, da in den datensatzunabhängigen Metadaten keine datensatzspezifischen Informationen gespeichert werden.

Forensisch sicheres und nicht rekonstruierbares Löschen

Im Gegensatz zu ganzen Dateien sind die Datensätze in DBS meist von viel geringerer Größe. Wie in Abschnitt 2.1 beschrieben, können schon nach einmaligem Überschreiben von Speicherbereichen nur noch einzelne Bits der ehemals auf den Bereichen gespeicherten Informationen mit einer Wahrscheinlichkeit von über 50% rekonstruiert werden [Wri08]. Da zur Speicherung von wichtigen Informationen oft mehr als nur einzelne Bits benötigt werden, reicht also ein einfaches Überschreiben der Speicherbereiche aus, damit die vorher auf den Bereichen gespeicherten Werte nicht rekonstruiert werden können. Dadurch kann die Wahrscheinlichkeit der Rekonstruktion ganzer Datensätze hinreichend verringert werden, was in der Praxis meinst schon ausreicht, damit die Daten nicht rekonstruiert werden können [Fox09]. Jedoch muss unterschieden werden, ob noch einzelne Bits von Datensätzen wiederherstellt werden können, oder ob keine Möglichkeit besteht, Bestanteile der Daten wiederherstellen zu können.

Definition 3.3 (Nicht rekonstruierbares Löschen)**.** *Datensätze werden als nicht rekonstruierbar gelöscht angesehen, wenn der Speicherbereich, auf dem sie gespeichert waren, ein mal mit zufällig generierten Daten überschrieben wurde.*

Definition 3.4 (Forensisch sicheres Löschen)**.** *Datensätze gelten als forensisch sicher gelöscht, wenn keine Möglichkeit besteht, auch nur ein Bit der Daten wieder rekonstruieren zu können.*

Anhand der oben gegebenen Definition kann ein Datensatz dann als forensisch sicher gelöscht angesehen werden, wenn seine Speicherbereiche 33 mal mit zufällig generierten

Daten überschreiben wurden [Sch07] oder wenn auf ihnen die von Gutmann vorgestellte Methode [Gut96] angewendet wurde. Bei dieser Methode werden die Speicherbereiche insgesamt 35 mal mit den vorgegebenen Pattern und mit zufälligen Daten überschrieben. Wird im Laufe dieser Arbeit nicht genauer spezifiziert, wie oft die Speicherbereiche eines gelöschten Datensatzes überschrieben wurden, wird der als **sicher gelöscht** bezeichnet.

Domänenwissen

Je nach Komplexität der Daten, die in der Datenbank gespeichert sind, können Abhängigkeiten zwischen den Werten der Datenbank bestehen. Dabei können Abhängigkeiten zwischen zwei verschiedenen Datensätzen, aber auch Abhängigkeiten zwischen den Attributen eines Datensatzes existieren.
Soll ein Datensatz sicher gelöscht werden, ist es notwendig, die Werte all seiner Attribute sicher zu löschen. Dabei können die Attribute des Datensatzes, je nachdem, welcher Normalform die Datenbank unterliegt, auch in unterschiedlichen Tabellen existieren. Bei dem durchgeführten Löschen des Datensatzes muss dies natürlich berücksichtigt werden. Sind noch einzelne Attribute des personenbezogenen Datensatzes innerhalb des DBS vorhanden, während die anderen Attribute sicher gelöscht wurden, wird der Datensatz im nachfolgenden als partiell sicher gelöscht bezeichnet.

Definition 3.5 (Partiell sicher gelöschte Daten)**.** *Ein Datensatz wird als partiell sicher gelöscht angesehen, wenn mindestens eines seiner Attribute noch vollkommen rekonstruierbar ist. Es darf jedoch nicht die Möglichkeit bestehen, den ganzen Datensatz wieder rekonstruieren zu können.*

Dabei kann die Rekonstruktion der Attribute auch durch datensatzabhängige Metadaten geschehen. Gibt es Abhängigkeiten zwischen unterschiedlichen Datensätzen, können je nach Anwendungsfall Rückschlüsse auf die Werte eines gelöschten Datensatzes durch noch existente Datensätze gemacht werden. Bei dieser Problematik ist zu beachten, dass das DBS über kein Domänenwissen verfügt, um dies zu verhindern. Damit eine vollkommene oder partielle Wiederherstellung von Datensätzen auf Basis dieser Informationen verhindert werden kann, müssen durch den Administrator der Datenbank auf der Grundlage seines Domänenwissens Regeln festgelegt werden. Dabei muss beachtet werden, dass durch die Regeln keine datensatzabhängigen Metadaten persistent auf dem Sekundärspeicher gespeichert werden, weil sonst Datensätze auf Grundlage dieser Regeln wieder rekonstruiert werden können.

3.2 Allgemeine Strategien zum sicheren Löschen von Datensätzen

Ziel dieser Arbeit ist die Entwicklung einer Lösung, durch die Datensätze sicher aus Datenbankmanagementsystemen gelöscht werden können. Dabei soll die entstehende Lösung einfach in bestehende DBMS integrierbar sein.

Generell soll das sichere Löschen eines Datensatzes nicht nachträglich nach dem Löschen des Datensatzes aus der DB durch ein zusätzliches Programm durchgeführt werden.

Solch eine Lösung hätte unter anderem den Nachteil, dass durch das Programm nicht sichergestellt werden kann, dass auch alle Informationen des Datensatzes sicher gelöscht werden, wenn Abhängigkeiten zwischen den Datensätzen der Datenbank existieren. Außerdem ist es notwendig, dass das Programm auch schreibend auf die Metadaten des DBMS zugreift. Dabei kann ein derartigen Zugriff zu unvorhersehbaren Seiteneffekten führen. Ein weiteres Problem besteht darin, dass solch ein Programm immer erst dann die Daten verändern darf, wenn gerade keine Kopie der Daten im Arbeitsspeicher der Datenbank vorhanden ist. Würde dies nicht beachtet werden, kann es vorkommen, dass die Änderungen, die das Programm durchführt, vom Datenbankmanagementsystem wieder rückgängig gemacht werden. Es ist somit notwendig, die Funktionalität des sicheren Löschens von Datensätzen in das DBS zu integrieren.

Damit beim Überschreiben der Speicherbereiche nicht unnötig viel Ausführungszeit entsteht, ist eine Lösung zu bevorzugen, durch die so wenig wie möglich zusätzliche Operationen und Transformationen vom DBMS durchgeführt werden müssen. Da, wie schon erwähnt, der Datensatz auch nachdem er gelöscht wurde mit Metadaten oder Informationen aus dem Logbuch wieder rekonstruiert werden kann, ist es notwendig, mit den Erweiterungen der Funktionalität nicht erst beim Schreiben des Datensatzes anzusetzen. Es muss schon bei einer weitaus früheren Stelle, angesetzt werden, wenn ein Datensatz sicher gelöscht werden soll.

Zur Veranschaulichung, welche allgemeinen Ansatzpunkte in den DBMS existieren, sind in Abbildung 3.1 die Schichten gezeigt, die eine Operation durchlaufen muss, wenn sie auf einem DBS ausgeführt werden soll. Dabei ist das DBMS des DBS in die in der Fünf-Schichten-Architektur beschriebenen Schichten unterteilt.

Abbildung 3.1: Schichten der Transformation.

Wird mit den Veränderungen an einer der oberen Schichten angefangen, ist es notwendig, auch die darunter liegenden Schichten mit zu verändern, damit die Veränderung auch an die unteren Schichten propagiert wird. So ist generell anzunehmen, dass je weiter oben die Veränderungen beim DBMS durchgeführt werden, desto mehr Komponenten müssen durch die Veränderungen angepasst werden. Jedoch ist auch zu vermuten, dass nur durch Lösungen, die an einer der obersten Schichten ansetzen, ein vollkommenes sicheres Löschen von Datensätzen durchgeführt werden kann.

Mengenorientierte Schnittstelle

Lösungen, die an der mengenorientierten Schnittstelle ansetzen, haben eine Erweiterung der Funktionalität aller Komponenten des DBMS zur Folge. Dabei zählen alle Erweiterungen der SQL Syntax zu den Veränderungen. So könnte zum Beispiel das `DELETE` und das `UPDATE` Statement um ein optionales zusätzliches Schlüsselwort erweitert werden, das besagt, dass die Werte des Datensatzes sicher gelöscht werden sollen. Durch diese SQL Erweiterung währe es dann möglich genau festzulegen, welche Datensätze sicher gelöscht werden sollen und welche nicht. Eine andere Möglichkeit, die auch an der mengenorientierten Schnittstelle ansetzt, ist eine Erweiterung des `CREATE TABLE` Statements um ein Schlüsselwort, durch das festgelegt werden kann, dass Werte einer Spalte der Tabelle wichtige personenbezogene Daten beinhalten. Der Vorteil dieser Lösung gegenüber einer Lösung, die nur besagt, dass der ganze Datensatz sicher gelöscht werden muss, ist ihre Granularität. So muss bei einer Tabelle mit sehr vielen Spalten nicht der ganze Datensatz sicher gelöscht werden, wenn die wichtigen personenbezogenen Daten nur in einer Spalte gespeichert sind. Die Lösungsvariante, durch die das `CREATE TABLE` Statement erweitert werden würde, hätte als weiteren Vorteil, dass das DBMS schon bei der Erstellung der Tabelle Informationen besitzt, welche Spalten personenbezogene Daten beinhalten. Durch diese Informationen ist es beispielsweise möglich, zu entscheiden, über welche Attribute der Tabellen welche Indexe angelegt werden dürfen.

Wird mit den Veränderungen unterhalb der mengenorientierten Schnittstelle angesetzt, kann durch den Benutzer nicht mehr feingranular unterschieden werden, ob ein Datensatz sicher gelöscht werden soll oder ein einfaches Löschen für ihn ausreicht. Außerdem kann, wenn sowohl die Möglichkeit des nicht rekonstruierbaren Löschens als auch die des forensisch sicheren Löschens im DBS vorhanden sind, durch den Benutzer nicht mehr entschieden werden, welche Variante er verwendet.

Satzorientierte Schnittstelle

Werden die Erweiterungen des DBMS erst ab der satzorientierten Schnittstelle vollzogen, kann immer noch verhindert werden, dass der Datensatz nach seiner Löschung unter Zuhilfenahme von Metadaten und historischen Daten wieder rekonstruiert werden kann. Der Grund dafür ist, dass zwischen der mengenorientierten und der satzorientierten Schnittstelle nur datensatzunabhängige Metadaten, wie Informationen über das Schema einer Tabelle, verwendet werden. Er kann somit sicher gelöscht werden. Dies gilt unter der Voraussetzung, dass die Änderungen in allen darunter liegenden Sichten konsistent im Hinblick auf das Löschen der Informationen des Datensatzes durchgeführt werden. Es ist jedoch nicht mehr möglich zu unterscheiden, ob ein Datensatz sicher gelöscht werden soll oder ob ein einfaches Löschen des Datensatzes ausreicht.

Interne Satzschnittstelle

Setzt man mit der Lösung an der internen Satzschnittstelle an, kann nicht mehr sichergestellt werden, dass ein Datensatz sicher gelöscht wird. Der Grund dafür ist, dass im Zugriffssystem (Abbildung 2.3) datensatzabhängige Metadaten verwendet werden. Dabei handelt es sich um Metadaten, wie Indexstrukturen oder Tabellenstatistiken, durch die die Werte einzelne Attribute gespeichert sein können. Es besteht somit nur noch die Möglichkeit, einen Datensatz partiell sicher zu löschen. Dies gilt jedoch nur unter der Voraussetzung, dass nicht über jedes Attribut des Datensatzes ein Index angelegt wurde.
Jedoch kann verhindert werden, dass alte Versionen des Datensatzes durch Informationen aus dem Logbuch des DBS wieder rekonstruiert werden können.

Segmentschnittstelle und Dateischnittstelle

Die untersten Ansatzpunkte sind die Segmentschnittstelle und die Dateischnittstelle. Da im Puffersystem keine Informationen gespeichert werden, durch die ein Datensatz rekonstruiert werden kann, bieten Lösungen, die an der Segmentschnittstelle und an der Dateischnittstelle ansetzen, die gleichen Möglichkeiten des sicheren Löschens von Datensätzen. Generell kann durch den Ansatz an diesen Stellen nur noch verhindert werden, dass die Werte eines gelöschten Datensatzes auf Dateiebene weiter existieren.

Eine Möglichkeit zur Durchführung des sicheren Löschens von Datensätzen aus der DB setzt zu dem Zeitpunkt an, an dem der alte, nicht mehr benötigte Datensatz, auf der Seite, auf der er gespeichert ist, als veraltet markiert wird. So könnten zu diesem Zeitpunkt die Werte des Datensatzes mit anderen Daten überschrieben werden. Da zu diesem Zeitpunkt die Seite, auf der der Datensatz steht, von der Pufferverwaltung gehalten wird, sind bei dieser Lösung nur sehr geringe Laufzeiteinbußen zu erwarten. Jedoch kann mit einer Erweiterung an dieser Stelle nicht sichergestellt werden, dass keines der Bits des Datensatzes rekonstruiert werden können. Wird als Transaktionsverwaltung das MVCC verwendet, muss darauf geachtet werden, dass auch auf bereits gelöschte Datensätze weiter zugegriffen werden kann. Eine genauere Betrachtung der Problematik folgt in Abschnitt 3.3.

Generelle Grenzen der Ansatzpunkte

Ein Überblick über die bisher beschriebenen Startpunkte der Strategien zum sicheren Löschen in DBS ist in Tabelle 3.1 gegeben. Von ihnen bieten nur die Lösungen, die an der mengenorientierten und an der satzorientierte Schnittstelle ansetzen, die Möglichkeit, den Datensatz vollkommen sicher zu löschen. Doch auch diese Lösungen haben ihre Grenzen.

Wie schon in Abschnitt 2.4 gesagt, werden zur größeren Datensicherheit Kopien von Datenbanken angelegt. Dies geschieht unter anderem aus dem Grund, dass bei Medienfehlern nicht alle Änderungen, die auf der Datenbank seit ihrem Start durchgeführt wurden, wiederholt werden müssen. Diese Kopien oder auch Backups werden dann auf anderen Speichermedien gespeichert und sicher verwahrt. Problematisch ist deren Verwaltung, wenn die Datensätze der Datenbank aus rechtlichen Gründen nicht mehr

Startpunkt der Veränderung	Nicht gelöschte Informationen	Besonderheiten
mengenorientierte Schnittstelle	keine	Unterscheidung möglich, ob Datensätze sicher gelöscht werden sollen oder nicht.
satzorientierte Schnittstelle	keine	
interne Satzschnittstelle	datensatzabhängige Metadaten	
Segmentschnittstelle & Dateischnittstelle	datensatzabhängige Metadaten , Logbucheinträge	

Tabelle 3.1: Ansatzpunkt der Veränderungen mit den durch sie nicht betrachteten Informationen der Datensätze und Besonderheiten.

gespeichert werden dürfen. Es ist dann notwendig, die Daten aus den Kopien der Datenbank sicher zu entfernen. Jedoch besteht das Problem, dass von dem DBMS aus nicht auf diese Datenbestände zugegriffen werden kann. Ein sicheres Löschen der Datensätze ist somit von dem DBMS aus nicht durchführbar.

Außerdem ist es möglich, dass durch nebenläufige Prozesse Kopien von personenbezogenen Informationen angelegt werden. So kann von keiner der angegebenen Strategien verhindert werden, dass durch das Umsortieren der Datensätze beim Ausführen des `VACUUM` Prozesses DB- oder DS-Slacks entstehen. Damit diese Slacks verhindert werden, ist es notwendig,s die Funktionsweise des Prozesses um das Überschreiben von Datensätzen zu erweitern. So müssen die Speicheradressen eines Datensatzes überschrieben werden, nachdem er an eine neue Speicheradresse verschoben wurde.

Zusammenfassend kann durch die gezeigten Lösungen das sichere Löschen von Datensätzen nur dann sichergestellt werden, wenn die Datensätze von keinem anderen Prozess, durch Einwirkung des Benutzers oder durch ein Auslagern des Arbeitsspeichers kopiert oder verschoben wurden.

3.3 Besondere Komponenten und Prävention

Nachdem im vorhergehenden Abschnitt das DBMS konzeptuell im Hinblick auf die Fünf-Schichten-Architektur betrachtet wurde, wird in diesem Abschnitt näher auf einzelne Komponenten des DBMS eingegangen. Dabei werden diese Komponenten im Hinsicht auf ihre Bedeutung für ein sicheres Löschen im DBS untersucht. Des Weiteren wird darauf eingegangen, in wieweit es verhindert werden kann, einzelne Informationen, die zu einem späteren Zeitpunkt sicher gelöscht werden müssen, auf dem Sekundärspeichermedium zu speichern.

Indexstrukturen

Indexstrukturen werden in DBMS meist dazu verwendet, um schnell auf gewünschte Daten zuzugreifen. Dazu werden in vielen Indexstrukturen Informationen über die Daten gespeichert. Soll ein indexierter Datensatz sicher gelöscht werden, müssen auch seine indexierten Informationen sicher gelöscht werden. Geschieht eine Betrachtung nicht, kann der Datensatz nur partiell sicher gelöscht werden.

Wie schon in Abschnitt 2.4 beschrieben, können die Indexstrukturen in verschiedene Klasse eingeteilt werden. Je nach verwendeter Implementierung kann es sich bei einem Index um einen Primär oder Sekundärindex, geclustert oder nicht geclusterten Index, um einen dicht oder dünn besetzten Index und um einen Ein- oder Mehrkomponenten Index handeln [SHS05]. Dabei wird in den unterschiedlichen Klassen nicht immer die gleiche Menge an Informationen über die Daten gespeichert. Es werden zum Beispiel in dünn besetzten Indexen nicht die Werte aller existierender Ausprägungen des indexierten Attributes gespeichert. In ihnen werden nur die Werte weniger Attributausprägungen gespeichert. Jedoch muss die darunter liegende Tabelle nach dem Attribut geordnet vorliegen. Da eine Indexstruktur je nach Implementierung sowohl ein dicht oder dünn besetzter Index sein kann, muss eine datenbankmanagementsystemspezifische Betrachtung der Indexstrukturen durchgeführt werden.

Von Stahlberg et al. wurde gezeigt, dass das sichere Löschen der Informationen von Datensätzen aus B^+-Bäumen in InnoDB in effizienter Zeit durchgeführt werden kann. Dies ist der Tatsache geschuldet, dass die Indexstruktur zu den Zeitpunkt, an dem die in ihr gespeicherten Informationen angepasst werden, im Arbeitsspeicher gehalten wird [SML07]. Da es wahrscheinlich ist, dass die Indexstrukturen, wenn auf sie zugegriffen werden, immer im Arbeitsspeicher des Datenbankensystems vorliegen, kann ihre Lösung ohne Laufzeitverlust auch potentiell auf andere Indexstrukturen angepasst werden.
Jedoch wird auch gesagt, dass einige Indexstrukturen historisch abhängig sind. Das bedeutet, dass sich aus ihrer Struktur Rückschlüsse auf die Einfügereihenfolge der Datensätze ziehen lassen. Sollen solche Rückschlüsse verhindert werden, ist es notwendig, historisch unabhängige Indexstrukturen zu verwenden oder historisch abhängige in unabhängige zu verändern.

Pufferverwaltung

Durch Modifikationen an der Propagierungsstrategie des DBS kann verhindert werden, dass einzelne Bits von nicht rekonstruierbar gelöschten Datensätzen wieder rekonstruiert werden können. Um dies sicherzustellen, muss die Propagierungsstrategie dahingehend erweitert werden, dass die Möglichkeit besteht, eine Seite ohne große Performanceeinbußen mehrfach mit veränderten Werten schnell hintereinander zu propagieren. Bei den unterschiedlichen Seitenversionen, die dabei propagiert werden, müssen immer die Speicherbereiche, die von nicht mehr benötigten Daten verwendet wurden, mit anderen Daten überschrieben werden. Dabei ist es möglich, entweder das 33 fache Überschreiben mit zufälligen Werten zu verwenden, oder die von Gutmann vorgestellte Methode [Gut96], durch die Bereiche 35 mal überschrieben werden.

MVCC

Wird im DBMS das MVCC in der Transaktionsverwaltung verwendet, tritt eine Besonderheit auf. Diese hat Auswirkungen auf den Löschzeitpunkt der Datensätze. So kann ein Datensatz nicht sofort sicher gelöscht werden, wenn die Transaktion, durch die er gelöscht wurde, erfolgreich abgeschlossen ist. Es muss noch abgewartet werden, bis alle Transaktionen, mit denen vor dem Ende der Änderungstransaktion begonnen wurde, abgeschlossen sind. Der Grund dafür ist, dass bis zu diesem Zeitpunkt noch die Möglichkeit besteht, dass auf den Datensatz zugegriffen wird. Dies gilt jedoch nur unter der Bedingung, dass auch alle Benutzer auf den sicher zu löschenden Datensatz zugreifen dürfen. Jedoch kann zwischen dem Endzeitpunkt der Änderungstransaktion und dem Ende aller potentiell auf den Datensatz zugreifenden Transaktionen viel Zeit vergehen. In diesem Zusammenhang kann die Problematik bestehen, dass der Speicherbereich, auf dem der Datensatz gespeichert ist, zu dem Zeitpunkt, zu dem der Datensatz gelöscht werden darf, nicht mehr im Puffer des Systems gehalten wird. Diese Transaktionsverwaltung bedarf daher einer gesonderten Betrachtung, wenn sie in einem System verwendet wird, aus dem Datensätze sicher gelöscht werden sollen.

Protokollierung des Logbuchs

Bei der durch das Logbuch durchgeführten Protokollierung der Veränderungen innerhalb der DB kann zwischen zwei verschiedenen Arten der Protokollierung unterschieden werden. Dabei handelt es sich um die physische und die logische Protokollierung. Soll ein Datensatz verändert werden, wird in einem physisch geführten Logbuch ein *Before* Image des Speicherbereichs gespeichert, auf dem sich der Datensatz befindet. Als Größe für einen Speicherbereich wird in den meisten Fällen eine Seite verwendet. Bei der logischen Protokollierung werden hingegen höherwertige Operationen abgespeichert [SHS05]. Für die Wiederherstellung eines Datensatz nach einem Fehlerfall ist somit bei einer physischen Protokollierung nur das letzte *Before* Image des Speicherbereichs notwendig. Dem gegenüber müssen bei der logischen Protokollierung alle höherwertigen Operationen, die den Datensatz erstellt und verändert haben, neu ausgeführt werden.

Von Stahlberg et al. wurde beschrieben, dass ein sicheres Löschen von Datensätzen aus einem physisch protokollierten Logbuch durch eine Verschlüsselung des Logbuches vollzogen werden kann [SML07]. Dabei wird für jeden Speicherbereich eine eigene Schlüsselkette verwendet. Wird der Speicherbereich verändert, wird das *Before* Image des Speicherbereiches im Logbuch gespeichert. Dabei wird das Image unter Zuhilfenahme des nächsten Schlüssels der Schlüsselkette verschlüsselt und der vorher verwendete Schlüssel wird zerstört. Somit wird von der Tatsache Gebrauch gemacht, dass bei einer physischen Protokollierung nur das letzte *Before* Image benötigt wird, um einen Datensatz im Fehlerfall wieder zu rekonstruieren.

Da logisch protokollierte Logbücher anders aufgebaut sind, muss hier eine andere Lösung entwickelt werden. Dabei ist zu beachten, dass durch einen Eintrag in das logische Logbuch nicht nur Änderungen an einem Datensatz durchgeführt worden sein können. Es können stattdessen Änderungen an beliebig vielen Datensätzen durchgeführt worden sein. Somit kann nicht einfach beim sicheren Löschen eines Datensatzes jeder Logbucheintrag, durch dessen Operationen der zu löschende Datensatz verändert wurde,

mit gelöscht werden. Wird dies trotzdem gemacht, kann es in Fehlerfällen zu Problemen kommen, da existierende Datensätze nicht wieder konsistent hergestellt werden können. Eine Lösung, durch die ein sicheres Löschen eines Datensatzes durchgeführt werden könnte, müsste aus allen Logbucheinträgen die Daten des Datensatzes löschen, ohne die Rekonstruktionsmöglichkeiten der anderen Datensätze zu einträchtigen.

Spaltenorientierte Speicherung

Neben der zeilenorientierten Speicherung von Datensätzen existieren auch einige DBS, die ihre Datensätze spaltenorientiert speichern. Bei dem bereits erwähntem MonetDB handelt es sich um solch ein DBMS [Bon02]. Diese Speicherung bietet in einigen Anwendungsgebieten Vorteile, besonders im Bereich von Data Warehouse Prozessen. Auf diese Vorteile wird jedoch nicht näher eingegangen, da sie nicht im Fokus dieser Arbeit liegen. Ein Vergleich dieser beiden Speicherstrukturen im Hinblick auf ihre Anfrageverarbeitung ist in [AMH08] gegeben. Soll ein spaltenorientiert gespeicherter Datensatz sicher gelöscht werden, können die gleichen Methoden verwendet werden, die auch schon bei einem zeilenorientiert gespeicherten Datensatz verwendet werden.

Vorbereitendes Löschen

In vielen Anwendungsfällen ist es denkbar, dass schon bei der Erstellung der Datenbank Informationen darüber vorhanden sind, welche Art vom im System existenten Datensätzen wieder sicher gelöscht werden müssen. Aus diesem Grund kann versucht werden, nicht alle Informationen dieser Datensätze persistent zu speichern.
Dies gilt nicht für die Datensätze selbst, die weiterhin so gespeichert werden müssen, dass sie im Fehlerfall konsistent weiterverwendet werden können. Außerdem ist es weiterhin noch notwendig, das Logbuch der Datenbank persistent zu speichern. Es kann aber versucht werden, Informationen im Arbeitsspeicher des Systems zu halten, die bei einem Fehlerfall nicht benötigt werden. Bei solchen Informationen handelt es sich um datensatzabhängige Metadaten, also um Werte, die zum Beispiel in Indexen oder Statistiken gespeichert sind. Jedoch ist dabei zu beachten, dass dieses Änderungen zulasten des frei verfügbaren Arbeitsspeichers durchgeführt werden. Da weiterhin noch darauf geachtet werden muss, dass keine Auslagerung des Arbeitsspeichers durchgeführt wird, weil dadurch personenbezogenen Informationen persistent gespeichert werden, die nur schwer wieder auffindbar sind. In diesem Zusammenhang ist es sinnvoll, dem System Informationen zukommen zu lassen, in welchen Spalten personenbezogene Daten gespeichert sind. Somit könnte gezielt das Speichern von Statistiken oder Indexen über einzelne Spalten verhindert werden.

3.4 Die verwendete Strategie und ihre Grenzen

Von den in Abschnitt 3.2 vorgestellten Lösungen bieten nur die Lösungen, die an der mengenorientierten und an der satzorientierten Schnittstelle ansetzen, die Möglichkeit, einen Datensatz sicher aus einem DBS zu löschen. Sollen diese Lösungen auf ein reales DBMS angewendet werden, so müssen sehr viele Komponenten des DBMS angepasst werden. Dabei ist zum Beispiel die Erstellung eines Logbuchs erforderlich, aus dessen Informationen der konsistente Zustand einer Datenbank wiederhergestellt werden kann,

jedoch nicht bereits gelöschte Datensätze. Außerdem muss darüber hinaus sichergestellt werden, dass alle datensatzabhängigen Metadaten nach dem Löschen eines Datensatzes dahingehend verändert werden, dass der Datensatz aus ihnen nicht rekonstruiert werden kann. Da dies jedoch den Rahmen dieser Arbeit sprengen würde, wird hier von einer derartigen Lösung abgesehen.

Bei jeder der oben beschriebenen Methoden ist es notwendig, die Werte des Datensatzes innerhalb der DB sicher zu löschen, nachdem der Datensatz als nicht mehr benötigt markiert wird. Dieses Löschen der Datensätze ist somit die Grundlage aller beschriebenen Strategien. Aus diesem Grund wird im Laufe dieser Arbeit das sichere Löschen der Datensätze aus der DB näher betrachtet.

Grenzen der Strategie

Bei der verwendeten Strategie werden nur die Datensätze selbst sicher gelöscht, deshalb ist sie als allumfassende Lösung für das forensisch sichere Löschen von Datensätzen aus DBMS nicht geeignet. Es ist weiterhin noch möglich, die Datensätze partiell unter Zuhilfenahme von datensatzabhängigen Metadaten, die in Datenbankstatistiken oder Indexstrukturen gespeichert sind, zu rekonstruieren. Des Weiteren besteht die Möglichkeit, die Daten durch die in Logbüchern gespeicherten Informationen wieder herzustellen. Jedoch ist eine solche Rekonstruktion der Daten aufwendiger als das einfache Auslesen der Dateien, in denen die Daten gespeichert sind. Eine Rekonstruktion der Daten wird somit durch die verwendete Strategie erschwert, aber nicht verhindert.

3.5 Zusammenfassung

In diesem Kapitel wurden theoretische Betrachtungen dahingehend durchgeführt, in wieweit bestehende Datenbankmanagementsysteme angepasst werden müssen, wenn Datensätze aus ihnen sicher gelöscht werden sollen. Dazu wurden zu Beginn dieses Kapitels einige Begriffe definiert. Es folge eine Beschreibung der allgemeinen Ansatzpunkt, an denen mit den Veränderungen der Funktionalität begonnen werden kann. Abschließend wurde dann konzeptuell auf einzelne Bestandteile von DBMS eingegangen.

4. Speicherung der Datensätze in ausgewählten DBMS

In den vorhergehenden Kapiteln (Kapitel 2 und Kapitel 3) wurden generelle Grundlagen aufgeführt, um zu verstehen, welche Änderungen an einem DBMS durchgeführt werden müssen, um Datensätze sicher aus ihnen löschen zu können. In diesem Kapitel wird auf konkrete DBMS Implementierungen eingegangen. Dabei werden die Dateien näher betrachtet, in denen die DBMS ihre Datensätze speichern. Des Weiteren wird gezeigt, in wieweit Werte von eigentlich gelöschten Datensätzen in bestehenden DBS noch in Dateien innerhalb von DB-Slacks vorhanden sind und in wieweit somit auch ohne Zuhilfenahme von Metadaten oder des Logbuchs die Möglichkeit besteht, diese Datensätze wiederherzustellen. Da nicht alle existierenden DBMS näher betrachtet werden können, gibt diese Arbeit einen Einblick in zwei Open Source DBMS. Um zu zeigen, dass die in den DBMS verwendeten Techniken typisch für DBMS sind, wird am Ende dieses Kapitels erläutert, wie Datensätze typischerweise gelöscht werden.

Bei den beiden in dieser Arbeit näher betrachteten DBMS handelt es sich um PostgreSQL und HyperSQL DataBase (HSQLDB). Da es sich hier um Open Source DBMS handelt, ist eine Erweiterung ihrer Funktionalität im Hinblick auf das sichere Löschen von Datensätzen möglich. Neben diesen beiden DBMS gibt es noch eine große Anzahl weiterer Open Source Datenbankmanagementsysteme, wie zum Beispiel MySQL[1] oder Apache Derby[2], auf die in dieser Arbeit nicht weiter eingegangen wird. Da in dieser Arbeit nicht alle Open Source Datenbankmanagementsysteme näher betrachtet werden können, werden mit PostgreSQL ein C basiertes DBMS und mit HyperSQL ein Java basiertes DBMS analysiert. Mit dieser Auswahl wird untersucht, ob programmiersprachenspezifische Besonderheiten bei der Umsetzung des sicheren Löschens zu Problemen führen.

[1] http://www.mysql.de/, letzter Zugriff 19.12.2011
[2] http://db.apache.org/derby/, letzter Zugriff 19.12.2011

4.1 PostgreSQL

PostgreSQL ist ein objektrelationales Datenbankmanagementsystem, das auf den Entwicklungen des in den Jahren 1975-1977 entstandenen RDBMS Ingres basiert. Ingres war neben R eines der ersten bekannteren RDBMS [Bon02]. In den ersten Jahren wurde es als universitäres Projekt in Berkley entwickelt. Im Jahre 1986 wurde das Ingres Projekt unter der Leitung von Michael Stonebraker unter dem Namen Postgres als Open Source Projekt weiterentwickelt und 1996 in PostgreSQL umbenannt [WD02, LAHG05, SR86].

Aktuell wird PostgreSQL unter der PostgreSQL License[3] weiterentwickelt. Es wird nicht nur in konventionellen Datenbankanwendungen verwendet, sondern auch als Datenbankschicht von HadoopDB, einem parallelen DBMS mit shared-nothing-Architektur, das die Lücke zwischen SQL und MapReduce Datenbanken schließen soll [ABPA+09].

Zur strukturierten Analyse von PostgreSQL wird zu zunächst auf die Architektur dieses DBMS eingegangen.

4.1.1 Architektur

Innerhalb der Architektur von PostgreSQL wird in dieser Arbeit zuerst auf den logischen Aufbau eines PostgreSQL DBS eingegangen. Nachdem dies geschehen ist, wird dargelegt, wie Datensätze innerhalb einer PostgreSQL DB gespeichert werden.

Logischer Aufbau

Die drei logischen Hauptkomponenten von PostgreSQL sind der **postmaster**, das **backend** und das **frontend** oder auch der Client. Dabei stammen der postmaster und das backend aus den gleichen ausführbaren Dateien. Wird ein PostgreSQL DBS gestartet, so startet sich ein postmaster Prozess [LAHG05]. Innerhalb dieses postmaster Prozesses arbeitet PostgreSQL hierarchisch. In Abbildung 4.1 ist dieser Aufbau grafisch dargestellt. An oberster Stelle steht hierbei ein Datenbank Cluster, der in einem eigenen Verzeichnisbaum arbeitet. Dieser Cluster besteht aus einer Menge von Datenbanken, wobei jede Datenbank durch ihren Namen im Cluster eindeutig identifizierbar sein muss. Dieser Name muss jedoch nicht eindeutig über mehrere Cluster hinweg sein. Jede Datenbank ist wieder selbst eine Menge von Tabellen, Funktionen, Operationen, Sichten, Indexen und so weiter, die wieder eindeutig über ihren Namen identifizierbar sein müssen [MRD10].

Versucht sich ein Client auf der Datenbank eines Clusters anzumelden, wird vom postmaster Prozess gegebenenfalls die Gültigkeit der Anmeldung überprüft. Sind der Benutzername und das Passwort gültig, startet der postmaster Prozess für den Client einen eigenen backend Prozess. PostgreSQL verfügt somit über ein ein-Prozess-pro-Benutzer Modell [SR86]. In Abbildung 4.2 ist dieser Aufbau grafisch dargestellt. Durch den backend Prozess werden alle Operationen entgegengenommen, die vom Client auf der Datenbank durchgeführt werden sollen. Dies geschieht solange wie die Verbindung zwischen dem Client und dem Server besteht [LAHG05].

[3] http://www.postgresql.org/about/licence/, letzter Zugriff 19.12.2011

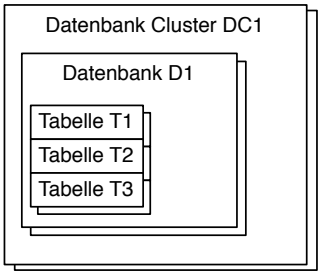

Abbildung 4.1: Hierarchischer Aufbau innerhalb eines postmaster Prozesses.

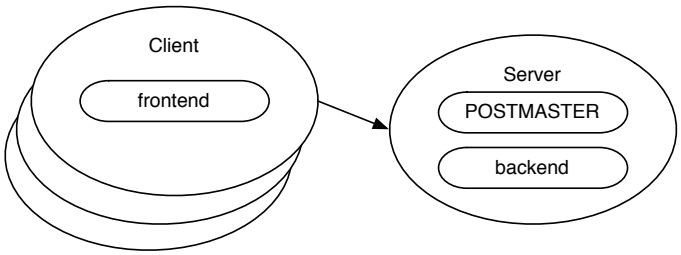

Abbildung 4.2: PostgreSQL Prozessaufbau nach [Loc98].

Physische Speicherung

In PostgreSQL werden zur Identifizierung alle Objekte und Operationen vier Arten von Identifikatoren verwendet. Dabei handelt es sich um die **Objekt ID (OID)**, die **XID**, die **CID** und die **TID**. Bei der OID handelt es sich um den Objektidentifikator eines Objektes. Da es sich bei PostgreSQL um ein objektrelationales Datenbankmanagementsystem handelt, wird jedes Objekt innerhalb eines Datenbank Cluster mit einer derartigen ID versehen. Dabei kann es sich um Tabellen aber auch um Datensätze oder Indexe handeln. Zur Identifizierung der Transaktionen wird jeder Transaktion eine fortlaufende Nummer gegeben, die als XID bezeichnet wird. Bei der CID handelt es sich um einen Kommandoidentifikator und bei der `TID` um einen Tupel beziehungsweise Zeilenidentifikator[4].

Wird ein Datenbank Cluster auf einem System erstellt, wird für diesen Cluster ein eigener Ordner angelegt. Innerhalb dieses Ordners werden in verschiedenen Unterordnern unterschiedliche clusterspezifische Informationen gespeichert. Dabei handelt es sich um allgemeingültige Informationen, wie zum Beispiel die Optionen, mit denen der Cluster gestartet wurde, sowie um datenbankspezifische Informationen. Ein Überblick über die verschiedenen Unterordner und die in ihnen gespeicherten Informationen wird in [Gro11] gegeben.

Bei der Erstellung einer Datenbank innerhalb des Clusters wird innerhalb des „base“ Ordners des Clusters ein Unterordner angelegt. Als Name dieses Ordners wird dabei die OID der Datenbank verwendet. Für jede Tabelle und für jeden Index dieser Da-

[4] http://www.postgresql.org/docs/9.1/static/datatype-oid.html, letzter Zugriff 18.01.2012

tenbank werden dann innerhalb dieses Ordners eigene Dateien angelegt. Der Name dieser Dateien wird dabei durch den *filenode* der Tabelle beziehungsweise des Indexes festgelegt. Dieser *filenode* kann über die clusterweite Tabelle `pg_class.relfilenode` ermittelt werden. An dieser Stelle sei angemerkt, dass der *filenode* nicht mit dem OID der Tabelle übereinstimmen muss. Dies ist zwar im Normalfall so, jedoch nicht, wenn Operationen wie TRUNCATE, REINDEX, CLUSTER auf der Tabelle beziehungsweise dem Index durchgeführt wurden [Gro11, MRD10].

In den meisten Fällen werden für jede Tabelle und für jeden Index drei verschiedene Arten von Dateien angelegt. Eine Ausnahme bilden dabei Tabellen, deren Veränderungen im Logbuch nicht protokolliert werden und Indexe, die über diesen Tabellen angelegt sind. Für diese nicht absturzsicheren Tabellen beziehungsweise Indexe wird eine weitere Datei angelegt. Bei den drei üblicherweise angelegten Dateien handelt es sich um eine Datei, in der die Tabelleninhalte selbst gespeichert werden und zwei Dateien, in denen Informationen über die erste Datei gespeichert sind. In der ersten dieser beiden Dateien werden Informationen über freien Raum innerhalb der ersten Datei gespeichert. Sie wird in PostgreSQL *Free Space Map* genannt. Zur Identifikation endet der Dateiname dieser Datei mit der Endung `_fsm`. Die andere Datei, beinhaltet Informationen, welche Seiten der Datei keine toten Datensätze beinhalten. Zur Identifizierung wird sie als *visibility map* bezeichnet und endet mit dem Suffix `_vm`. Da in dieser Arbeit nicht genauer auf diese Dateien eingegangen wird, können in der Dokumentation[5] weitere Informationen zu den `_fsm` Dateien nachgeschlagen werden. Zusätzliche Informationen zu den `_vm` Dateien sind innerhalb der PostgreSQL Dokumentation[6] zu finden. Der Grund dafür, dass hier keine tiefer gehende Betrachtung dieser Dateien durchgeführt wird, liegt darin, dass in den Dateien keine Datensatzinhalte gespeichert werden. Damit ist eine Betrachtung dieser Dateien in Hinblick auf das sichere Löschen von Datensätzen nicht notwendig.

Wird eine der oben beschriebenen Dateien größer als 1 GB, wird die Datei umbenannt und eine neue Datei wird angelegt. Dabei werden die Dateien mit dem Namen „filenode.1“, „filenode.2“ und so weiter durchnummeriert [MRD10]. Durch diesen Mechanismus kann PostgreSQL auch auf Dateisystemen arbeiten, die eine maximale Größe für Dateien besitzen [Gro11].

4.1.2 Seitenaufbau

Sollen die Datensätze nicht rekonstruierbar aus dem DBMS gelöscht werden, ist es wichtig, den genauen Aufbau der Dateien zu verstehen, in denen die Datensätze gespeichert werden, damit nicht Bestandteile des Datensatzes weiter auf der Festplatte verbleiben beziehungsweise für das DBMS wichtige Informationen mit gelöscht werden. Dazu wird in diesem Abschnitt genau auf den Aufbau dieser Dateien eingegangen.

Generell werden die Dateien, in denen die Datensätze gespeichert werden, zur internen Verwaltung in 8Kb große Seiten unterteilt [Gro11]. Dabei kann beim Kompilieren der Server Quelldateien auch eine andere Größe angegeben werden. In Tabelle 4.1 ist der generelle Aufbau dieser Seiten beschrieben und in Abbildung 4.3 grafisch gezeigt. Die

[5] http://www.postgresql.org/docs/9.1/static/storage-fsm.html, letzter Zugriff 02.11.2011
[6] http://www.postgresql.org/docs/9.1/static/storage-vm.html, letzter Zugriff 02.11.2011

Komponente	Beschreibung
PageHeaderData	24 byte Länge. Beinhaltet generelle Informationen über die Seite und einen Zeiger zum Start des FreeSpace Bereiches.
ItemIDData	Bereich von Zeigerparren (Offset, Länge) zu den auf der Seite gespeicherten Datensätzen. Dabei hat jedes Zeigerpaar eine Länge von 4 Byte.
Free Space	Bereich für unallocierten Speicher. Wird ein neuer Datensatz auf der Seite gespeichert, wird sein Zeigerpaar an den Anfang des Bereiches gespeichert und der Datensatz selbst an das Ende dieses Bereiches.
Items	Bereich, in dem die Werte der Datensätze gespeichert werden. Zusätzlich zu den Werten des Datensatzes werden in diesem Bereich die *HeadTupleHeaderData* Informationen des Datensatzes gespeichert.
Special Space	Bereich in dem Spezifische Daten für Zugriffsmethoden gespeichert werden. Verschiedene Methoden speichern dabei unterschiedliche Daten. Bei normalen Tabellen ist dieser Bereich leer.

Tabelle 4.1: Speicherstruktur einer Seite in PostgreSQL.

Seiten sind in 5 Bereiche unterteilt. Die ersten 24 Byte einer Seite werden *PageHeaderData* genannt. Sie werden verwendet, um generelle Informationen über die Seite zu speichern. Dabei handelt es sich um Informationen wie zum Beispiel den Offset zum Start des freien Bereichs, den Offset zum Ende des freien Bereichs und Informationen über den Zeitpunkt der letzten Änderung, die auf der Seite durchgeführt wurde. Der freie Bereich wird im nachfolgenden *FreeSpace* genannt. Eine genaue Beschreibung der Daten des *PageHeaderData* Bereiches ist in der PostgreSQL Dokumentation gegeben[7].

Der zweite Bereich der Seite wird zur Verwaltung der Tupel verwendet. Er wird im Weiteren nach seiner internen Benennung in PostgreSQL *ItemIDData* genannt. In diesem Bereich wird für jedes Tupel, das auf der Seite gespeichert wird, ein Zeigerpaar angelegt. In diesem Zeigerpaar sind der Offset zwischen dem Anfang der Seite und dem Anfang des Datensatzes gespeichert sowie Informationen über die Länge des Datensatzes. Nachfolgend befindet sich der *FreeSpace* Bereich. Soll ein neuer Datensatz auf der Seite gespeichert werden, wird das Zeigerpaar des Datensatzes an den Anfang des *FreeSpace* Bereiches gespeichert und der Datensatz selbst an das Ende. Reicht der freie Bereich nicht aus, um diese Informationen zu speichern, wird eine neue Seite angelegt. Auf dieser neu angelegten Seite wird dann der Datensatz und das zu ihm gehörende Zeigerpaar eingefügt. Dabei wird darauf geachtet, dass Informationen zu einem Datensatz nur auf einer Seite gespeichert werden. Nach dem freien Bereich sind dann die Daten der Datensätze gespeichert, wobei zusätzlich zu den Daten auch Metadaten zu diesen

[7] http://www.postgresql.org/docs/9.1/static/storage-page-layout.html#PAGEHEADERDATA-TABLE, letzter Zugriff 21.12.2011

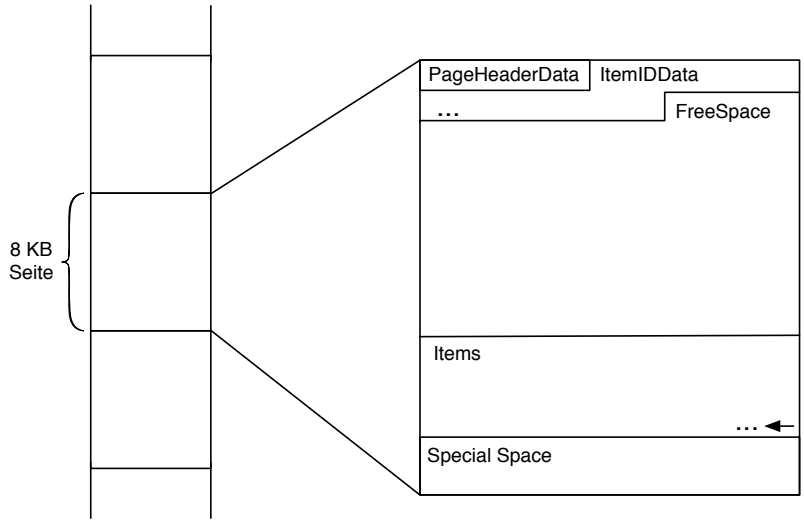

Abbildung 4.3: Grafische Darstellung der Struktur einer Seite in PostgreSQL.

Daten gespeichert werden. Die Metadaten werden im Nachfolgenden in Anlehnung an ihre interne Benennung in PostgreSQL *HeadTupleHeaderData* genannt. Die letzte Region der Seite wird benötigt, wenn Daten in der Tabelle gespeichert sind, die spezielle Zugriffsmethoden benötigen. Werden solche Zugriffsmethoden nicht benötigt, ist dieser Bereich nicht vorhanden beziehungsweise seine Länge ist Null.

Wie schon oben erwähnt, werden im Bereich, in dem die Datensätze gespeichert werden, zusätzlich zu den Daten des Datensatzes selbst noch die *HeadTupleHeaderData* Informationen des Datensatzes gespeichert. Diese Metadaten besitzen eine Länge von mindestens 23 Byte. Um welche Informationen es sich dabei handelt ist in Tabelle 4.2 beschrieben. In Tabelle Tabelle 4.2 wird ebenfalls gezeigt, welche Speicherlänge für die einzelnen Bestandteile dieser Daten benötigt wird. Bei diesen Daten ist jedoch zu beachten, dass in den Informationen entweder `t_cid` oder `t_xvac` gespeichert werden. Diese Informationen können auch im PostgreSQL Quelltext in der Datei SRC/INCLUDE/ACCESS/HTUP.H nachgelesen werden.

Löschen von Datensätzen

Eines der Ziele bei der Entwicklung der PostgreSQL Speicherverwaltung war es, eine Speicherverwaltung zu erschaffen, in der alle historischen Werte von Datensätzen erhalten bleiben [Sto87]. Dazu wird bei jeder `UPDATE` Operation ein neuer Datensatz in den Dateien angelegt und der alte Datensatz wird infolgedessen als nicht mehr aktuell markiert. Wird ein Datensatz gelöscht, so wird er ebenfalls als nicht mehr aktuell markiert [Mom01]. Damit nicht unnötig viel Speicherplatz für nicht mehr benötigte Daten verwendet wird, existiert in PostgreSQL der in Abschnitt 2.4 beschriebene `VACUUM` Prozess [Sto87]. Dieser Prozess kann vom Anwender gesteuert werden. Dabei kann der Prozess mit verschiedenen Einstellungen ausgeführt werden. So kann zum Beispiel festgelegt werden, dass nur einzelne Tabellen betrachtet werden. Wie schon in Abschnitt 2.4

Feld	Länge	Beschreibung
t_xmin	4 Byte	XID Nr. der Transaktion, in der der Datensatz auf die Seite eingefügt wurde.
t_xmax	4 Byte	XID Nr. der Transaktion, in der der Datensatz gelöscht wurde. Dabei wird sowohl beim `UPDATE` als auch beim `DELETE` des Datensatzes dieses Attribut gesetzt.
t_cid	4 Byte	CID der Operation, durch die das Tupel zuletzt verändert wurde. Als Speicherbereich wird für diese Attribut der gleiche Bereich wie für t_xvac verwendet.
t_xvac	4 Byte	XID des Datensatzes, damit VACUUM operation moving a row version
t_ctid	6 Byte	Aktuelle TID von dieser oder einer neuen Zeilen Version des Datensatzes.
t_infomask2	2 Byte	Anzahl der Attribute des Datensatzes und Bits, in denen Flags gesetzt werden.
t_infomask	2 Byte	Diverse Bit Flags.
t_hoff	1 Byte	Offset zu den Benutzerdaten des Datensatzes.

Tabelle 4.2: Aufbau der HeapTupleHeaderData Informationen.

beschrieben, wird durch diesen Prozess jedoch nicht sicher gestellt, dass nicht mehr benötigte Datensätze nicht rekonstruierbar gelöscht sind. Eine Rekonstruktion gelöschter Datensätze ist somit auch noch auf Dateiebene möglich.

4.2 HSQLDB

Bei dem zweiten DBMS, das in dieser Arbeit näher betrachtet wird, handelt es sich um HSQLDB. HSQLDB ist die führende in Java geschriebene SQL Relationale Datenbank Umgebung, sie wird unter anderem als DB für OpenOffice 3.0 verwendet[8].

Die Datensätze einer HSQLDB DB können in drei verschiedenen Arten von Tabellen speichert werden. Dabei handelt es sich um `TEXT`, `MEMORY` und `CACHED` Tabellen. `TEXT` Tabellen speichern ihre Informationen in .CSV Dateien, weshalb sie praktisch für Applikationen sind, deren Informationen in einem austauschbaren Format vorliegen sollen. Jedoch sollten sie nicht verwendet werden, wenn kein Austausch der Datensätze durchgeführt werden soll. Es wird stattdessen empfohlen, `MEMORY` und `CACHED` Tabellen zu verwenden[9].

Sowohl `MEMORY`, als auch `CACHED` Tabellen besitzen ihre Vor- und Nachteile und bedienen unterschiedliche Anwendungsgebiete. So wird der Inhalt von `MEMORY` Tabellen in

[8] http://www.hsqldb.org/, letzter Zugriff 23.11.2011
[9] http://www.hsqldb.org/, letzter Zugriff 23.11.2011

einer .script Datei gespeichert, die beim Schließen der DB persistent geschrieben und beim Starten einer HSQLDB Instanz eingelesen wird. Während das DBS läuft, werden die aktuellen Werte der DB im Arbeitsspeicher gehalten. Dadurch ist ein sehr schneller Zugriff auf die Inhalte dieser Tabellen möglich. Jedoch können hier keine großen Datenmengen von mehreren Gigabyte gespeichert werden. Für die Speicherung großer Datenmengen können `CACHED` Tabellen verwendet werden, da immer nur Teile der Informationen einer `CACHED` Tabelle im RAM gehalten werden [Gro12]. Dabei werden die Informationen aller `CACHED` Tabellen in einer Datei gespeichert. Zur internen Verwaltung dieser Datei verwendet HSQLDB die Klasse `java.io.RandomAccessFile`, wobei jeder Datensatz der DB innerhalb eines Objektes gespeichert wird, das dann in die Datei serialisiert wird.

Löschen von Datensätzen

Soll das Löschen der Datensätze in HSQLDB betrachtet werden, muss unterschieden werden, ob der zu löschende Datensatz in einer `CACHED` oder in einer `MEMORY` Tabelle gespeichert ist. Wurde der Datensatz in einer `MEMORY` Tabelle gespeichert, ist er nach dem Beenden der Datenbank Instanz auf Dateiebene gelöscht. Der Grund dafür ist, dass beim Beenden der Datenbank alle Datensätze der Datenbank neu in eine Datei geschrieben werden. Jedoch kann es passieren, dass die Werte von gelöschten und auch von nicht gelöschten Datensätzen innerhalb von DS-Slacks weiter existieren. Dieses Verhalten kann dann auftauchen, wenn die beim Beenden der DB erstellte .script Datei an andere Speicherbereiche geschrieben wird als die schon existente .script Datei oder auch wenn die neu erstellte Datei weniger Speicherplatz verbraucht als die vorherige.
Ein anderes Verhalten tritt auf, wenn die Datensätze in `CACHED` Tabellen gespeichert sind. Wie in PostgreSQL werden auch die Daten dieser Tabellen beim Löschen aus der Datenbank nicht auf Dateiebene gelöscht. Es wird intern ein Verweis angelegt, dass der Bereich, auf dem der Datensatz geschrieben war, von einem neuen Datensatz genutzt werden kann. Eine Möglichkeit diese DB-Slacks zu eliminieren kann beim Schließen der Datenbank aufgerufen werden. So kann dabei neben der Anweisung `SHUTDOWN` die Anweisung `SHUTDOWN COMPACT` verwendet werden. Beim Ausführen dieser Anweisung werden alle von der Datenbank verwendeten Datensätze in einer neu erstellten Datenbankdatei gespeichert. Danach wird die alte Datenbankdatei gelöscht und die neue Datei wird umbenannt. Durch diese Prozedur werden zwar alle DB-Slacks vernichtet, es entsteht jedoch für jeden Datensatz ein DS-Slack. Die Problematik des forensisch sicheren Löschens ist also durch diese Anweisung von dem DBS auf das Dateisystem verschoben worden.

4.3 Löschen von Datensätzen in anderen DBMS

In den bisher betrachteten DBMS sind die Datensätze, nachdem sie aus den DBMS gelöscht wurden, in den Datensatzdateien noch rekonstruierbar enthalten. Dieses Problem taucht auch in anderen DBMS auf. In diesem Abschnitt wird gezeigt, wie das Löschen von Datensätzen in einigen bekannten DBMS durchgeführt wird. Dabei wird dargelegt, dass es auch in diesen DBMS möglich ist, gelöschte Datensätze bereits auf Dateiebene wieder rekonstruieren zu können.

Oracle Database

Oracle Database ist eines der meist genutzten DBMS. Mit ihm können relationale und objektrelationale Daten gespeichert werden. Der Aufbau der Dateien, in denen Oracle 10g Release 2 seine Daten speichert, ist dem Aufbau der Dateien von PostgreSQL sehr ähnlich. Jede Datei ist in Blöcke aufgeteilt, wobei in unterschiedlichen Blöcken auch verschiedenartige Objekte wie zum Beispiel Datensätze, Indexe und Cluster-Daten gespeichert werden können. Jeder dieser Blöcke besitzt einen eigenen Header, in dem allgemeine Informationen über den Block gespeichert werden. Gefolgt wird dieser Block von einem Tabellen Directory und einem Datensatz Directory. Nach dem Datensatz Directory folgt ein freier Bereich. Wird ein Datensatz in dem Block gespeichert, wird ein Identifikator des Datensatzes an das Datensatz Directory angehangen. Der Datensatz selbst wird vom Ende des freien Bereichs gespeichert. Zusätzlich zu den Werten des Datensatzes werden dabei allgemeine Informationen über den Datensatz gespeichert. Dieser besteht aus drei Byte, wobei das erste Byte Informationen über den Zustand des Datensatzes enthält. Wird ein Datensatz gelöscht, wird das fünfte Bit dieses Bytes gesetzt, während die Werte des Datensatzes weiterhin in dem Block enthalten bleiben. Die Werte der Datensätze sind somit weiter in den Dateien enthalten und es kann mittels einer einfachen Analyse der Dateien herausgefunden werden, welche Datensätze gelöscht wurden [Lit07b].

SQL Server

Auch die Dateien, in denen von SQL Server Datensätze gespeichert werden, sind in Seiten organisiert. Wird ein Datensatz von einer dieser Seiten gelöscht, so wird er, wie in den bisher betrachteten DBMS nicht physisch gelöscht, er wird dereferenziert, so dass er bei SQL Anfragen nicht berücksichtigt wird. Die einzige Ausnahme bilden Daten, die in Tabellen existieren, die einen geclusterten Index besitzen. Diese Daten werden sofort überschrieben. Der Grund dafür ist, dass diese Indexe beschreiben, wie SQL Server intern die Daten auf einer Datenseite ordnet. Daher ist es oft der Fall, dass die Daten auf diesen Seiten überschrieben werden. Mit der Ausnahme von Daten aus Tabellen mit geclusterten Indexen ist es somit möglich, gelöschte Daten wieder zu rekonstruieren. Auf physischer Ebene wird beim Löschen eines Datensatzes die Beschreibung seiner Länge auf 0 gesetzt. Da im Normalfall kein Datensatz die Länge 0 hat, sind alle Datensätze mit dieser Länge gelöschte Datensätze. Somit ist es möglich, gelöschte Datensätze einfach wieder herzustellen [Fow08].

MySQL

Bei dem nächsten bekannten und hier erwähnten DBMS handelt es sich um MySQL Database 5.1.32. MySQL unterstützt mehrere Speicher-Engines, von denen eine InnoDB ist. Sie ist transaktionssicher, beinhaltet die Möglichkeit eines Rollbacks und Möglichkeiten, Daten nach einem Absturz des Systems wiederherzustellen[10]. Daher besitzt sie viele Funktionen, die nötig sind, um wichtige Daten verlustsicher zu speichern und zu verändern.
Die Dateien, in denen die Datensätze einer mit InnoDB verwalteten Datenbank gespeichert werden, sind in sieben Bereiche unterteilt. In einem dieser Bereiche werden alle

[10] http://dev.mysql.com/doc/refman/5.1/en/storage-engines.html, letzter Zugriff 8.12.2011

Daten der Benutzer gespeichert. Zusätzlich zu den Daten der Datensätze werden in diesem Bereich noch Metainformationen gespeichert. Eine dieser Informationen ist ein Bit, in dem gespeichert ist, ob ein Datensatz gelöscht ist. Wird ein Datensatz durch eine Operation gelöscht, so wird nur der Wert dieses Bits verändert. Der Datensatz selbst bleibt erhalten, bis er durch einen neuen Datensatz überschrieben ist. Besitzt dieser neue Datensatz NULL Werte, bleiben die Werte des alten Datensatzes weiter erhalten, da InnoDB die NULL Werte nicht in der Datei speichert. Es werden lediglich Informationen über die Länge des Datensatzes in den Metadaten des Datensatzes aktualisiert [FHMW10]. Es kann also vorkommen, dass die Werte gelöschter Datensätze innerhalb von Slacks noch über lange Zeit hinweg in den Dateien erhalten bleiben.

4.4 Fazit

In diesem Kapitel wurde gezeigt, dass in vielen DBMS nach dem Löschen eines Datensatzes die Werte des Datensatzes noch auf Dateiebene erhalten bleiben. Obwohl dieses Auslesen der Werte der Datensätze nur unter Zuhilfenahme von Hexeditoren funktioniert, können somit die Werte gelöschter Datensätze einfach ermittelt werden. Dies ist dadurch begünstigt, das Informationen über die Struktur und Beschaffenheit der Daten oft in frei verfügbarer Literatur wie zum Beispiel in [FHMW10, Fow08, Lit07b, Gro11] gegeben werden.

Zur Lösung dieser Problematik, dass gelöschte Datensatzwerte bereits auf Dateiebene wiederherstellt werden können, wurde von Stahlberg et al. eine Methode vorgestellt, bei der die Speicherbereiche der nicht mehr benötigten Daten mit zufällig generierten Daten überschrieben werden. Jedoch wird eine Lösung nur exemplarisch für InnoDB gezeigt. Es muss also untersucht werden, ob die Lösung auch auf andere DBMS angewendet werden kann oder ob eine Adaption ihrer Lösung zu signifikanten Laufzeiteinbußen führt.

5. Analyse und Implementierung

Im vorhergehenden Kapitel wurde beschrieben, wie die Dateien in den näher betrachteten DBMS aufgebaut sind. Außerdem wurde erläutert, welche Auswirkungen ein Löschen der Datensätze auf die Repräsentation der Daten in den Dateien hat. In diesem Kapitel wird auf die Implementierung eingegangen, durch die verhindert werden kann, dass beim Löschen eines Datensatzes Werte des Datensatzes noch in DB-Slacks weiter existieren. Die generelle Idee dabei ist es, die Datensätze mit zufällig generierten Daten zu überschrieben.

Damit sich der Vorgang des Überschreibens in den Ablauf des DBMS einfügt, wird in Abschnitt 5.1.1 analysiert, wie die Strategie in PostgreSQL umgesetzt werden kann. In Abschnitt 5.1.2 folgt dann die genauere Betrachtung der Implementierung. Damit auch sicher gegangen werden kann, dass bei dem Überschreiben der Werte in PostgreSQL keine DB-Slacks entstehen, wird in Abschnitt 5.1.3 ein Programm vorgestellt, das die Dateien analysiert, in denen die Datensätze gespeichert werden. Nachfolgend wird in Abschnitt 5.2.1 analysiert, wie ein sicheres Löschen von Datensätzen aus Datenbanken in HSQLDB umgesetzt werden kann. Abschließend wird in diesem Kapitel noch in Abschnitt 5.2.2 auf die Umsetzung der Implementierung in HSQLDB eingegangen.

5.1 PostgreSQL

Im Verlauf dieser Arbeit wurde die PostgreSQL Version 9.1 analysiert und erweitert. Sie befindet sich zum gegenwärtigen Zeitpunkt (Dezember 2011) noch in der Entwicklungsphase und kann unter der Adresse [1] erreicht werden. Damit beim Überschreiben der nicht mehr benötigten Datensätze nicht unnötig viele zusätzliche Operation durchgeführt werden, muss analysiert werden, an welchen Stellen im Programmablauf alte nicht mehr benötigte Datensätze als veraltet markiert werden.

[1] http://www.postgresql.org/ftp/source/, letzter Zugriff 20.11.2011

5.1.1 Analyse

Bei der Analyse des Programmablaufes haben sich zwei Zeitpunkte herausgestellt, an denen das Überschreiben der Werte des Datensatzes potentiell durchgeführt werden kann, ohne zusätzliche Operationen, wie ein Holen der Seite, durchführen zu müssen. Bei dem Ersten dieser beiden Zeitpunkte handelt es sich um den Moment, in dem der alte Datensatz persistent als veraltet markiert wird. Dies geschieht dann, wenn die Operation der Änderungstransaktion ausgeführt wird, durch die der Datensatz verändert wird. Generell wird ein Datensatz durch ein Setzen seines `t_xmax` Attributes als veraltet markiert. Wird zu diesem Zeitpunkt der Datensatz überschrieben, kann es zu Fehlern im Mehrbenutzerbetrieb kommen. Dies rührt von der Tatsache her, dass in PostgreSQL das MVCC verwendet wird. Es kann also vorkommen, dass der als veraltet markierte Datensatz noch von anderen Transaktionen verwendet werden kann, die lesend auf ihn zugreifen wollen. Dabei kann so ein Zugriff von allen Transaktionen durchgeführt werden, mit denen vor dem Beenden der Änderungstransaktion begonnen wurde. Außerdem kann es zu Problemen kommen, wenn die Änderungstransaktion abgebrochen wird. Da zu dem Zeitpunkt, zu dem die Transaktion abgebrochen wird, die unveränderte Version des Datensatzes bereits mit zufällig generierten Daten überschrieben wurde, können die alten Werte des Datensatzes nur noch unter Zuhilfenahme des Logbuches wieder rekonstruiert werden. Jedoch besitzt dieses Verfahren einen Vorteil. Es kann mit Sicherheit gesagt werden, dass die ursprünglichen Werte des Datensatzes nach dem Abschluss der Transaktion, durch die der Datensatz verändert wurde, nicht mehr in der DB enthalten sind.

Der andere Punkt, an dem das Überschreiben des Datensatzes durchgeführt werden kann, ist derjenige, zu dem der Datensatz nicht mehr von laufenden Transaktionen verwendet wird. Zu diesem Zeitpunkt wird innerhalb der *HeadTupleHeaderData* des Datensatzes markiert, dass der Datensatz beim nächsten Ausführen des `VACUUM` Prozesses überschrieben werden kann. Wird das Überschreiben zu diesem Zeitpunkt durchgeführt, können keine Seiteneffekte im Mehrbenutzerbetrieb auftreten, da alle Transaktionen, die diese Kopie des Datensatzes verwenden, bereits abgeschlossen wurden. Außerdem können auch beim Abbruch einer Transaktion die Werte von bereits veränderten Datensätzen wieder rekonstruiert werden. Jedoch kann es passieren, dass zwischen dem Ersetzen eines Datensatzes und dem Beenden aller Transaktionen, die auf die alte Datensatzkopie zugreifen, viel Zeit verstreicht.

Eigenschaft	**Methode 1**	**Methode 2**
Transaktionssicher	-	✓
Tauglich im Mehrbenutzerbetrieb	-	✓
Zeitpunkt des Löschens	unmittelbar	verzögert

Tabelle 5.1: Eigenschaften der Methoden des Löschens.

In Tabelle 5.1 sind die beiden Möglichkeiten mit ihren Eigenschaften zusammengefasst. Es ist somit möglich, dass Löschen der Datensätze nicht transaktionssicher zu dem

Zeitpunkt durchzuführen, an dem der Benutzer das Löschen anstrebt oder mit dem Löschen zu warten, bis es transaktionssicher durchgeführt werden kann. In dieser Arbeit werden beiden Varianten implementiert und im Hinblick auf ihre Laufzeit evaluiert.

5.1.2 Implementierung

Bei der Implementierung wurde das Überschreiben des Datensatzes so modular wie möglich innerhalb einer Methode gekapselt. Der Quellcode dieser Methode ist in Listing 5.1 gezeigt. Als Parameter werden von dieser Methode die Seite (*Page*), auf der der Datensatz gespeichert ist, die *ItemID* des Datensatzes und die *HeadTupleHeader* Informationen des Datensatzes benötigt. Dabei sind im *HeadTupleHeader* Objekt die in Abschnitt 4.1.2 beschriebenen *HeadTupleHeaderData* Informationen gespeichert.

Damit beim Überschreiben ausschließlich die Werte des Datensatzes überschrieben werden, wird in Zeile 2 ein Zeiger auf das erste Byte der Benutzerdaten des Datensatzes gesetzt. Dazu werden die Speicheradresse der Seite sowie der Offset vom Start der Seite bis zum Anfang des Datensatzes benötigt. Außerdem muss beim Setzen des Zeigers noch beachtet werden, dass die `HeadTupleHeaderData` des Datensatzes nicht überschrieben werden. Da dieser Bereich eine variable Länge hat, werden die in `t_hoff` gespeicherten Informationen verwendet, um nur die Werte des Datensatzes zu überschreiben. Wird dieser Zeiger falsch gesetzt, können unvorhersehbare Seiteneffekte entstehen, die eventuell zum Absturz des DBMS führen. Nachdem dieser Zeiger gesetzt wurde, werden in den darauf folgenden Zeilen (Zeile 5-8 Listing 5.1) die Benutzerdaten des Datensatzes überschrieben. Dabei wird jede einzelne Speicheradresse, auf denen die Daten gespeichert sind, mit zufällig generierten Werten überschrieben.

Aufgrund der Kapselung kann die Funktion für beide Methoden, die im vorhergehenden Abschnitt dargelegt wurden, verwendet werden. Soll das Löschen der Datensätze wie in der zuerst vorgestellten Methode nicht transaktionssicher vollzogen werden, so muss die zusätzliche Funktionalität in den Methoden heap_delete(..) und heap_update(..) der Klasse `src/backend/access/heap/heapam.c` aufgerufen werden. Wenn das transaktionssichere Löschen vollzogen werden soll, muss die Funktion in der Methode HeadTupleHeaderAdvanceLatestRemoveXid(..) der Klasse `src/backend/access/heap/heapam.c` aufgerufen werden. Bei dieser Variante ist zusätzlich zu dem Aufruf der Methode noch der Kopf der übergeordneten Funktion anzupassen, um alle nötigen Informationen, die für das Löschen benötigt werden, zu besitzen.

Listing 5.1: Quellcode für das nicht rekonstruierbare Löschen eines Datensatzes in PostgreSQL.

```
1 void forensicDel(HeapTupleHeader tu, Page pa, ItemId id){
2  char* userData = ((char *) pa + id->lp_off + tu->t_hoff);
3  srand(time(NULL));
4  int bits = 0;
5  while (bits < (id->lp_len - tu->t_hoff)){
6   userData[bits] = (char) (rand()%255);
7   ++bits;
8  }
9 }
```

5.1.3 Analysesoftware

Der Aufbau der Dateien, in denen die Datensätze in einem PostgreSQL DBS gespeichert werden, ist, wie in Abschnitt 4.1.2 beschrieben, komplex. Deshalb kann durch eine einfache Betrachtung der Daten mittels hexadezimal Editor nicht mit Sicherheit festgestellt werden, dass alle Werte von gelöschten Datensätzen durch die beschriebene Methode überschrieben werden. Aus diesem Grund wurde im Rahmen dieser Arbeit ein Programm entwickelt, welches diese Dateien im Hinblick auf ihre Struktur und die in ihnen gespeicherten Werte analysiert. Der Grund für die Entwicklung eines solchen Programms ist die Tatsache, das es noch keine Programme gibt, mit deren Hilfe Informationen aus den Datenbankdateien heraus gelesen werden können, außer mit hexadezimal Editoren [SML07].

Wird eine Datei durch das Programm analysiert, wird es zuerst in ihre Seiten aufgeteilt. Ist diese Aufteilung geschehen, wird jede Seite einzeln analysiert. Diese Einzelanalyse der Seiten kann durchgeführt werden, weil es keine Datensätze gibt, für deren Rekonstruktion Informationen von mehreren Seiten benötigt werden. Bei der Analyse der Seiten wird zuerst der *PageHeaderData* Bereich ausgewertet. Aus diesem Bereich werden dann Informationen wie der Offset vom Seitenanfang zum Anfang des *Freespace* Bereichs geholt. Durch diesen Offset kann gleichzeitig auch ermittelt werden, wie viele Datensätze auf der Seite gespeichert sind. Bei diesen ermittelten Datensätzen handelt es sich um gelöschte, aber auch um aktive Datensätze. Diese Ermittlung ist möglich, weil für jeden Datensatz innerhalb des *ItemIDData* Bereichs ein 4 Bytes großer Bereich verwendet wird.

Durch eine Analyse des *ItemIDData* Bereichs kann die genaue Position der Datensätze ermittelt werden. Dabei muss dieser Bereich in 8 Byte große Abschnitte unterteilt werden, wobei in jedem Abschnitt der Offset und die Länge eines Datensatzes enthalten sind. Unter Zuhilfenahme dieser Informationen kann dann der Datensatz selbst ermittelt werden.

Bei der Auswertung der Datensätze ist zu beachten, dass in den ersten 24 Byte des Datensatzes die in Tabelle 4.2 beschriebenen *HeadTupleHeaderData* Metadaten gespeichert sind. Hier ist noch darauf zu achten, dass dieser Bereich auch länger sein kann. Es muss also der in `t_hoff` gespeicherte Wert analysiert werden, in dem die Länge eines potentiellen Offset zu den Benutzerdaten gespeichert ist.

Für die Analyse der Datensatzinhalte sind dann zusätzliche Informationen über das Schema der Tabelle, in der sich der Datensatz befindet, notwendig.

5.2 HSQLDB

Gegenstand der Betrachtung bezüglich HSQLDB war die Version 2.2.5. Bei dieser Version handelte es sich zum gegenwärtigen Zeitpunkt (November 2011) um die aktuelle Version des DBMS. Ziel der weitergehenden Betrachtung von HSQLDB ist es, zu verhindern, dass beim Löschen von Datensätzen DB-Slacks entstehen.

5.2.1 Analyse

Wie im Abschnitt 4.2 beschrieben, gibt es in HSQLDB drei unterschiedliche Tabellenarten, in denen Datensätze gespeichert werden können. Dabei handelt es sich um `TEXT`, `MEMORY` und `CACHED` Tabellen. Da von diesen Tabellenarten nur `CACHED` Tabellen DB-Slacks verursachen können, werden diese in der vorliegenden Arbeit betrachtet. Sollen Datensätze, die in solch einer Tabelle gespeichert sind, persistent auf dem Sekundärspeichermedium gespeichert werden, werden sie in ein Objekt gespeichert, das dann serialisiert wird. Bei der Analyse der vorliegenden Version hat sich herausgestellt, dass beim Ausführen einer `UPDATE` Operation die Funktionen zum Hinzufügen eines Datensatzes und zum Löschen eines Datensatzes verwendet werden. Es werden damit bei einer `UPDATE` Operation der alte Datensatz gelöscht und ein neuer Datensatz hinzugefügt. Dabei wird der neue Datensatz nicht unbedingt an die Stelle gespeichert, an der der alte Datensatz gespeichert war. Es können somit DB-Slacks entstehen.

Als Transaktionsverwaltung wird in HSQLDB ein zwei Phasen Sperrprotokoll verwendet. Dank dieses Sperrprotokolls kann es nicht vorkommen, dass mehrere Versionen eines Datensatzes gleichzeitig von unterschiedlichen Transaktionen verwendet werden. Somit kann das Überschreiben der nicht mehr benötigten Datensätze zu dem Zeitpunkt durchgeführt werden, zu dem die Transaktion, durch deren Ausführung der Speicherbereich frei geworden ist, erfolgreich abgeschlossen wird. Bei der Analyse hat sich jedoch herausgestellt, dass die Daten von beendeten Transaktionen nicht zu dem Zeitpunkt, an dem die Transaktion abgeschlossen wird, persistent auf den Sekundärspeicher geschrieben werden. Es erfolgt lediglich ein Eintrag in das Logbuch der Datenbank, um im Fehlerfall die Transaktion erneut ausführen zu können. Durch dieses Verhalten werden Datensätze also erst zu einem späteren Zeitpunkt persistent gespeichert. Dieses Verhalten ist dann problematisch, wenn ein Datensatz, der in einer früheren Version der Datenbank erstellt wurde, gelöscht werden soll. Da der Datensatz erst zu dem Zeitpunkt gelöscht wird, wenn sein Speicherbereich persistent geschrieben wird. Das kann im schlimmsten Fall erst beim Beenden der Datenbank-Instanz geschehen. Damit ein Datensatz zu dem Zeitpunkt, an dem er sicher gelöscht werden soll, auch nicht mehr in der Datei weiter existiert, ist es notwendig, die Propagierungsstrategie von HSQLDB zu erweitern. Es wurde entschieden, nach jedem erfolgreichen Beenden einer Transaktion ihre veränderten Werte auf dem Sekundärspeichermedium persistent zu speichern. Dies geschieht jedoch nur dann, wenn innerhalb der Transaktion Datensätze gelöscht oder verändert wurden.

5.2.2 Implementierung

Bei der Implementierung wurde die Methode erweitert, die beim erfolgreichen Abschließen einer Transaktion die Änderungen der Transaktion protokolliert. Dabei wird, wie schon erwähnt, die Protokollierung nur im Logbuch der Datenbank durchgeführt. Bei der erweiterten Methode handelt es sich um die Methode `commitRow(..)` der Klasse `org.hsqldb.persist.RowStoreAVLDisk`. Innerhalb dieser Methode wird unterschieden, ob ein Datensatz durch die abgeschlossene Transaktion eingefügt oder gelöscht wurde. Da nur das Verhalten beim Löschen von Datensätzen angepasst werden soll, wurde dieser Bereich der Methode um den in Listing 5.2 gezeigte Quellcode erweitert.

An diese Methode werden unter anderem die veränderte Zeile und ein Flag übergeben, in dem gespeichert ist, welche Änderungen an der Zeile durchgeführt wurden.

Listing 5.2: Quellcode für das nicht rekonstruierbare Löschen eines Datensatzes in HSQLDB.

```
1  Random ra = new Random();
2  this.cache.writeLock.lock();
3  byte[] object = new byte[row.getStorageSize()];
4  ra.nextBytes(object);
5  try {
6   this.cache.dataFile.seek((long)row.getPos()*this.cache.cacheFileScale);
7   this.cache.dataFile.write(object, 0, object.length);
8  } catch (IOException e) {
9   e.printStackTrace();
10 }
11 this.cache.commitChanges();
12 this.cache.writeLock.unlock();
```

In dem in Listing 5.2 gezeigten Quellcode wird in Zeile 1 ein Random Objekt erzeugt. Durch dieses Objekt können dann zufällige Werte generiert werden. Dabei wird der Erstellungszeitpunkt des Objektes in Nanosekunden als Seed für den Generator verwendet. Nachdem dies geschehen ist, wird in Zeile 2 die im Cache gehaltene Datei für andere Zugriffe gesperrt und am Ende der Veränderungen in Zeile 12 wieder entsperrt. Wird dieses Sperren der Datei nicht durchgeführt, kann es bei einem Mehrbenutzerbetrieb zu Problemen kommen. Zwischen diesem transaktionssicheren Bereich werden dann die Werte der gelöschten Datensätze überschrieben. Dazu wird in Zeile 3 ein Byte Feld erstellt, was die Speicherlänge des gespeicherten Datensatzes besitzt. Dieses Feld wird dann mit zufälligen Werten gefüllt (Zeile 4). In den Zeilen 6 und 7 wird dann das erstellte Feld an die Stelle des gelöschten Datensatzes geschrieben. Dazu werden die Position des alten Datensatzes und Informationen benötigt, welche Skalierung bei dem im Cache verwendeten Bereich benutzt wird. Nachdem der Speicherbereich des nicht mehr benötigten Datensatzes mit zufälligen Daten überschrieben wurde, müssen die Änderungen persistent gespeichert werden. Dazu wird in Zeile 11 eine Methode aufgerufen, durch die alle Änderungen, die an den Datensätzen durchgeführt wurden, persistent auf das Speichermedium geschrieben werden.

5.3 Zusammenfassung

In diesem Kapitel wurde beschrieben, wie die näher betrachteten Datenbankmanagementsysteme angepasst wurden. Dazu wurde analysiert, an welchen Stellen im Programmablauf ein Überschreiben der Werte des Datensatzes durchgeführt werden kann, ohne unnötig viele zusätzliche Operationen auszuführen. Als Besonderheit ist dabei zu erwähnen, dass in PostgreSQL zwei unterschiedliche Punkte existieren, in denen das Überschreiben der Werte potentiell durchgeführt werden kann. Außerdem wurde gezeigt, dass es in HSQLDB nicht ausreicht, das Überschreiben der Speicherbereiche durchzuführen. Es muss weiterhin noch eine Veränderung an der Propagierungsstrategie durchgeführt werden, um sicherzustellen, dass zu löschende Datensätze auch zeitnah sicher gelöscht werden. Des Weiteren wurde die Funktionsweise des Programms beschrieben, durch das die Dateien analysiert werden können, in denen von PostgreSQL Datensätze gespeichert werden.

6. Evaluierung

Wird ein Datensatz in den veränderten Versionen der betrachteten DBMS gelöscht, wird er durch die im vorhergehenden Kapitel beschriebene Implementierung beim Löschen mit zufällig generierten Daten überschrieben. Im Abschnitt 6.1 und Abschnitt 6.3 wird evaluiert, ob dieses zusätzliche Überschreiben der Werte der Datensätze eine Auswirkung auf die Ausführungszeit der DBMS darstellt. Des Weiteren wird im Abschnitt 6.2 mittels des in Abschnitt 5.1.3 beschriebenen Programms überprüft, ob der in PostgreSQL eingefügte Quelltext das sichere Löschen der Datensätze korrekt durchführt.

6.1 Evaluierung der in PostgreSQL durchgeführten Implementierung

Zur Evaluierung seiner Ausführungszeit besitzt PostgreSQL ein eigenes Modul mit dem Namen **pgbench**[1]. Mit ihm können über die Kommandozeile mit der Übergabe von verschiedenen Parametern unterschiedliche Benchmarks durchgeführt werden. Der Aufruf des in dieser Arbeit durchgeführten Benchmarks ist im nachfolgendem gezeigt.

```
./pgbench -r -t 5000 postgresDB -f ./pgBenchSkript.txt
```

Der Parameter `-r` bedeutet, dass die durchschnittliche Laufzeit pro Statement in dem durch pgbench erstellten Bericht zurückgegeben wird. Unter Zuhilfenahme dieses Parameters kann evaluiert werden, wie viel Zeit bei Lösch- beziehungsweise Updateoperationen benötigt wird. Mit dem Parameter `-t` kann die Anzahl der Transaktionen, die von jedem Client durchgeführt werden, gesetzt werden. Wird dieser Parameter nicht angegeben, ist er standardmäßig auf 10 gesetzt. Innerhalb des durchgeführten Tests wurde er auf 5000 gesetzt. Dies geschah, damit die Tests gegenüber kurzzeitigen Schwankungen robust sind, da in den ersten durchgeführten Tests sehr große Schwankungen verzeichnet

[1] http://developer.postgresql.org/pgdocs/postgres/pgbench.html, letzter Zugriff am 24.11.2011

wurden. Als dritten Parameter wird der Name der Datenbank angegeben, auf denen die Tests durchgeführt werden. Im durchgeführten Tests handelt es sich um eine Datenbank mit den Namen `postgresDB`. Durch den Parameter `-f` kann bei dem Benchmark ein eigenes Skript aufgerufen werden. Dafür wurde im Rahmen dieser Evaluierung ein eigenes Skript geschrieben, dessen Name `pgBenchSkript.txt` ist. Ausschnitte des Inhalts dieser Datei sind in Listing 6.1 gezeigt.

Listing 6.1: Ausschnitte des Skriptes, das für die für PostgreSQL durchgeführte Laufzeitevaluierung verwendet wurde.

```
1  \set naccounts 100000 * :scale
2  \setrandom firstInt 1 :naccounts
3  ...
4  \setrandom delta -5000 5000
5  BEGIN;
6  INSERT INTO oneInt (v1) VALUES (:firstInt);
7  UPDATE oneInt SET v1 = :firstInt +:delta WHERE v1 = :firstInt;
8  DELETE FROM oneInt WHERE v1 = :firstInt +:delta;
9  ...
10 END;
```

Generell wurde alle Benchmarks 10 mal durchgeführt. Dabei wurden Messungen mit unterschiedlich langen Datensätzen vollzogen. Dabei wurden exemplarisch Datensätze mit einer Länge von 4 und 20 Byte gelöscht und verändert. Da das Schreiben der zufällig generierten Werte unabhängig von dem Typ der Spalten des Datensatzes ist, wurde darauf verzichtet, eine Evaluierung gleichlanger Datensätze mit unterschiedlichen Typen durchzuführen.

Ausführungszeit in ms	DELETE (4Byte)	DELETE (20Byte)	UPDATE (4Byte)	UPDATE (20Byte)
ohne sicheres Löschen	0,44 (+/-0,1)	0,44 (+/-0,1)	0,50 (+/-0,1)	0,49 (+/-0,1)
sicheres Löschen (sofort)	0,44 (+/-0,1)	0,44 (+/-0,1)	0,51 (+/-0,1)	0,49 (+/-0,1)

Tabelle 6.1: Evaluierung der Laufzeitveränderung in PostgreSQL.

Die Ergebnisse der durchgeführten Evaluierung für das sofortige Überschreiben der Datensätze sind in Tabelle 6.1 gezeigt. In der Tabelle sind der Mittelwert der durchgeführten Tests sowie die Standardabweichung gezeigt. Die Messungen ergaben, dass durch das sofortige Überschreiben der Datensätze *keine* Laufzeiteinbußen verursacht werden.

Soll die zusätzlich benötigte Ausführungszeit für das transaktionssicheren Überschreibens getestet werden, kann nicht die Laufzeit beim Löschen beziehungsweise beim Ändern des Datensatzes gemessen werden, da das Überschreiben der Datensätze erst zu einem späteren Zeitpunkt vollzogen wird. Jedoch hat schon das nicht transaktionssichere Überschreiben gezeigt, dass durch die verwendete Implementierung keine Laufzeiteinbußen zu erwarten sind. Deshalb wurde an dieser Stelle darauf verzichtet, die zusätzliche Zeit, die für das Überschreiben der Datensätze nötig ist, durch einen Mittelwert über die gesamte Laufzeit der Tests zu berechnen. Es wurde stattdessen die Ausführungszeit der Operationen gemessen, durch die das Überschreiben der Datensätze durchgeführt

wird. Bei dieser Messung hat sich herausgestellt, dass auf dem verwendeten System sowohl für das Überschreiben von 4 Byte langen Bereichen als auch für das Überschreiben von 20 Byte langen Bereichen im Durchschnitt 10-12 Nanosekunden benötigt werden.

Für die Ausführungszeit des Systems ist es somit nicht von Relevanz, welche der vorgestellten Möglichkeiten des Überschreibens von Datensätzen verwendet werden, da keine dieser Varianten Laufzeiteinbußen verursacht. Der Grund dafür ist die Tatsache, dass bei dem Überschreiben der Werte nur auf Bereichen gearbeitet wird, die schon im Puffer des DBS gehalten werden. Außerdem werden auch in der schon in PostgreSQL existierenden Implementierung die Seiten, nachdem sie durch eine Transaktion verändert wurden, auf das Sekundärspeichermedium zurückgeschrieben.

6.2 Rekonstruierbarkeit gelöschter Datensätze in PostgreSQL

In diesem Abschnitt wird mithilfe des in Abschnitt 5.1.3 beschriebenen Programms evaluiert, ob alle Werte der Datensätze in der veränderten PostgreSQL Version überschrieben werden. Exemplarisch wird für diesen Test eine Tabelle angelegt, in die Personen mit ihrem Nachnamen und ihrem Vornamen eingetragen werden. Nachdem diese Personen eingetragen sind, wird eine von ihnen durch ein `UPDATE` verändert und die andere durch ein `DELETE` gelöscht. Die dabei ausgeführten SQL-Statements sind in Listing 6.2 gezeigt.

Listing 6.2: SQL Skript zur Evaluierung des Analyseprogramms.

```
1 CREATE TABLE persons (vorname varchar(20), nachname varchar(20));
2 INSERT INTO persons (vorname,nachname) VALUES ('Max','Musterman');
3 INSERT INTO persons (vorname,nachname) VALUES ('Erika','Mustermann');
4 UPDATE persons SET nachname = 'Mustermann' WHERE vorname='Max';
5 DELETE FROM persons WHERE vorname='Erika';
```

Nach der Ausführung des Skripts wurde die Datei, in der die Datensätze der `persons` Tabelle gespeichert sind, analysiert. Dies geschah durch das vorgestellte Programm. Bei der Analyse hat sich herausgestellt, dass die Werte der nicht mehr verwendeten Datensätze überschrieben wurden. Die Ergebnisse dieser Analyse sind in Tabelle 6.2 gezeigt. Dabei sind alle nicht darstellbaren Zeichen durch Fragezeichen ersetzt worden. Zur besseren Übersichtlichkeit wurden die einzelnen Operationen des Skriptes in einzelnen Transaktionen ausgeführt. Durch dieses Aufteilen des Skriptes ist in Tabelle 6.2 in den Feldern `t_xmin` und `t_xmax` zu sehen, welcher Datensatz durch welche Operation erstellt beziehungsweise als veraltet markiert wurde. So wird der Datensatz mit der Objekt-Nr. 0 durch das `INSERT` Statement aus der Zeile 2 (Listing 6.2) erstellt, weil die Transaktion des `INSERT` die Nr. 776 besitzt. Für die Erstellung des Datensatzes 1 ist die Operation aus Zeile 3 (Listing 6.2) verantwortlich. Wie schon in Abschnitt 4.1 beschrieben, wird in PostgreSQL beim Ändern eines Datensatzes für die neue Version des Datensatzes ein neuer Speicherbereich verwendet. Dieses Verhalten ist auch in Tabelle 6.2 zu erkennen. So wird durch die in Zeile 4 (Listing 6.2) durchgeführte `UPDATE` Operation der Datensatz mit der Objekt-Nr. 2 (Tabelle 6.2) erstellt. Außerdem wird der Datensatz mit der Nr. 0 als veraltet markiert. Dies geschieht durch ein Setzen seines

`t_xmax` Attributes. Abschließend wird durch die Zeile 5 des Skriptes der Datensatz mit der Objekt-Nr. 1 als nicht mehr benötigt markiert.

Objekt-Nr.	Start	Length	t_xmin	t_xmax	...	Values
0	8152	38	776	778	...	?????????????
1	8108	41	777	779	...	??????????????
2	8068	39	778	0	...	Max?Mustermann

Tabelle 6.2: Physisch gespeicherte Werte der `persons` Tabelle, nachdem das Skript aus Listing 6.2 ausgeführt wurde. Hierbei wurden die Datensätze durch die nicht transaktionssichere, vorgestellte Lösch-Variante überschrieben.

Durch die implementierte Methode kann somit das Entstehen von DB-Slacks durch das Löschen und Aktualisieren von Datensätzen in PostgreSQL verhindert werden.

6.3 Evaluierung der in HSQLDB durchgeführten Implementierung

Bei den durchgeführten Tests wurde versucht, Seiteneffekte so weit wie möglich auszuschließen. Dabei ist zu beachten, dass Java-Benchmarks nicht nur von der Hardware des verwendeten Systems beeinflusst werden, sondern auch von anderen Einflussfaktoren. Das sind beispielsweise die verwendete Virtuelle Maschine, die verwendete Heap-Größe und die Garbage Collection [GBE07]. Damit diese Störfaktoren weitestmöglich ausgeschlossen werden können, wurde bei den durchgeführten Tests immer die gleiche VM mit den gleichen Parametern verwendet. Außerdem wurden die Tests 20 mal durchgeführt, wobei die Ergebnisse der ersten 5 Tests nicht berücksichtigt werden, damit die Zeitmessung nicht durch Inline Optimierung verfälscht wird. Außerdem wurden in jedem der durchgeführten Tests 5000 Datensätze erstellt und auch wieder gelöscht beziehungsweise verändert. Dadurch wird der Einfluss der Inline Optimierung weiter verringert. Des Weiteren wurde für jeden der durchgeführten Tests eine eigene Datenbank erstellt. Ziel der Erstellung einer neuen Datenbank für jeden durchlaufenen Test war es, Einflüsse auf die Ausführungszeit, die zum Beispiel ein Caching verursacht, zu verhindern.

Ausführungszeit in ms	DELETE (4Byte)	DELETE (20Byte)	UPDATE (4Byte)	UPDATE (20Byte)
ohne sicheres Löschen	48,82 (+/-6,72)	55,69 (+/-4,79)	78,26 (+/-4,02)	80,3 (+/-4,25)
sicheres Löschen	70,86 (+/-8,76)	83,31 (+/-6,12)	101,35 (+/-2,17)	108,58 (+/-8,23)

Tabelle 6.3: Evaluierung der Laufzeitveränderung in HSQLDB.

Die Ergebnisse der durchgeführten Evaluierung sind in Tabelle 6.3 gezeigt. In der Tabelle ist zu erkennen, dass durch die durchgeführten Veränderungen signifikante Laufzeiteinbußen verursacht werden. Dabei ist zu sehen, dass durch die Implementierung sowohl wenn 4 Byte, als auch wenn 20 Byte lange Datensätze gelöscht werden, durchschnittliche Laufzeiteinbußen von ungefähr 25 Millisekunden erzeugt werden.

Im Anschluss an die hier vorgestellte Evaluierung wurde noch eine weitere Evaluierung durchgeführt. Durch sie sollte herausgefunden werden, durch welche der zusätzlich ausgeführten Operationen die Laufzeiteinbußen verursacht werden. Dabei hat sich herausgestellt, dass diese Einbußen durch das Propagieren entstehen. Der Grund für diese Laufzeitverschlechterung ist somit die Tatsache, dass in der veränderten Implementierung nach jedem erfolgreichen Abschließen einer Änderungstransaktion die Daten persistent auf das Speichermedium geschrieben werden. Somit werden anders als in der originalen Implementierung, die Änderungen von Transaktionen nicht mehr im Cache gehalten, sondern gleich nach dem Abschluss der Transaktion persistent gespeichert. Reicht es aus, die Datensätze erst nach dem Beenden der Datenbank Instanz sicher zu löschen, kann auch eine Implementierung verwendet werden, durch die keine Veränderungen in der Laufzeit entstehen.

6.4 Fazit

In diesem Kapitel wurde gezeigt, dass das Überschreiben der Werte von Datensätzen mit zufällig generierten Daten selbst keine Veränderung der Ausführungszeit von Transaktionen verursacht. Es muss jedoch darauf geachtet werden, zu welchem Zeitpunkt das Überschreiben der Werte vollzogen wird. Dabei wurde bei der für PostgreSQL durchgeführten Evaluierung gezeigt, dass in dem DBMS sowohl ein transaktionssicheres, als auch ein nicht transaktionssicheres Überschreiben der Datensätze ohne Laufzeitverluste durchgeführt werden kann.
Des Weiteren wurde mit der Evaluierung von HSQLDB dargelegt, dass die in PostgreSQL gemessenen Werte nicht verallgemeinert werden können. Der Grund dafür ist die Tatsache, dass in diesem DBS Änderungen nicht zu dem Zeitpunkt, an dem die Änderungstransaktion abgeschlossen ist, propagiert werden. Es ist jedoch notwendig dieses persistente Überschreiben der Datensätze so schnell wie möglich durchzuführen um sicherzustellen, dass die Daten auch sicher gelöscht sind. Somit ist es erforderlich, den Datensatz so schnell wie möglich zu überschreiben, wenn er nicht mehr benötigt wird. Somit kann das sichere Löschen von Datensätzen nicht in jedem DBMS ohne Laufzeiteinbußen durchführbar werden.

7. Zusammenfassung

Ziel dieser Arbeit war es, einen Einblick in die Problematik des forensisch sicheren Löschens von Datensätzen aus RDBMS zu geben. Dazu wurde analysiert, worauf generell geachtet werden muss, wenn elektronisch gespeicherte Informationen forensisch sicher gelöscht werden sollen. Da im Fokus dieser Arbeit das Löschen von Datensätzen aus DBS steht, wurde nachfolgend dazu auf den Aufbau dieser Systeme eingegangen. In diesem Zusammenhang wurde beschrieben, welche Informationen persistent von einem DBS gespeichert werden. Im Anschluss daran erfolgte eine Analyse, welche Bestandteile eines DBS angepasst werden müssen, um einen Datensatz forensisch sicher zu löschen. Aufbauend darauf wurde definiert, wann ein Datensatz nicht mehr rekonstruierbar ist und wann seine Werte als forensisch sicher gelöscht angesehen werden können.

Im zweiten Teil dieser Arbeit wurden dann exemplarisch bestehende DBMS analysiert. Zu diesem Zweck wurde beschrieben, welche Informationen eines Datensatzes, auch nachdem er gelöscht wurde, noch weiter in den Dateien der Datenbank enthalten sind. Dabei wurde festgestellt, dass in vielen DBS die Werte der Datensätze auch nach dem Löschen noch vollkommen auf Dateiebene vorhanden sind. Um dies zu verhindern, wurde bereits in existierenden Arbeiten eine Lösung entwickelt, durch die die Werte der Datensätze mit zufällig generierten Daten überschrieben werden. Jedoch wurde diese Lösung nur in ein DBMS exemplarisch integriert und evaluiert und nicht auf ihre allgemeine Anwendbarkeit hin überprüft. Aus diesem Grund wurde diese Lösung in zwei DBMS integriert. Bei den beiden Systemen handelt es sich mit PostgreSQL um ein C-basiertes und mit HSQLDB um ein Java-basiertes DBMS. Damit bei der Integration der Lösung keine unnötigen zusätzlichen Operationen durchgeführt werden, wurden die beiden Systeme analysiert. Bei dieser Analyse hat sich herausgestellt, dass in PostgreSQL das Überschreiben der Werte der Datensätze zu unterschiedlichen Zeitpunkten durchgeführt werden kann. Außerdem hat sich bei der Analyse von HSQLDB gezeigt, dass zusätzlich zu dem Überschreiben der Werte des gelöschten Datensatzes noch eine Veränderung der Propagierungsstrategie durchgeführt werden muss, um einen Datensatz zeitnah sicher zu löschen.
Abschließend wurde in dieser Arbeit überprüft, ob durch die integrierten Lösungen eine

Veränderung der Laufzeit der betrachteten Datenbankmanagementsysteme verursacht wird. Dabei hat sich herausgestellt, dass das Überschreiben der Datensatzwerte zu keiner Veränderung der Effizienz führt. Da aber die Propagierungsstrategie in HSQLDB angepasst werden musste, wurde bei den durchgeführten Tests eine Laufzeitverlängerung ermittelt. Somit kann gesagt werden, dass das Überschreiben der Datensatzwerte ohne Performanceeinbußen durchgeführt werden kann. Es muss jedoch darauf geachtet werden, zu welchem Zeitpunkt das Löschen durchgeführt wird und ob noch weitere Anpassungen an dem DBS durchgeführt werden müssen, um die Datensätze sicher zu überschreiben.

8. Zukünftige Arbeiten

In dieser Arbeit wurde eine Möglichkeit des sicheren Löschens von Datensätzen aus Datenbanken auf ihre Allgemeingültigkeit hin überprüft. Um jedoch einen Datensatz vollkommen forensisch sicher aus einem DBS zu löschen, ist mehr notwendig. Der Grund dafür liegt darin, dass Werte der Datensätze sowohl in den Metadaten als auch in den historischen Daten des DBS gespeichert werden.

Obwohl die Problematik des forensisch sicheren Löschens von Datensätzen aus Datenbanksystemen schon in einigen Arbeiten betrachtet wurde, existiert noch Forschungsbedarf.

Löschen von Datensätzen

In dieser Arbeit wurde gezeigt, wie das Löschen ganzer Datensätze aus Datenbanken durchgeführt werden kann. Jedoch ist es auch in einigen Anwendungsfällen notwendig, Teile von größeren Datensätzen sicher zu löschen. So kann es zum Beispiel erforderlich sein, bestimmte Informationen aus BLOBs zu löschen, während der Rest der Dateien noch erhalten bleiben soll. In diesem Zusammenhang muss noch untersucht werden, ob die verwendete Variante des sicheren Löschens in diesen Anwendungsfällen genutzt werden kann oder ob effizientere Möglichkeiten existieren.

Des Weiteren muss noch darauf geachtet werden, dass zwischen Datensätzen funktionale Abhängigkeiten bestehen können. Durch diese Abhängigkeiten ist es auch denkbar, dass Rückschlüsse auf die Werte sicher gelöschter Datensätze vollzogen werden können. In diesem Zusammenhang könnte sich in zukünftigen Arbeiten damit beschäftigt werden, Möglichkeiten zu finden, wie diese funktionalen Abhängigkeiten durch das System herausgefunden werden können, um den Anwender Informationen zur Verfügung zu stellen, dass solche Abhängigkeiten existieren.

Betrachtung von Metadaten

Wie schon erwähnt, müssen auch die datensatzabhängigen Metadaten des Datenbanksystems angepasst werden, wenn ein Datensatz sicher gelöscht werden soll. Dabei ist

es notwendig, alle über einen Datensatz angelegten Metadaten sicher zu löschen, oder die Daten dahingehend abzuändern, dass keine Rückschlüsse auf die Daten des Datensatzes druchgeführt werden können. In diesem Zusammenhang wurden auch schon Lösungen präsentiert, wie dieses Löschen aus Indexstrukturen in effizienter Zeit durchgeführt werden kann [SML07]. Jedoch wurde die Lösung nur für B^+-Bäume in InnoDB, einer MySQL Speicher-Engine evaluiert. Es ist somit noch notwendig, auch für andere Indexstrukturen zu überprüfen, ob ein sicheres Löschen von Datensätzen aus ihren gespeicherten Daten in effizienter Zeit durchgeführt werden kann. Außerdem besteht das Problem, dass einige Indexstrukturen, wie zum Beispiel der schon erwähnte B^+-Baum, historisch abhängig sind. Das bedeutet, dass aus den von ihnen gespeicherten Informationen Rückschlüsse auf die Einfügereihenfolge der von ihnen indexierten Datensätze gezogen werden können. Es muss also noch untersucht werden, wie historisch abhängige in historisch unabhängige Indexstrukturen umgewandelt werden können.

Es müssen auch Histogramme beachtet werden, wenn ein Datensatz sicher gelöscht werden soll. Wie in Abschnitt 2.4 beschrieben, werden zum Beispiel in SQL Server, Werte von Datensätzen als Grenzen der einzelnen Histogrammbereiche verwendet. Dadurch werden die Werte der Datensätze auch persistent innerhalb der Informationen der Histogrammen gespeichert. Außerdem besteht noch die Problematik, dass durch Analysen der in den Histogrammen gespeicherten Verteilungen und den noch in den Tabellen existierenden Datensätzen Rückschlüsse auf die Werte gelöschter Datensätze zogen werden können. Dabei muss noch überprüft werden, in wieweit die tatsächlichen Werte der Datensätze dabei bestimmt werden können.

Des Weiteren muss beim Löschen der über einen Datensatz angelegten Metadaten noch beachtet werden, dass von einigen Datenbankmanagementsystemen noch zusätzliche Metadaten gespeichert werden. Das bereits erwähnte MonetDB liefert ein Beispiel für dieses Problem. So werden in Anfragen berechnete aggregierte Werte persistent gespeichert. Soll ein Datensatz sicher gelöscht werden, müssen auch diese Informationen mit aus dem System entfernt werden. In diesem Zusammenhang muss überprüft werden, wie viel Zeit benötigt wird, um alle aggregierten Daten eines sicher zu löschenden Datensatzes, sicher zu löschen.

Historische Kopien der Datensätze

Bei der Protokollierung der an den Datensätzen durchgeführten Veränderungen innerhalb des Logbuchs, gibt es zwei verschiedene Arten der Speicherung der Änderungen. In Kapitel 3 wurde beschrieben, dass es sich dabei um die logische und um die physische Protokollierung handeln kann. Für die Problematik des sicheren Löschen von Datensätzen aus dem Logbuch eines DBS haben schon Stahlberg et al. [SML07] eine Möglichkeit vorgestellt, wie ein physisch protokolliertes Logbuch verändert werden kann, um zu verhindern, dass ein Datensatz nach der erfolgreichen Löschung wider hergestellt werden kann. Jedoch kann ihre Lösung nicht einfach auf ein logisch protokolliertes Logbuch angewendet werden. Der Grund dafür ist, dass bei einem logisch protokolliertem Logbuch, anderes als bei einem physisch protokolliertem, mehrere Datensätze durch einen Eintrag

im Logbuch verändert werden können. Somit können nicht einfach alle Änderungen, die den zu löschenden Datensatz verändert haben, aus dem Logbuch gelöscht werden. Es muss damit eine Möglichkeit gefunden werden, dass Logbuch der Datenbank in effizienter Zeit dahingehend anzupassen, dass gelöschte Datensätze nicht mehr wiederherstellt werden können, ohne dabei die Möglichkeit zu verlieren, noch verwendete Datensätze wiederherstellen zu können. Dabei muss darauf geachtet werden, dass in den einzelnen im Logbuch gespeicherten Informationen auch datensatzspezifische Werte gespeichert sein können, die dann gelöscht werden müssen. Dies darf jedoch nur dann passieren, wenn der Wert des Datensatzes einzigartig ist.

Wie schon in Abschnitt 2.4 beschrieben, werden auch zur größeren Datensicherheit Kopien der Datenbank auf einem Tertiärspeicher abgelegt. Wird ein Datensatz sicher aus dem DBS gelöscht, ist es auch notwendig, ihn aus den Backups der Datenbank zu löschen. Da die Datenbank jedoch keinen Zugriff auf die Backups besitzt, muss eine Möglichkeit entwickelt werden, Datensätze nach einer bestimmten Zeit aus den Backups zu löschen, ohne dabei die Backups vollkommen zu vernichten.

Datenbankmanagementspezifische Operationen

Generell kann nicht verhindert werden, dass ungewollte Kopien eines Datensatzes angelegt werden. Dies kann sowohl durch das Auslagern von im Arbeitsspeicher gespeicherten Informationen als auch durch den Benutzer geschehen. Jedoch nehmen auch viele Operationen des Datenbankmanagementsystems einen Einfluss auf die physische Repräsentation der Daten. Dabei muss beachtet werden, dass in unterschiedlichen Datenbankmanagementsystemen unterschiedliche Implementierungen für die Operationen verwendet sein können und das solche Operationen auch in unterschiedlicher Menge auftreten können. Ein Beispiel für eine solche Operation ist der `VACUUM` Prozess, der wie schon erwähnt zum Beispiel in PostgreSQL, MySQL, IBM DB2 und SQLite existiert. Durch ihn wird eine Umsortierung der Datensätze vorgenommen, durch die potentiell der vom DBMS benötigten Speicherplatz verringert werden soll. Dabei können DB-Slacks und auch DS-Slacks sowohl für gelöscht als auch für noch aktive Datensätze entstehen. Neben dieser Operation existieren auch noch andere Operation, die einen Einfluss auf die physische Repräsentation der Daten nehmen. So werden zum Beispiel in PostgreSQL während der Ausführung einer `CLUSTER` Operation auf einer Tabelle temporäre Kopien der Datensätze der Tabelle angelegt. In diesem Zusammenhang muss beachtet werden, dass, wenn in der Tabelle personenbezogene Daten gespeichert sind, diese Daten auch wieder gelöscht werden müssen.
Sollen Daten aus einem spezifischen DBMS sicher gelöscht werden, muss untersucht werden, welche durch das DBMS durchführbaren Operationen einen Einfluss auf die physische Repräsentation der Daten haben. Dabei ist es vor allem wichtig, eine datenbankmanagementspezifische Untersuchung durchzuführen.

Erweiterung der SQL Syntax

Schwerpunkt einer anderen zukünftigen Arbeit könnte einer Erweiterung der SQL Syntax sein, durch die ein feingranulares sicheres Löschen von Datensätzen unterstützt werden soll. Eine Vorbedingung dieser Arbeit ist das oben beschrieben Untersuchen, welche Operationen einen Einfluss auf die physische Repräsentation der Daten haben.

Außerdem müsste vorher noch untersucht werden, bei welcher der veränderten Operationen eine Änderung der Ausführungszeit entsteht. Durch diese Evaluierung könnte dann argumentiert werden, inwieweit eine feingranulare Unterscheidung einen Vorteil verspricht.

Besondere Arten von DBMS

Neben den in dieser Arbeit betrachteten DBMS gibt es noch eine weitaus größere Anzahl von DBMS, die dahingehend betrachtet werden müssen, wie ein Datensatz sicher aus ihnen gelöscht werden kann. Dabei ist auch zu beachten, dass in einigen Anwendungsfällen besondere Arten von DBMS zu Einsatz kommen. Dabei kann es sich sowohl um verteilte oder föderierte Datenbanken als auch um Data Warehouses handeln.

Bei verteilten und föderierten Datenbanken ist dabei zu beachten, dass innerhalb des Netzwerkes der Datenbanksysteme redundante Kopien der Datensätze vorliegen können. Diese müssen natürlich auch mit gelöscht werden. Des Weiteren muss darauf geachtet werden, dass das sichere Löschen der Datensätze auch dann gewährleistet werden muss, wenn ein Teil des Netzwerkes, in dem sich der Datensatz befindet, zum Löschzeitpunkt nicht erreichbar ist. Außerdem muss verhindert werden, dass Informationen über die Werte der Datensätze innerhalb des zugrunde liegenden Netzwerkprotokolls gespeichert werden. Des Weiteren muss bei föderierten DBS noch darauf geachtet werden, dass in dem Netzwerk auch unterschiedliche DBMS zum Einsatz kommen können.

Sollen Datensätze aus einer Datenbank sicher gelöscht werden, deren Daten in regelmäßigen Abständen in ein Data Warehouse übertragen werden, so muss der Datensatz auch aus dem Data Warehouse sicher gelöscht werden. Bei dem Löschen des Datensatzes aus dem Data Warehouse ist dann zum Beispiel auch zu beachten, dass der Datensatz nicht durch aggregierte Werte, die im Data Warehouse gespeichert sind, wiederhergestellt werden kann. Dabei muss noch eruiert werden, ob es ausreicht, das Löschen des Datensatzes aus dem Data Warehouse zum nächsten Update Zeitpunkt des Data Warehouse durchzuführen, oder ob das Löschen aus dem Data Warehouse zu einem früheren Zeitpunkt geschehen soll. Außerdem ist zu analysieren, in wieweit Kopien von Datensätzen während der Überführung des Datensatzes aus der Datenbank in das Data Warehouse angelegt werden. Da die Datensätze durch den extraktions-transformations-lade-Prozesses (ETL) auch verändert und zwischengespeichert werden, ist dieses Entstehen von Kopien sehr wahrscheinlich. Es muss damit noch einen Möglichkeit gefunden werden, wie diese Kopien auch wieder sicher gelöscht werden können.

Ganzheitliche Lösungen und Veränderungen zur Laufzeit

In der Vergangenheit wurden schon Lösungen vorgestellt, wie ein sicheres Löschen von Datensätzen aus Teilen der vom DBS gespeicherten Informationen durchgeführt werden kann. Dabei wurden auch bereits Evaluierungen der Laufzeit durchgeführt. Es muss jedoch noch evaluiert werden, welche Einbußen eine Lösung verursacht, durch die ein Datensatz sicher gelöscht wird. In diesem Zusammenhang kann auch analysiert werden, ob Laufzeitunterschiede bestehen, wenn ein Datensatz forensisch sicher oder nicht rekonstruierbar gelöscht werden soll. Dabei ist es dann auch notwendig, einen einheitlichen Benchmark zu entwickeln, durch den unterschiedliche Systeme, die ein sicheres Löschen von Datensätzen durchführen können, verglichen werden.

Außerdem muss noch darauf geachtet werden, dass es zu Änderungen in gesetzlichen Bestimmungen kommen kann. Werden in einem DBS Daten gespeichert, die sich ändernden Bestimmungen unterliegen, ist es notwendig das DBS dahingehend anzupassen, dass die gesetzlichen Vorgaben umgesetzt werden. Bei solchen Anpassungen ist es im Normalfall nötig, das DBS zu beenden, die Änderungen einzupflegen und dann das DBS neu zu starten. Dabei kann jedoch das Problem auftauchen, dass das DBS zu einer Anwendung gehört, die sieben Tage in der Woche, jeweils 24 Stunden erreichbar sein muss. Außerdem besteht noch die Problematik, dass viele DBS eine gewisse Laufzeit benötigen, um eine optimale Ausführungszeit von Operationen zu erlangen. Es wäre somit von Vorteil, die Änderungen an dem DBS während der Laufzeit durchführen zu können. Dabei muss unterschieden werden, ob das DBS in einer Sprache geschrieben ist, die statisch oder dynamisch getypte ist. Handelt es sich bei der Sprache um eine dynamische getypte Sprache, können die Änderungen auch während der Laufzeit durchgeführt werden. Dies gilt jedoch nicht, wenn eine statisch typisierte Programmiersprache bei der Entwicklung des DBS verwendet wurde. Jedoch wurden in diesem Bereich in den letzten Jahren einige Lösungen entwickelt, durch die ein Großteil aller an einem Programm durchführbaren Änderungen zur Laufzeit durchgeführt werden kann [PKC+12]. Es muss jedoch noch evaluiert werden, in wieweit diese Lösungen aus DBS angewendet werden können, da diese Lösungen meinst nur für eine Programmiersprache gezeigt wurden.

Literaturverzeichnis

[ABPA+09] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proc. VLDB Endow.*, 2:922–933, August 2009. (zitiert auf Seite 34)

[AMH08] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 967–980, New York, NY, USA, 2008. ACM. (zitiert auf Seite 30)

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. (zitiert auf Seite 17)

[Bon02] Peter Alexander Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Ph.d. thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002. (zitiert auf Seite 30 and 34)

[BP01] Steven Bauer and Nissanka B. Priyantha. Secure data deletion for Linux file systems. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, pages 12–12, Berkeley, CA, USA, 2001. USENIX Association. (zitiert auf Seite 5)

[Bun04] Bundesministeriums der Justiz. Telekommunikationsgesetz, 2004. (zitiert auf Seite 1)

[Cod82] E. F. Codd. Relational database: a practical foundation for productivity. *Commun. ACM*, 25(2):109–117, February 1982. (zitiert auf Seite 1)

[Con96] United States Congress. Health insurance portability and accountability act (hipaa). http://www.hhs.gov/ocr/privacy/, 1996. (zitiert auf Seite 1)

[Dat86] Database Architecture Framework Task Group (DAFTG) of the ANSI/X3/SPARC Database System Study Group. Reference Model for DBMS Standardization. *Sigmod Records*, 15(1):19–58, March 1986. (zitiert auf Seite 6, 7, and 8)

[Dat04] Chris J. Date. *An Introduction to Database Systems*. Pearson Addison-Wesley, Boston, MA, 8. edition, 2004. (zitiert auf Seite 15)

[FHMW10] Peter Fruhwirt, Marcus Huber, Martin Mulazzani, and Edgar R. Weippl. InnoDB Database Forensics. *Advanced Information Networking and Applications, International Conference on*, 0:1028–1036, 2010. (zitiert auf Seite 42)

[fJ11] Bundesministerium für Justiz. Eckpunktpapier zur Sicherung vorhandener Verkehrsdaten und Gewährleistung von Bestandsdatenauskünften im Internet, Januar 2011. (zitiert auf Seite 1)

[Fow08] Kevvie Fowler. *SQL Server Forensic Analysis*. Addison-Wesley Professional, 2008. (zitiert auf Seite 12, 15, 18, 41, and 42)

[Fox09] Dirk Fox. Sicheres Löschen von Daten auf Festplatten. *Datenschutz und Datensicherheit - DuD*, 33:110–113, 2009. (zitiert auf Seite 5 and 22)

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA*, pages 57–76. ACM, 2007. (zitiert auf Seite 52)

[Gro11] The PostgreSQL Global Development Group. PostgreSQL: Documentation: PostgreSQL 9.1,1. http://www.postgresql.org/docs/9.1/static/, 2011. (zitiert auf Seite 35, 36, and 42)

[Gro12] The HSQL Development Group. Hypersql user guide. http://hsqldb.org/doc/guide/, 2012. (zitiert auf Seite 40)

[Gue10] Shary Gueron. *Intel® Advanced Encryption Standard (AES) Instructions Set*, January 2010. (zitiert auf Seite 4)

[Gut96] Peter Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *In Proceedings of the 6th USENIX Security Symposium*, pages 77–89, 1996. (zitiert auf Seite 5, 23, and 28)

[HS00] Andreas Heuer and Gunter Saake. *Datenbanken: Konzepte und Sprachen (2. Auflage)*. MITP, 2000. (zitiert auf Seite 6 and 8)

[IKNG09] Milena G. Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo A.P. Gonçalves. An Architecture for Recycling Intermediates in a Column-store. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 309–320, New York, NY, USA, 2009. ACM. (zitiert auf Seite 17)

[LAHG05] David Litchfield, Chris Anley, John Heasman, and Bill Grindlay. *The Database Hacker's Handbook: Defending Database Servers*. John Wiley & Sons, 2005. (zitiert auf Seite 34)

[Lit07a] David Litchfield. Oracle forensics part 1: Dissecting the Redo Logs. *NGS-Software Insight Security Research (NISR) Publication, Next Generation Security Software,(Version dated 21 March 2007*, 2007. (zitiert auf Seite 18)

[Lit07b] David Litchfield. Oracle forensics part 2: Locating dropped objects. *NGS-Software Insight Security Research (NISR) Publication, Next Generation Security Software,(Version dated 24 March 2007*, 2007. (zitiert auf Seite 41 and 42)

[Loc98] Thomas Lockhart. Postgresql tutorial. URL, 1998. (zitiert auf Seite 35)

[LS87] Peter C. Lockemann and Joachim W. Schmidt, editors. *Datenbankhandbuch*. Springer, 1987. (zitiert auf Seite 9)

[Mar09] Martin S. Olivier. On metadata context in Database Forensics. *Digital Investigation*, 5(3-4):115–123, 2009. (zitiert auf Seite 6, 12, and 13)

[Mic73] Michael E. Senko and Edward B. Altman and Morton M. Astrahan and P. L. Fehder. Data Structures and Accessing in Data-Base Systems. I: Evolution of Information Systems. *IBM Systems Journal*, pages 30–93, 1973. (zitiert auf Seite 8 and 10)

[MLS07] Gerome Miklau, Brian Neil Levine, and Patrick Stahlberg. Securing history: Privacy and accountability in database systems. In *CIDR*, pages 387–396. www.crdrdb.org, 2007. (zitiert auf Seite 6, 13, and 17)

[Mom01] B. Momjian. *PostgreSQL: introduction and concepts*. Addison-Wesley, 2001. (zitiert auf Seite 38)

[MRD10] Ivana Miniarikova, João Rodrigues, and Pedro Dias. Overview of PostgreSQL 9.0 , 2010. (zitiert auf Seite 34 and 36)

[NIS01] NIST. *Advanced Encryption Standard (AES) (FIPS PUB 197)*. National Institute of Standards and Technology, November 2001. (zitiert auf Seite 4)

[PKC+12] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schröter, and Gunter Saake. JavAdaptor – Flexible Runtime Updates of Java Applications. *Software: Practice and Experience*, 2012. early view. (zitiert auf Seite 61)

[Sch07] Stefan Schumacher. Daten sicher löschen. *UpTimes*, 3 2007. (zitiert auf Seite 4, 5, and 23)

[SHS05] Gunter Saake, Andreas Heuer, and Kai-Uwe Sattler. *Datenbanken - Implementierungstechniken (2. Auflage)*. MITP, 2005. (zitiert auf Seite 6, 8, 9, 10, 12, 16, 17, 28, and 29)

[SML07] Patrick Stahlberg, Gerome Miklau, and Brian Neil Levine. Threats to Privacy in the Forensic Analysis of Database Systems. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 91–102, New York, NY, USA, 2007. ACM. (zitiert auf Seite 4, 5, 13, 14, 16, 17, 18, 28, 29, 46, and 58)

[SR86] Michael Stonebraker and Lawrence A. Rowe. The Design Of Postgres. In *IEEE Transactions on Knowledge and Data Engineering*, pages 340–355, 1986. (zitiert auf Seite 34)

[Sto87] Michael Stonebraker. The Design of the POSTGRES Storage System. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 289–300. Morgan Kaufmann, 1987. (zitiert auf Seite 14 and 38)

[Tra11] Florin Cosmin Trandafir. Legal and Practical Aspects in the Computer Science Investigation. *Journal of Mobile, Embedded and Distributed Systems, vol. III, no. 2*, pages 82–90, 2011. (zitiert auf Seite 4)

[UNI02] DAS EUROPÄISCHE PARLAMENT UND DER RAT DER EUROPÄISCHEN UNION. Richtlinie 2002/58/eg des europÄischen parlaments und des rates, 12. Juli 2002. (zitiert auf Seite 1 and 3)

[Vos94] Gottfried Vossen. *Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme. 2. Auflage*. Addison-Wesley, 1994. (zitiert auf Seite 6, 8, 10, 11, 12, and 16)

[WD02] John C. Worsley and Joshua D. Drake. *Practical PostgreSQL. A Hardened, Robust, Open Source Database.* O'Reilly, January 2002. ISBN-10: 1565928466 ISBN-13: 978-1565928466. (zitiert auf Seite 34)

[Wri08] Wright, Craig and Kleiman, Dave and Sundhar R.S., Shyaam. Overwriting Hard Drive Data: The Great Wiping Controversy. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 243–257, Berlin, Heidelberg, 2008. Springer-Verlag. (zitiert auf Seite 4, 5, and 22)

[WV01] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. (zitiert auf Seite 18)

[ZFZ+08] Hong Zhu, Ge Fu, Yi Zhu, Renchao Jin, Kevin Lü, and Jie Shi. Dynamic data recovery for database systems based on fine grained transaction log. In Bipin C. Desai, editor, *IDEAS*, volume 299 of *ACM International Conference Proceeding Series*, pages 249–253. ACM, 2008. (zitiert auf Seite 18)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 27. Februar 2012