Zugriffskontrolle in Webdatenbanken mit Query Rewriting

Michael Rossel · Benjamin Große · Saša Prijović · Peter Trommler

Georg-Simon-Ohm Hochschule Nürnberg
michael.rossel@gmx.de, benjamin.grosse@i-woas.net
sasa@prijovic.de, peter.trommler@ohm-hochschule.de

Zusammenfassung

Webanwendungen bieten indirekt Zugriff auf firmeninterne Datenbanken und können bei Fehlern in der Autorisierung missbraucht werden, um unberechtigt auf Daten zuzugreifen. Im Sinne einer „Defence in Depth“ Strategie kann eine zweite Autorisierung in der Datenbank die Sicherheit der Webanwendung verbessern. In diesem Beitrag wird eine Spezifikationssprache mit einem einfachen Modell präsentiert und untersucht, wie mittels Query Rewriting die Anfragen an die Datenbank derart modifiziert werden können, dass nur autorisierte Zugriffe auf die Daten zugelassen werden. Erste Experimente mit einem Prototypen zeigen, dass die mit Query Rewriting erzeugten Queries unwesentlich mehr Aufwand an Rechenzeit benötigen.

1 Einleitung

Webanwendungen spielen eine zentrale Rolle in der Umsetzung von elektronischen Geschäftsprozessen, insbesondere mit Endkunden. Dazu lesen und verändern sie Daten auf internen Datenbanken nach dem Geschäftsprozess entsprechenden Regeln. Gemäß diesen Regeln ist eine eingeschränkte Menge an Operationen auf der Datenbank zulässig. Die Autorisierung der Operationen ist als ein Teil des Geschäftsprozesses zu verstehen und als solcher auch mit der übrigen Geschäftslogik zu implementieren.

Wird die Autorisierung fehlerhaft implementiert, so wird dieser Fehler im Fall einer zu Unrecht abgewiesenen Operation (z. B. ein Konto fehlt in der Liste der Konten) als Problem der Geschäftsregeln aufgefasst. Anders stellt sich die Interpretation einer fehlerhaften Autorisierung, bei der fälschlicherweise zu viel Information gegeben wird, dar. In diesem Fall wird von einer Sicherheitslücke gesprochen.

Die Problematik besteht darin, dass die Autorisierung über den gesamten Code der Webanwendung verteilt ist und damit zum Nachweis der Vollständigkeit und Korrektheit der Autorisierung auch der gesamte Code betrachtet werden muss. Diese Überprüfung ist damit nicht von der Fachabteilung leistbar, es entsteht ein Bruch zwischen den Spezifikationsdokumenten und der Umsetzung der Spezifikation im Code.

Wird nun die Spezifikation ausreichend formalisiert, dann kann diese als Konfiguration eines Autorisierungsmechanismus dienen. Dieser Autorisierungsmechanismus kann in verschiedener Weise in eine Webapplikation oder deren Umgebung eingebunden werden. Mittels einer Web Application Firewall kann bereits vor der eigentlichen Webapplication geprüft werden, ob eine Anfrage des Kunden zulässig ist. Zur Integration der Autorisierung in die Webapplikation ste-

N. N · N. N (Hrsg.) · D•A•CH Security 2012 · syssec (2012) 1-12.

hen Mechanismen wie Aspect Oriented Programming zur Verfügung, mit denen Autorisierung in die Webanwendung automatisch eingewoben werden kann. Schliesslich kann die Autorisierung in der Datenbank erfolgen, indem die Anfragen an die Datenbank auf Zulässigkeit geprüft werden.

In allen drei Fällen werden die Identität des Benutzers der Webanwendung und der Inhalt der Datenbank für eine Autorisierungsentscheidung herangezogen. Die Web Application Firewall hat über das Verfolgen der Session-Id leicht Zugriff auf die Identität des Benutzers, muss jedoch separat auf die Datenbank zugreifen, um Informationen für die Autorisierungsentscheidung zu erhalten. Die Integration in die Webanwendung scheitert bei den meisten Programmierumgebungen an der Unterstützung für Aspect Oriented Programming. Bei der Umsetzung in der Datenbank muss lediglich in der Webanwendung die Session-Id bis zur Datenbank durch gereicht werden.

Unter den Aspekten Performance und Implementierungsaufwand erschien uns die Implementierung der Autorisierung in der Datenbank als die beste Lösung. Dabei soll diese Autorisierung jedoch nicht die Autorisierung in der Webanwendung ersetzen, sondern diese im Sinne von „Defence in Depth“ ergänzen. Wir beziehen uns dabei auf einen Vorschlag von Roichman und Gudes [RoGu07], der das Konzept einer parametrisierten View zur Umsetzung der feingranularen Autorisierung in der Datenbank beschreibt.

Parametrisierte Views sind kein Standardmechanismus in SQL und werden dementsprechend nach unserem Wissen auch von keiner Datenbank unterstützt. Neben dem Vorschlag von Roichman und Gudes, parametrisierte Views mit User defined Functions zu implementieren, gibt es weitere Ansätze [RMSR04], die eine Umsetzung durch Query Rewriting beschreiben. Die so umgeschriebenen Queries entsprechen dann Standard SQL und können auf jedem standardkonformen Datenbank-Management-System ausgeführt werden.

In Abschnitt 2 stellen wir ein Modell für die Formulierung der Autorisierungsregeln einschließlich einer Spezifikationssprache vor. Im folgenden Abschnitt 3 zeigen wir, wie aus der Spezifikation der Autorisierungsregeln eine parametrisierte View abgeleitet wird und wie eine SQL Query mit parametrisierten Views in eine Standard SQL Query transformiert wird. Abschnitt 4 stellt verwandte Arbeiten vor und im Abschnitt 5 findet sich ein Ausblick auf zukünftig geplante Arbeiten.

2 Modell der Zugriffskontrolle

In diesem Abschnitt wird kurz ein fein-granulares Modell der Zugriffskontrolle vorgestellt. Das Modell soll dabei drei Anforderungen Rechnung tragen, die im Rahmen einer Studie [PrTr11] an einem Projekt aus der Pflegeunterstützung erarbeitet und validiert [Trom12] wurden:

1. Zugriff auf eine Teilmenge der Daten einer Tabelle sowie deren Spalten.
2. Zugriff auf die *eigenen Daten* eines Benutzers innerhalb dieser Teilmenge. Die Beschränkung erfolgt auf Zeilenebene.
3. Zugriff auf Daten ist von Aktionen, die in der Vergangenheit erfolgten, abhängig.

Anhand dieser Anforderungen werden im Abschnitt 2.1 wichtige Begriffe des Modells erläutert. Abschnitt 2.2 stellt eine Spezifikationssprache vor, mit der Personal auf einer nicht-technischen Ebene die Möglichkeit gegeben werden soll, Zugriffskontrollregeln selbst festzulegen oder zumindest besser zu verstehen.

2.1 Elemente des Modells

Unser Sicherheitsmodell lässt sich in vier Begriffen beschreiben:

- Kategorien: Spezialform einer Rolle.
- Principals: Wer braucht Zugriff?
- Ressourcen: Auf was bzw. auf welche Zeile soll Zugriff gegeben werden?
- Aktionen: Welche Operationen sind für bestimmte Ressourcen zulässig?

Die Daten, die einem Principal einer Kategorie zur Verfügung gestellt werden, sollen als seine eigenen Daten bezeichnet werden. Nicht jeder Principal soll über die gleichen Daten verfügen oder die gleichen Manipulationsrechte besitzen. Darin liegt der Unterschied zu einer Rolle. Eine Kategorie ist somit als ein Spezialfall bzw. eine Spezialisierung der Rolle zu betrachten.

Diese Begriffe sind als ein Mengenkonstrukt zu verstehen, sodass eine Security Policy eine Relation auf diesen Mengen ist. Relationen innerhalb dieses Konstrukts bilden dabei die Kategorien und Principals (*PCA*, **P**rincipals und **Ca**tegories) sowie Ressourcen und Aktionen (*PAR*, **P**rincipals, **A**ctions und **R**essources). Ein Principal kann mehreren Kategorien angehören, sodass sich folgende Relation ergibt $PCA \subseteq P \times C$. Mit der Relation $PAR \subseteq P \times A \times R$ wird der Zugriff von Principals auf Ressourcen mit Aktionen definiert. Hierzu werden die Ressourcen definiert sowie die auf den Ressourcen zugelassenen Operationen [Bark08].

2.1.1 Modellierung des Zugriffs auf Ressourcen

Für eine domänenspezifische Sprache zur Spezifikation von fein-granularer Zugriffskontrolle müssen folgende Elemente vorhanden sein:

- View
- Navigation
- Anker
- Spur (Trace)

Mit Hilfe einer View können Daten spezifiziert werden, die für einen Benutzer einsehbar sind und die derjenige bearbeiten kann. Eine Datenbank View kann hierfür eingesetzt werden.

Zusätzlich zur Einschränkung durch die View, schränkt die Navigation den Zugriff auf einzelne Datentupel ein. Dabei wird die aus UML-Diagrammen bekannte Navigation für den Zweck der Zugriffskontrolle so interpretiert, dass die Datentupel der Tabelle, an der die Navigation beginnt, bestimmen, welche Datentupel der Zieltabelle zugreifbar sind. Durch die Navigation wird also auf der Ebene des einzelnen Datentupels ein Graph definiert, der über die Erreichbarkeitsrelation die Menge der zugreifbaren Datentupel bestimmt.

Die Navigation hat den Ursprung im Anker, der ein Datentupel ist, das für den an der Web-Anwendung angemeldeten Benutzer steht. Verschiedene Benutzerkategorien können dabei in unterschiedlichen Tabellen verwaltet werden.

Für eine Zugriffskontrolle sind zudem temporale Aspekte bedeutend. Die Zulässigkeit einer Aktion hängt in diesem Fall von anderen Aktionen in der Vergangenheit ab. In dem Fall, dass die Aktion im Rahmen des Geschäftsfalls in der Datenbank in einem Datensatz dokumentiert wird, z.B. durch das Anlegen eines neuen Datensatzes oder auch durch das Löschen eines Datensatzes, wird diese Information für die Entscheidung über den Zugriff herangezogen. Hinterlässt der

Geschäftsfall jedoch keine „natürliche“ Spur in der Datenbank, wie dies z. B. beim Lesen eines Datensatzes der Fall ist, muss die betreffende Aktion zusätzlich eine Spur in der Datenbank anlegen.

2.1.2 Manipulation der Daten nach CRUD

Sind die zugreifbaren Datensätze (Ressourcen) festgelegt, muss noch spezifiziert werden, welche Operationen auf diesen Datensätzen zulässig sind. Bei Datenbanken wird dabei häufig von CRUD-Operationen gesprochen, die wir auch hier für die Gruppierung der Operationen verwenden wollen. Für die Autorisierung muss also für Create, Read, Update sowie Delete spezifiziert werden, was zulässig ist.

Das Lesen (Read) wird durch die View und die Navigation über diese geregelt. Jeder bekommt dadurch das zu sehen was er sehen darf.

Das Ändern (Update) von Daten gestaltet sich einfach, da dies generell nicht möglich ist, sofern es sich nicht um die eigenen Daten handelt. Bei der Änderung von Daten, die in der Navigation verwendet werden, ist der Fall nicht so einfach, da eine solche Aktion ggf. die Rechte von Datensätzen ändert. Ob dies zulässig ist, hängt von der Security Policy ab. Wenn diese eine Rechteweitergabe nicht vorsieht gibt es zwei Möglichkeiten. Die Erste ist eine strikte Ablehnung von Änderungen. Die zweite besteht darin die möglichen Werte derart einzuschränken, dass das Ergebnis immer noch den eigenen Daten entspricht

Das Einfügen (Create) von Datensätzen gestaltet sich analog zum Ändern. Löschen (Delete), sofern es zulässig ist, gestaltet sich wie das Lesen einfach, da hier als Parameter lediglich Datensätze der eigenen Daten mitgegeben werden und auch nur diese gelöscht werden können.

2.2 Domänenspezifische Sprache

In diesem Abschnitt wird eine Spezifikationssprache nach den oben genannten Anforderungen entwickelt. Das Endprodukt der Spezifikation ist eine Beschreibung einer parametrisierten View für jede Kategorie von Principals.

Zur Beschreibung einer solchen View werden Anker, Navigation und Zugriff angegeben. Der Anker wird durch eine Abbildung der Information aus der Authentisierung (z.B. ein Benutzername) auf einen Datensatz in einer bestimmten Tabelle festgelegt. Für die Navigation wird festgelegt, welche Assoziationen navigiert werden können, um in der Zieltabelle die zulässigen Datensätze zu bestimmen. Schließlich werden die zulässigen Operationen auf den so ausgewählten Datensätzen direkt als erlaubte Aktion (read, write, . . .) oder als SQL View angegeben.

```
View ::= ViewName ":" Anchor Navigation Access
Anchor ::= "Anchor:" Tablename "=" Mapping ";"
Navigation ::= Navigationitem+ ";"
Navigationitem ::= Tablename "->" Tablename "VIA" Linkspec
Linkspec ::= Tablename "." Columnname
           | Tablename "." Columnname
               "<->" Tablename "." Columnname
           | Linkspec "AND" Predicate
Access ::= Tablename ":" Accessspecifier+ ";" | SQLView
```

3 Query Rewriting

Das Konzept des Query Rewriting für fein-granulare Zugriffskontrolle beschreibt einen Prozess (siehe Abb. 1), bei dem vor der Ausführung einer SQL Query diese so umgeschrieben wird, dass sie nur auf einer Teilmenge des Datenbestandes ausgeführt wird. Hierdurch entsteht eine personalisierte Form einer Rolle, die Kategorie (siehe Abschnitt 2), in der ihre Benutzer über unterschiedliche Rechte verfügen können.

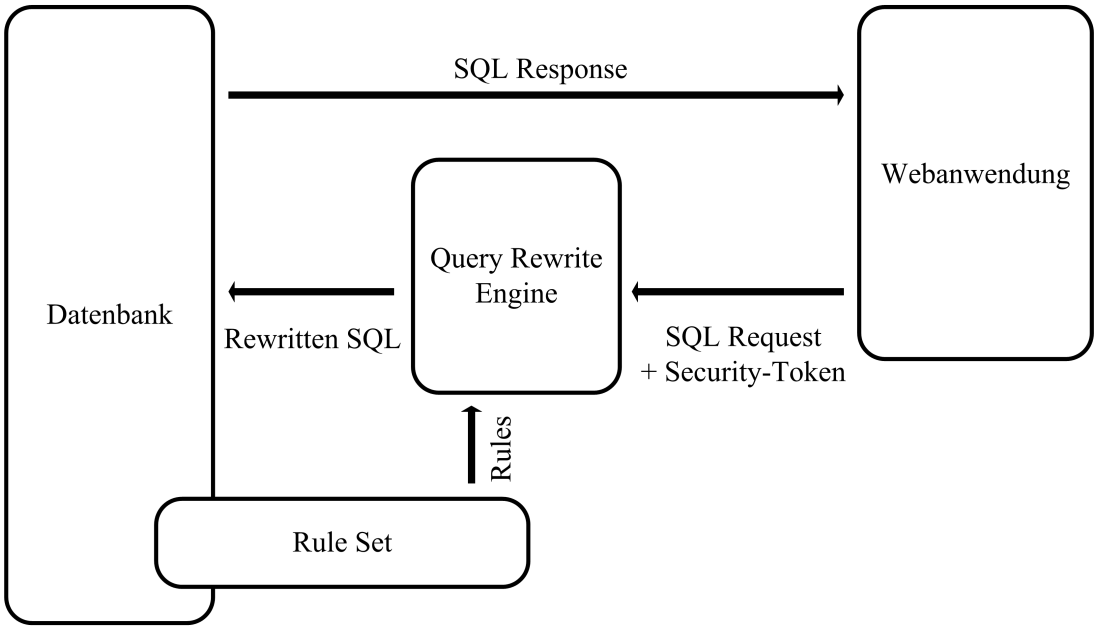

Abb. 1: Query Rewriting Schema

Das Umschreiben der SQL Queries geschieht für die Webanwendung transparent. Es wird lediglich ein Security-Token von der Webanwendung mit übergeben, das den zugreifenden Benutzer eindeutig identifiziert. Dieses Token wird dem Viewnamen in Klammern angehängt: `VIEW(Token)`. Dieses Security-Token wird bei Authentisierung des Benutzers zu Beginn einer Web-Session in der Datenbank generiert und an den Benutzer gegeben. Bei jeder SQL Query an die Datenbank wird dieser Token mit übergeben. Durch ihn lässt sich eine eindeutige Zuordnung des Principals und seiner Kategorie bestimmen. Das Eingrenzen des zugreifbaren Datenbestandes wird, mit Hilfe dieses Security-Tokens, in zwei Schritten erledigt. Der Erste umfasst die Beschränkung der vertikalen Ebene einer Tabelle, den Spalten. Dies erfolgt durch die in der Anwendung benutzte View. Im zweiten Schritt werden die Zugriffsberechtigungen auf horizontaler Ebene, den Zeilen, für den jeweiligen Benutzer gewählt. Hierzu werden, ausgehend vom Benutzer als Ankerelement, die Tabellen der Datenbank genutzt, um die Verfügbarkeit der Datentupel mit geschachtelten Selects weiter einzugrenzen. Der genaue Ablauf wird beispielhaft im folgenden Abschnitt 3.1 erläutert.

3.1 Beispiele für Query Rewriting

Um das Vorgehen zu verdeutlichen, wird im Folgenden beispielhaft auf die SQL-Grundfunktionen Insert und Select eingegangen. Alle restlichen SQL Queries werden analog verarbeitet, sodass sich eine vollständige Kapselung der Datenbank durch Query Rewriting ergibt.

Alle Beispiele werden an Hand einer fiktiven Webanwendung gezeigt, mit deren Hilfe ein Lehrbetrieb unterstützt wird. Sämtliche hierzu benötigten Daten werden in einem DBMS (Datenbank Management System) gehalten. An der Weboberfläche können sich Dozenten sowie Studenten anmelden. Es wird ein vereinfachtes Datenmodell (siehe Abb. 2) verwendet, mit dessen Hilfe Dozenten, Studenten sowie ihre Kurse abgelegt werden. Jedem Kurs ist mindestens ein Dozent zugeordnet. Eine Anmeldung eines Studenten zu einem Kurs wird durch einen Eintrag in der Tabelle *kurs_anmeldung_student* abgebildet. Eine Anmeldung für eine kursbegleitende Prüfung wird durch einen Eintrag in der Tabelle *klausur_anmeldung_student* festgehalten. Jede der Tabellen enthält aus Gründen der Übersichtlichkeit einen nummerischen Primärschlüssel, der automatisch vergeben wird.

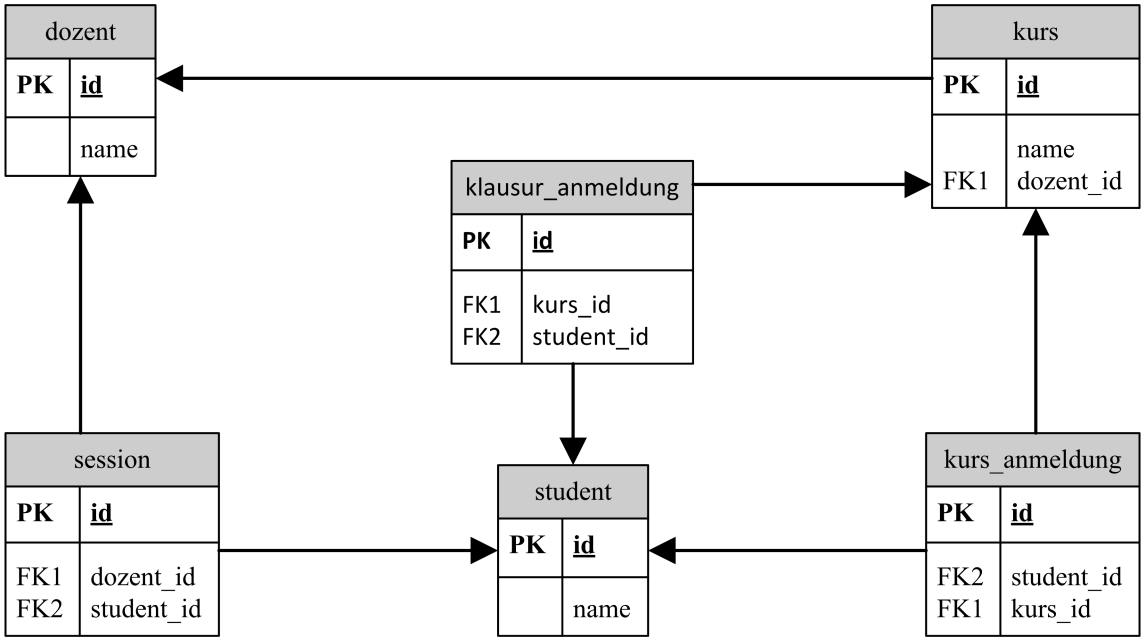

Abb. 2: Schema der Beispiel Datenbank

3.1.1 Der Select Befehl

Der Darstellung des SELECT Befehls wollen wir zunächst die Definition der domänenspezifischen Sprache voranstellen. Das Ankerobjekt stellt die userid des jeweiligen Student dar. Diese wird mit Hilfe des Security-Tokens ermittelt, das durch die Webanwendung an die Datenbank in Form eines Parameters für eine View übermittelt wird. Wie an der Sprache zu erkennen ist, wird von vom Studenten auf die Kursanmeldung über die entsprechende ID navigiert. Der Zugriff ist nur lesend und wird mit dem zugehörigen Flag gekennzeichnet.

```
Anchor: student.id = userid;
student -> kurs_anmeldung_student
        VIA kurs_anmeldung_student.student_id;
kurs_anmeldung_student: read only;
```

Das Ziel beim Query Rewriting eines Select Query ist sicherzustellen, dass Benutzern nur ihre eigenen Daten ausgegeben werden. Die hierzu benötigte Identifikation des Benutzers wird über das erwähnte Security-Token ermittelt, das der Query beigefügt wird [RoGu07].

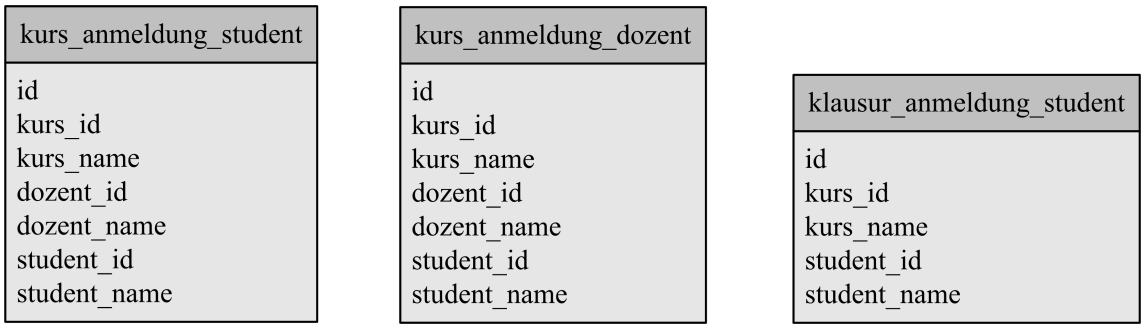

Abb. 3: Views für vereinfachten Zugriff auf Datenbank

Zur Einsichtnahme der Kursanmeldungen in der Webapplikation steht Studenten und Dozenten jeweils eine eigene View zur Verfügung (kurs_anmeldung_student, kurs_anmeldung_dozent), wie in Abbildung 3 dargestellt. Um zu verhindern, dass Studenten und Dozenten Einträge einsehen können, für die sie nicht freigegeben sind, sollen Abfragen dieser Views nur Ergebnisse zurückliefern, wenn Einsatzbereich und die Art, des in der Applikation angemeldeten Benutzers übereinstimmen. Weiterhin wird eine aktive Sitzung (Eintrag in der Tabelle Session) der Benutzer vorausgesetzt.

Ein Student mit der Session 123 möchte sich eine Übersicht seiner Anmeldungen zu Grundkursen (der Name soll die Zeichenfolge 101 enthalten) ausgeben lassen. Dieses Statement wird an die Datenbank gesendet:

```
SELECT * FROM kurs_anmeldung_student(123)
WHERE kurs_name LIKE '%101%'
```

Für die View *kurs_anmeldung_student* ist definiert, dass ein Parameter für einen Rewrite benötigt wird und dieser in eine vorhandene oder noch anzufügende Where-Klausel integriert wird: *student_ID = (SELECT student_id FROM session WHERE ID = :param_1)* . Der hier verwendete Parameter (:param_1) wird durch die Zeichenfolge in den Klammern hinter dem Viewnamen ersetzt. Die Klammern und deren Inhalt werden dann aus der SQL-Abfrage entfernt.

```
SELECT * FROM kurs_anmeldung_student
WHERE (kurs_name LIKE '%101%') AND
      (student_id = (SELECT student_id FROM session
                                       WHERE ID = 123))
```

Die Ergebnisse des erzeugten Selects werden grundsätzlich nach der ID des Studenten gefiltert, unabhängig von der vorangegangenen Where-Klausel.

3.1.2 Der Insert Befehl

Um sich für eine Prüfung anzumelden, muss der Student auch für den gleichen Kurs angemeldet sein. Für eine gültige Anwendung braucht dieser Schreibrechte. Der Student muss erst über den Kurs navigieren und von dort aus auf die Klausuranmeldung. Die Navigation erfolgt zunächst über die ID des Studenten im Kurs. Diese muss vorhanden sein, sofern er angemeldet ist. Anschließend folgt der Navigationsweg angefangen beim Kurs zur Klausuranmeldung des Kurses selbst. Navigiert wird hier über die ID des Kurses, um die letztlich mit beiden Daten die

Kursanmeldungsview zu beschreiben und den Anmeldeprozess damit zu beenden. Das entsprechende Flag muss auf create gesetzt werden, da der Student Schreibrechte benötigt, um sich für seinen Kurs anmelden zu können.

```
Anchor: student.id = userid;
student -> kurs_anmeldung_student
        VIA kurs_anmeldung_student.student_id;
kurs_anmeldung_student -> klausur_anmeldung_student
        VIA kurs_anmeldung_student.student_id <->
            klausur_anmeldung_student.student_id
        AND kurs_anmeldung_student.kurs_id <->
            klausur_anmeldung_student.kurs_id;
klausur_anmeldung_student: create;
```

Dieses Beispiel zeigt weiterhin einen temporalen Aspekt der Zugriffskontrolle. Die Anmeldung für einen Kurs stellt eine Aktion in der Vergangenheit dar, die bestimmt ob eine Anmeldung zur Prüfung in der Zukunft möglich ist. Die Beschränkung kann alleine durch die Existenz eines entsprechenden Datensatzes bei den Anmeldungen zu den Kursen geprüft werden. Zudem wird dieser Datensatz im neu anzulegenden Datensatz der Prüfungsanmeldung referenziert.

Mit Query Rewriting ist es möglich Zugriffskontrollen für Inserts zu implementieren, indem die Values-Clause durch ein Subselect ersetzt wird. Die Werte der Values-Clause werden als die Spalten des Subselects übernommen. Sollte das Select keine Zeilen zurückliefern, wird kein Datensatz angelegt.

Der Student mit der ID 16313 möchte sich für die Klausur des Kurses mit der ID 5 anmelden. Dies soll ihm durch das System nur dann gestattet werden, wenn er auch für diesen Kurs angemeldet ist und er über eine gültige Anmeldung an das System verfügt.

```
INSERT INTO klausur_anmeldung_student(123)(kurs_id,student_id)
  VALUES(5, 16313)
```

Für den Rewrite wurde definiert, dass nur Sätze angelegt werden dürfen, wenn der Student auch für den Kurs angemeldet war. Das Subselect benötigt hierzu die ID des Kurses und des Studenten. Der Zugriff auf die View *kurs_anmeldung_student* erfolgt wie in Abschnitt 3.1.1 beschrieben. Der hierfür benötigte Parameter wird dem für das Insert verwendeten View *klausur_anmeldung_student* angehängt.

```
INSERT INTO klausur_anmeldung_view(kurs_id,student_id)
SELECT 5, 16313 FROM kurs_anmeldung_view
WHERE student_id = (SELECT student_id FROM session WHERE ID = 123)
AND (kurs_id = 5
  AND student_id = 16313);
```

Die horizontale Selektion in diesen Beispielen wurde durch die verwendeten Views realisiert. Alternativ kann dies auch für den Query Rewrite definiert werden, indem man je Kategorie festlegt, welche Spalten eines Views diese einsehen oder ändern darf.

3.2 Technische Umsetzung

In diesem Abschnitt wird die Umsetzung des Query Rewriting in PostgreSQL betrachtet. Die schematische Darstellung des Query Rewriting Prozesses in Abbildung 4 zeigt, an welcher

Stelle der Queryverarbeitung eingegriffen und die *Query Rewrite Engine* integriert wird.

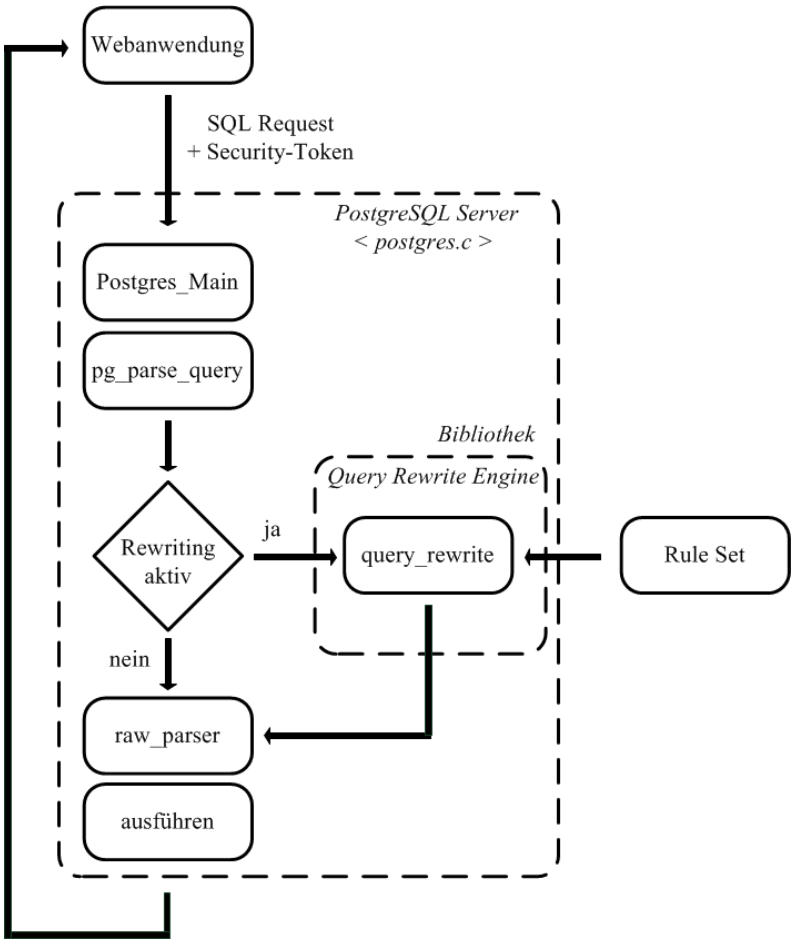

Abb. 4: Schema Queryverarbeitung

Der Ablauf der Queryverarbeitung wird vor dem *raw_parser* unterbrochen. An dieser Stelle wird die Query in einer dynamisch eingebundenen Bibliothek, an Hand des Regelsatzes (*Rule Set*) zur Autorisierung, umgeschrieben. Hierbei wird das Security Token ausgewertet und daraus das Ankerobjekt bestimmt, dessen Primary Key in die Query eingefügt wird. Das Ergebnis ist eine standardkonforme SQL Query, welche wieder in den regulären Ablauf der PostgreSQL Queryverarbeitung zurück gegeben wird. Dies geschieht durch Übergabe an den *raw_parser*. Das Ergebnis der modifizierten Query wird schließlich direkt an die Webanwendung zurück gegeben.

Alternativ zu dieser Vorgehensweise bietet sich die Modifikation des Quellcodes selbst an. Dieser Weg wurde nicht verfolgt, da in diesem Entwicklungsstadium zunächst ein Proof of Concept durchgeführt werden soll. Das zukünftige Ziel ist, dieses Verfahren in die offiziellen Quellen nativ zu integrieren.

Eine weitere Alternative ist die Entwicklung eines Proxy, der zwischen Datenbank und Webanwendung eingesetzt wird und das Query Rewriting durchführt. Aus der Sicht der Webanwendung verhält sich der Proxy wie eine beliebige Datenbank. Für die Datenbank sieht der Proxy wie die Webanwendung aus.

3.3 Performance

Dieser Abschnitt befasst sich mit der in Abschnitt 3.2 vorgestellten Implementierung. Die Performancetests umfassen hierbei drei Fälle, in denen jeweils 1000 Abfragen abgesetzt werden. Die Messungen wurden auf einem handelsüblichen PC durchgeführt.

Es werden die Kursanmeldungen aus Abschnitt 3.1 zu einem Studenten oder Dozenten, abhängig von der Session, selektiert. Der erste Fall behandelt die Option, dass die Webanwendung direkt die richtige Query an die Datenbank schickt. Dies bedeutet, dass eine legitime Abfrage erfolgt und für den autorisierten Benutzer die entsprechenden Daten angezeigt werden.

Der zweite Fall entspricht einem Standardfall. Ein Benutzer greift auf einen Datenpool zu, der Daten enthält, die nicht für ihn bestimmt sind. Das Ergebnis der Abfrage entspricht demnach einer Teilmenge, die durch einen Query Rewrite aus dem Datenpool herausgenommen wird.

Der dritte Fall behandelt das bloße Absetzen einer Query, wie sie als Ergebnis nach einem Rewrite wie in Testfall 2 entsteht. Dieser Test dient dazu die Performance der Query mit Rewrite mit der Query ohne Rewrite zu vergleichen.

Als Durchschnitt mehrerer Messungen ergaben sich folgende Zeiten:

- **Testfall 1**: 0:29 min
- **Testfall 2**: 3:03 min
- **Testfall 3**: 0:30 min

Die Implementierung ist etwa sechs Mal langsamer als eine Query ohne Autorisierung. Jedoch zeigt Testfall 3, dass die erweiterte Query nicht für die schlechtere Ausführungszeit verantwortlich ist. Im weiteren wird zu untersuchen sein, wie die Implementation und Integration der Rewrite Engine effizienter bewerkstelligt werden kann. Wie im voran gegangenen Abschnitt 3.2 angedeutet wurde, wird eine vollständige Integration in PostgeSQL langfristig die beste Option sein.

4 Verwandte Arbeiten

Die Thematik des Query Rewriting wurde bereits im Kontext von Data-Warehouse Architekturen untersucht [RoSc00]. Dort wurde der Begriff der Zugriffskontrolle in zwei Bereiche, auf „Was“ zugegriffen und „Wer“ darauf zugreifen darf, aufgeteilt. Die Untersuchung lieferte eine Möglichkeit, an das Data-Warehouse geschickte Anfragen abzufangen und derart zu verändern, dass eine äquivalente Anfrage entstand, sofern der jeweilige Nutzer die nötigen Zugriffsrechte auf der Ressource hatte.

[RoSD99] setzt die Idee des Query Rewriting fort und zeigt wie Rechteverwaltung auf Basis der gegebenen Daten und nicht Funktionen für verteilte Datenbanken funktionieren könnte. Wie in dieser Untersuchung erwähnt, greifen [RMSR04] die vorangegangenen Arbeiten des Query Rewriting auf und erweitern diese Technik um die Bedingung zur Überprüfung des Datenbankstatus. Das Papier zeigt, dass eine äquivalente Umformung einer Abfrage nicht reicht, da diese nicht in allen Fällen die gleichen Ergebnisse liefern. Dies wird in Anlehnung an einen Kinofilm als der Truman-Effekt bezeichnet.

Eine Alternative zum Query Rewriting stellt die Verwendung von parametrisierten Views in [RoGu07]. Die Autoren zeigen, wie mit User Defined Functions aus Standard SQL eine sogenannte parametrisierte View erzeugt wird, der ein Security Token übergeben wird und letztlich

nur die Daten liefert, für die der jeweilige Benutzer autorisiert ist. Dies unterscheidet sich von unserem Ansatz dadurch, dass die parametrisierte View in der Datenbank nachgebildet wird und nicht, wie in in unserer Darstellung, als konzeptionelles Konstrukt ausschließlich in der Webanwendung zu finden ist.

Folgende Arbeiten zielen nicht darauf ab die Zugriffskontrolle in der Datenbank abzuwickeln, liefern aber Ansatzpunkte für die grafischen Notation der hier vorgestellten Sprache. [Juer05] liefert in seinem Beitrag eine Erweiterung der UML um Bausteine, die einzelne Daten entsprechend markiert, um den Grad des Schutzes für das jeweilige Datum zu bestimmen. Anschließend können in Abhängigkeit solcher Markierungen entsprechende kryptografische Anforderungen definiert werden.

Ein Konzept zur Erstellung von sicherem Code zeigt [BaDL06]. Das Konzept basiert auf den Gedanken der modellgetriebenen Softwareentwicklung, mit der es möglich ist Teile der Webanwendung auf Basis eines Modells generieren zu lassen.

[WYLL+07] erweitern das Verfahren von [RMSR04] um einen Beweis der Korrektheit eines Algorithmus, der die definierte Security Policy für feingranulare Zugriffskontrolle umsetzt. Weiterhin stellen die Autoren eine Möglichkeit vor, die Zugriffskontrolle auf Zellebene erfolgen zu lassen und entsprechende Zellen zu maskieren.

5 Ausblick

Die hier vorgestellte Vorgehensweise wird im Zuge einer Bachelorarbeit für eine PostgreSQL Datenbank implementiert. Als nächster Schritt wird untersucht, wie die Leistung der Query Rewriting Engine und deren Integration in PostgreSQL verbessert werden kann.

Es wird ferner untersucht wie ein Kapern der Session durch Abfangen des Security Tokens verhindert werden kann. Hierzu wir unter anderem die Verwendung von rollierenden Schlüsseln diskutiert und nach performanteren Lösungen gesucht.

Mit Hilfe der DSL aus Abschnitt 2.2 wird ein Codegenerator entwickelt, der die Implementierung für PostgreSQL automatisiert. Somit wird ein Werkzeug zur Verfügung gestellt, das es einem Domänenexperten ermöglicht während der Modellierung eines Geschäftsprozesses direkt die Zugriffsregeln zu definieren, ohne dabei ein tieferes technisches Verständnis haben zu müssen.

Ferner wird auf Grund der Ergebnisse die textuelle DSL in eine grafische Notation überführt, um sie als Erweiterung von UML und ER-Diagrammen nutzen zu können.

Literatur

[BaDL06] D. Basin, J. Doser, T. Lodderstedt: Model driven security: From UML models to access control infrastructures. In: *ACM Trans. Softw. Eng. Methodol.*, 15, 1 (2006), 39–91.

[Bark08] S. Barker: Access control by action control. *In: Proceedings of the 13th ACM symposium on Access control models and technologies*, ACM, New York, NY, USA (2008), 143–152.

[Juer05] J. Juerjens: Secure Systems Development with UML. Springer Verlag, Heidelberg (2005).

[PrTr11] S. Prijovic, P. Trommler: Zugriffskontrolle in der Datenbank: Vamos eine Fallstudie. *In: P. Schartner, J. Taeger (Hrsg.), D-A-CH Security 2011* (2011), 147 – 156.

[RMSR04] S. Rizvi, A. Mendelzon, S. Sudarshan, P. Roy: Extending query rewriting techniques for fine-grained access control. *In: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, ACM, New York, NY, USA (2004), 551–562.

[RoGu07] A. Roichman, E. Gudes: Fine-grained access control to web databases. *In: Proceedings of the 12th ACM symposium on Access control models and technologies*, ACM, New York, NY, USA (2007), 31–40.

[RoSc00] A. Rosenthal, E. Sciore: View security as the basis for data warehouse security. *In: DMDW* (2000), 8.

[RoSD99] A. Rosenthal, E. Sciore, V. Doshi: Security Administration for Federations, Warehouses, and other Derived Data. *In: DBSec* (1999), 209–223.

[Trom12] P. Trommler: A Model for Fine-Grained Access Control to Web Databases: VAMOS—A Case Study. *In: P. Kommers, P. T. Isaías (Hrsg.), Proceedings of the IADIS International Conference Internet Applications and Research 2012, 17-19 July, Lisbon, Portugal*, IADIS Press (2012).

[WYLL+07] Q. Wang, T. Yu, N. Li, J. Lobo, E. Bertino, K. Irwin, J.-W. Byun: On the correctness criteria of fine-grained access control in relational databases. *In: Proceedings of the 33rd international conference on Very large data bases*, VLDB Endowment (2007), 555–566.