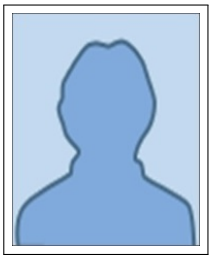
Technology
Arts Sciences
TH Köln

Technische Hochschule Köln
Campus Gummersbach

Praxisprojekt

Planung und Umsetzung einer erweiterten, relationalen Mehrkampfdatenbank mit Mehrbenutzerunterstützung unter Verwendung des Laravel-PHP-Frameworks und MariaDB (Fortführung des Informatikprojektes)

Matthias Frank
110 961 67



Daniel Morady
110 975 11

betreut durch
Prof. Dr. Birgit Bertelsmeier

21. Oktober 2016

Inhaltsverzeichnis

1. Eine kurze Projektbeschreibung

Das Praxisprojekt ist eine Fortführung des bereits umgesetzten Informatikprojektes.
Daher wird in den nachfolgenden Kapiteln ausschließlich auf die maßgeblichen Änderungen und Erweiterungen Bezug genommen.

Der Schwerpunkt ist die Konzeption und Umsetzung eines relationalen Datenbanksystems zur Erfassung, Verwaltung und Bearbeitung von leichtathletischen Mehrkampfwettbewerben.
Die Bedienung erfolgt über eine Weboberfläche, die Mithilfe des PHP-Frameworks Laravel[1] an das Datenbanksystem MariaDB[2] angekoppelt ist.

Die wichtigsten Themen im Projekt sind:

- Die Umsetzung und Koordinierung des Mehrbenutzerbetriebs
- Die Umsetzung und Kontrolle der Rechteverwaltung der Benutzer
- Die Möglichkeit zur Erfassung eigener Wettkämpfe bestehend aus beliebig vielen Disziplinen inklusive individueller Formeln zur Punkteberechnung

Die ersten beiden Punkte werden in dieser Ausarbeitung auf einer allgemeinen Ebene behandelt.

[1] [13]
[2] [14]

2. Motivation

Diese Ausarbeitung befasst sich vermehrt mit den theoretischen Problematiken in den verschiedenen Kapiteln. Die Grundlagen hierzu wurden aus den Erfahrungen in der Umsetzung des Praxisprojektes gesammelt und ausgewertet.

In Kapitel 3 *Mehrbenutzerbetrieb* werden die verschiedenen Besonderheiten und Problemstellungen für das Datenbanksystem bei der gleichzeitigen Anwendung durch mehrere Benutzer gezeigt. Außerdem werden unterschiedliche Lösungsansätze betrachtet.

In Kapitel 4 *Rechteverwaltung* werden zwei unterschiedliche Ansätze zur Verwaltung von Zugriffs- und Bearbeitungsrechte, in Bezug auf das Datenbanksystem, behandelt. Es werden Vor- und Nachteile beider Verfahren aufgeführt und die möglichen Einsatzbereiche abgegrenzt.

In Kapitel 5 *Relationale Grenzen* werden die theoretischen und praktischen Grenzen von relationalen Datenbankmodellen abgesteckt. Zusätzlich werden weitere, von außen auf den Modellansatz einwirkende, Beschränkungen betrachtet.

In Kapitel 6 *Pair-Programming* geht es um die Entwicklungs- und Arbeitstechnik des *Pair-Programmings*. Dabei werden kurz die Vor- und Nachteile aufgeführt und der Einsatz im Praxisumfeld beschrieben.

Im abschließenden Kapitel 7 *Erweiterte Datenbankfunktionen* werden kurz einige erweiterte Elemente in relationalen Systemen beleuchtet. Insbesondere werden dabei mögliche Vor- und Nachteile in der Verwendung anhand einer praktischen Umsetzung betrachtet und bewertet.

3. Probleme im Mehrbenutzerbetrieb

> "Mehrbenutzerbetrieb: gleichzeitiger Zugriff mehrerer Benutzer auf die Datenbank."
>
> — Kleinschmidt und Rank[1]

Dieses Kapitel behandelt eines der großen Anwendungsprobleme bei Datenbanken, den gleichzeitigen Zugriff mehrerer Benutzer auf ein Datenbanksystem. Dabei können zwei Arten von Zugriff unterschieden werden, der lesende und der schreibende Zugriff.

Im ersten Abschnitt werden praxisrelevante Probleme bei lesenden, im zweiten Abschnitt bei schreibenden Zugriffen beschrieben und mögliche Lösungsansätze verglichen. Alle Lösungen sind durchaus praktikabel und es muss im jeweiligen Projekt entschieden werden, welcher Ansatz den eigenen Anforderungen am Besten gerecht wird.

Zunächst werden kurz vier grundlegende Probleme des Mehrbenutzerbetriebs bei der Ausführung von Transaktionen betrachtet:

Nonrepeatable Read

Ein mehrfach aus der Datenbank ausgelesener Wert muss innerhalb einer Transaktion gleich sein, auch wenn ein anderer Benutzer zwischenzeitlich Änderungen vornimmt.
Beispiel:
Ein Datenbanksystem speichert die Informationen einer (Fußball-)Liga. Der Benutzer A fragt in einer Transaktion zuerst nach dem Vereinsnamen des aktuell Erstplatzierten und dann nach den Spielergebnissen dieses Vereins.
1a. *Selektiere Vereinsname des Vereins mit der Platzierung 1*
1b. *Selektiere Spielplan des Vereins mit der Platzierung 1*
Ein anderer Benutzer tauscht die Platzierungen von Verein A und B.
2a. *Setze Platzierung von Verein A auf alte Platzierung von Verein B*
2b. *Setze Platzierung von Verein B auf alte Platzierung von Verein A*
Erfolgt die Ausführung der einzelnen Aufgaben in der Reihenfolge 1a, 2a, 2b, 1b so erhält der Benutzer A zwar den Namen von Verein A (als ursprünglicher Erstplatzierter), jedoch den Spielplan von Verein B (als neuer Erstplatzierter).

Phantome

Datensätze, die nicht über den gesamten Zeitraum einer Transaktion existieren, dürfen nicht zu fehlerhaften Berechnungen führen.
Beispiel:

[1] [11, S. 5]

An die Mitarbeiter eines Unternehmens soll ein Bonus von 100.000 Euro ausgezahlt werden. Zuerst wird die Anzahl der Mitarbeiter durch eine Zählung aller Zeilen der Mitarbeitertabelle ermittelt. Der insgesamt auszuschüttende Betrag wird entsprechend durch diese Anzahl aufgeteilt. Bevor jedoch die Erfassung des Bonus bei allen Mitarbeitern in einer Tabellenspalte durchgeführt ist, wird ein neuer Mitarbeiter von einem anderen Benutzer erfasst. Da auch dieser Mitarbeiter bei der anschließenden Übermittlung einen Bonus erhält wurden mehr als die 100.000 Euro an Boni ausgezahlt.

Dirty Read

Vor dem (erfolgreichen) Abschluss einer Transaktion dürfen die getätigten Änderungen nicht für andere Benutzer sichtbar sein.
Beispiel:
Für eine Sprechstunde beim Professor können sich die Studenten in einer Datenbank eintragen, wobei die Zeitvergabe automatisch aufsteigend erfolgt. Student A bekommt einen Terminvorschlag für 10 Uhr, da um 9 Uhr bereits ein Termin vorliegt, und trägt sich dafür ein. Für Student B wird der (vorläufige) Eintrag von A gefunden und ein Termin für 11 Uhr zurückgegeben. Die Verbindung von A wird leider unterbrochen und das Datenbanksystem macht ein Rollback. Jetzt erst wird der Termin für B ins Datenbanksystem geschrieben und A versucht es erneut, erhält nun aber einen Termin um 12 Uhr (da ja 11 Uhr bereits belegt ist und die Zuweisung streng linear erfolgt). Der Termin um 10 Uhr ist nicht mehr belegbar, da die Abfrage immer nur den möglichen Eintrag nach dem letzten Termin zurückgibt.

Lost Updates

Mehrere zeitnah ausgeführte Transaktionen auf demselben Datensatz müssen über Veränderungen durch andere informiert sein.
Beispiel:
In einer Datenbank ist die Anzahl der für einen Kurs angemeldeten Studenten hinterlegt. Benutzer A und B sind Mitarbeiter, die die eingehenden Anmeldungen von Hand bearbeiten und die zusätzlichen Studenten erfassen. Benutzer A hat 10 neue Anmeldungen erhalten und ruft die bisherige Gesamtanzahl an Studenten ab. Da bisher nur 12 Studenten angemeldet sind und maximal 25 Studenten den Kurs belegen dürfen, ändert er den Wert auf 22 ab und speichert ihn in der Datenbank. Gleichzeitig hat Benutzer B 8 neue Anmeldungen erhalten und zeitnah zu Benutzer A die bisherige Anmeldezahl von 12 Studenten aus der Datenbank abgerufen. Er ändert diesen Wert auf 20 und speichert ihn kurz nach Benutzer A in der Datenbank. Die Datenbank weist nun eine Gesamtanzahl von 20 Studenten für den Kurs auf, obwohl insgesamt 30 Studenten angemeldet sind.

Die Lösung für die oben aufgeführten Problemstellungen des Mehrbenutzerbetriebs ist die Nutzung von Sperren und Schedulern[2,3,4], um das Prinzip des ACID-Modells weiterhin gewährleisten zu können.

[2] [7, S. 313-340]
[3] [17, S. 495-542]
[4] [4, S. 441-470]

3.1. Mehrbenutzerbetrieb bei lesenden Zugriffen

Bei vielen Webseiten und Anwendungen steht der Mehrzahl der Benutzer nur der lesende Zugriff auf das Datenbanksystem zur Verfügung:

- Der Stundenplan einer Hochschule
- Die Sportergebnisse einer Großveranstaltung
- Das Tagesprogramm eines Fernsehsenders

Der Datenbestand hat in diesen Fällen einen rein informativen Charakter für den Benutzer, daher dürfen keinerlei Möglichkeiten zur Änderung des Datenbestandes vorhanden sein. Wäre eine solche Einschränkung nicht vorhanden, würde jeder Fußballenthusiast die Spielergebnisse seines Vereins ändern. Und da dies alle tun würden, müssten sie sich mit den dabei entstehenden Problemen im nächsten Abschnitt befassen.

Grundsätzlich stellen die lesenden Zugriffe im Mehrbenutzerbetrieb kein Problem für das Datenbanksystem dar, da jeder Benutzer einfach nur den benötigten Datensatz abruft. Eine kleine Einschränkung hierzu kann aber darin bestehen, dass nach einem Abruf genau diese Daten im Datenbestand verändert werden. Da es sich bei der Anzeige des Benutzers um eine lokale Kopie der abgerufenen Daten handelt, wird er die Änderung nicht wahrnehmen.
In den meisten Fällen ist dies auch nicht notwendig, da nur selten ein Datenupdate erfolgt und beim nächsten Datenabruf die geänderten Werte geliefert werden. So liefert der Webauftritt einer Zeitung die angezeigten Artikel aus einem Datenbanksystem. Der Leser muss die Webseite immer wieder aktualisieren, damit ihm geänderte oder neu hinzugefügte Artikel angezeigt werden.

In Einzelfällen, wie bei einem Live-Ticker, sollte eine Aktualisierung aber automatisch erfolgen. Dies kann nur durch eine wiederholte Abfrage des Datenbestandes auf neue oder geänderte Daten geschehen. Dabei ist zu beachten, dass eine ständige Anfrage an das Datenbanksystem einen erhöhten, und möglicherweise unnötigen, Informationsaustausch erzeugt und dadurch die Antwortzeiten des Servers in den Keller treibt. Bei einer schlechten Implementierung endet es in einer Art DDoS-Attacke auf den eigenen Server, wenn 500.000 Benutzer sekündlich eine neue Kontrollanfrage senden.

3.2. Mehrbenutzerbetrieb bei schreibenden Zugriffen

> "Race-Condition: Einer gewinnt, alle anderen verlieren."

Im Gegensatz zu den lesenden Zugriffen, sind die Probleme bei schreibenden Zugriffen weitaus größer. Ohne eine Kontrollfunktion werden geänderte Daten einfach ins Datenbanksystem zurückgeschrieben und dabei die dort hinterlegten Daten überschrieben ohne eine Prüfung auf Veränderung im Datenbestand des Datenbanksystems durchzuführen. Dieses

Problem wird auch als *"Race-Condition"* bezeichnet, wobei in diesem "Rennen" derjenige "gewinnt", das als letzter seine Daten ins Datenbanksystem schreibt.

Die möglichen Konflikte sollen anhand des nachfolgenden Beispiels erläutert werden:
Der Benutzer liest einen Datensatz mit den Daten [A,B,C] aus dem Datenbestand und schreibt ihn anschließend geändert wieder zurück.
In der Tabelle 3.1 sind fünf Fälle aufgeführt, die alle ein mögliches Szenario bei der Aktualisierung dieses Datensatzes darstellen.

	lokale Daten		**Datenbestand bei**		
Fall	**alt**	**neu**	**Abruf**	**Eintrag**	**Erklärung**
1	[A,B,C]	[***Z***,B,C]	[A,B,C]	[A,B,C]	Benutzer ändert **A** in ***Z***, Datenbestand hat sich nicht verändert
2	[A,B,C]	[***Z***,B,C]	[A,B,C]	[***Z***,B,C]	Benutzer ändert **A** in ***Z***, Datenbestand weist dieselbe Änderung auf
3	[A,B,C]	[***Z***,B,C]	[A,B,C]	[A,B, ***Y***]	Benutzer ändert **A** in ***Z***, im Datenbestand wurde **C** in ***Y*** geändert
4	[A,B,C]	[***Z***,B,C]	[A,B,C]	[***X***,B,C]	Benutzer ändert **A** in ***Z***, im Datenbestand wurde **A** in ***X*** geändert
5	[A,B,C]	[***Z***,B,C]	[A,B,C]		Benutzer ändert **A** in ***Z***, Datensatz im Datenbestand gelöscht

Abb. 3.1.: mögliche Update-Konstellationen im Mehrbenutzerbetrieb

Die Fälle eins und zwei

Beide Fälle sind einfach zu lösen, da die Daten im Datenbestand entweder mit dem ursprünglich abgerufenen Datensatz (Fall eins) oder dem geänderten Datensatz (Fall zwei) übereinstimmt. Im ersten Fall werden die geänderten Daten einfach ins Datenbanksystem geschrieben, während im zweiten Fall (theoretisch) auf den Speichervorgang verzichtet werden könnte, wobei eine erneute Speicherung auch möglich ist.

Die Fälle drei und vier

Der dritte und vierte Fall erfordert eine etwas genauere Betrachtung und gegebenenfalls das Eingreifen des Benutzers. Im dritten Fall gibt es zwar keinen Konflikt bei den explizit geänderten Daten, trotzdem muss entschieden werden, welche Daten schlussendlich im Datenbestand stehen sollen. Da das Datenbanksystem hierüber (ohne ein Regelwerk) keine Entscheidung treffen kann, bleiben die Möglichkeiten:

a) Eine Rückmeldung an den Benutzer, der entscheiden muss, ob sein Datensatz gespeichert werden soll und das ***Y*** im Datenbestand überschrieben oder seine Änderung verworfen wird.

b) Der Datensatz im Datenbestand wird komplett überschrieben (*"Race-Condition"*).

c) Nur das geänderte $\boldsymbol{Z}$ in den Datenbestand geschrieben, das $\boldsymbol{Y}$ bleibt im Datenbestand erhalten.

Im vierten Fall unterscheidet sich das vom Benutzer geänderte $\boldsymbol{Z}$ von dem im Datenbestand neu hinterlegten $\boldsymbol{X}$. Die möglichen Entscheidungen sind:

a) Die Rückmeldung an den Benutzer, der entscheiden muss, ob $\boldsymbol{Z}$ oder $\boldsymbol{X}$ im Datenbestand stehen soll.

b) Der Datensatz im Datenbestand wird komplett überschrieben (*"Race-Condition"*).

Der Fall fünf

Der fünfte Fall ist eigentlich eine Sondervariante von Fall vier, nur sind jetzt die Daten auf "gelöscht" geändert worden. Trotzdem muss dies völlig anders betrachtet werden, da das Löschen von Daten einen anderen Stellenwert als eine einfache Änderung besitzt und Nebeneffekte erzeugen kann. Gegebenenfalls sind auch abhängige Daten in anderen Tabellen des Datenbanksystems entfernt worden und somit würde ein erneutes Eintragen der Daten einen veränderten fehlerhaften Datenbestand erzeugen, der so von keinem Benutzer erwünscht ist. Eigentlich bliebe hier nur ein Zurückweisen der Benutzeränderung, aber es gibt zwei Möglichkeiten, die gelöschten Daten wieder herzustellen:

a) Das Datenbanksystem kann mittels eines Rollbacks auf den ursprünglichen Zustand zurückgebracht werden, um danach die Änderung abzuspeichern. Dabei können natürlich auch weitere zwischenzeitliche Änderungen zu Problemen führen.
Auch könnten Rücksetzpunkte gar nicht mehr erreichbar sein. Zusätzlich können Zugriffssperren unterschiedlicher Art ("SLOCK/XLOCK"[5]) zu unterschiedlichen Ergebnissen führen. Da es zumeist auch von den Einstellungen des Datenbanksystems abhängt ist dies generell ein recht unsicheres Verfahren.

b) Bei einer Löschung werden die Datensätze nicht wirklich aus dem Datenbestand entfernt, sondern nur durch eine Markierung (*Flag*) als gelöscht gekennzeichnet, ebenso werden alle abhängigen Datensätze mittels Trigger als "gelöscht" markiert. Diese Daten werden normalerweise nicht mehr angezeigt und werden nach einiger Zeit (z.B. bei regelmäßigen Wartungsintervallen über entsprechende Trigger) tatsächlich aus dem Datenbestand getilgt. Soll aber vor der eigentlichen Löschung der Datensatz wiederhergestellt werden, können die entsprechenden Markierungen entfernt werden.

Ob eine Wiederherstellung gelöschter Daten sinnvoll ist und in welchem Umfang dies möglich sein soll, muss im Praxisfall entschieden werden.

Um im späteren Einsatz den jeweiligen Fall identifizieren zu können, empfiehlt sich eine genau festgelegte Prüfungsreihenfolge:

[5] [4, S. 453-454]

Definition 3.1: Vorgehensweise bei Datenupdates

Eine grundsätzliche Vorgehensweise zur Identifizierung des jeweiligen Falles durchläuft die nachfolgenden Schritte:

1. *Verläuft eine Prüfung, ob der geänderte Datensatz im Datenbestand gelöscht wurde, erfolgreich, liegt Fall 5 vor.*
2. *Der Datensatz im Datenbestand wird mit dem ursprünglich abgerufenen Daten verglichen. Dies kann auch durch den Vergleich eines möglichen Zeitstempels erfolgen. Sofern eine Übereinstimmung vorliegt, handelt es sich um Fall 1.*
3. *Der Datensatz im Datenbestand wird mit den geänderten Daten verglichen. Sofern beide identisch sind, handelt es sich um Fall 2.*
4. *Bei einer Abweichung von Daten des Datensatzes im Datenbestand mit den ursprünglich abgerufenen Daten wird geprüft, ob diese auch geändert wurden. Liegt dieser Konflikt nicht vor, handelt es sich um Fall 3.*
5. *Sofern im vierten Schritt ein Konflikt aufgetreten ist, handelt es sich um Fall 4.*

Welcher Lösungsansatz bei entsprechender Identifizierung eines der in Definition 3.1 aufgeführten Fälle in der jeweiligen Situation angewandt wird, muss im Vorfeld anhand der Art und Verwendung der Daten im Datenbanksystem, sowie dem Aufgabenbereich und Art der Benutzer ermittelt werden.
Bei einem automatisierten System als Benutzer machen Rückfragen keinen Sinn, ebenso bei menschlichen Benutzern, die nicht über den entsprechenden Wissenstand oder die Entscheidungskompetenz verfügen.

4. Varianten der Rechteverwaltung

In diesem Kapitel wird die Handhabung der Rechteverwaltung auf der Datenbank- (Abschnitt 4.1) und der Anwendungsseite (Abschnitt 4.2) beschrieben. Es werden die Vor- und Nachteile beider Verfahren betrachtet und grundlegende Notwendigkeiten im Praxiseinsatz aufgeführt.
Sofern ein Datenbanksystem über eine integrierte Rechteverwaltung verfügt (wovon in diesem Kapitel ausgegangen wird), existiert standardmäßig immer (mindestens) ein Benutzerkonto (meist als Administrator oder *"Admin"* bezeichnet), dem alle Rechte eingeräumt sind. Ob darüber hinaus eine weitergehende Rechteverwaltung eingesetzt wird, ist auch von der Art und vom Einsatzbereich des Systems abhängig. Nachfolgend wird dies anhand von drei Beispielen genauer betrachtet.

Einzelplatzanwendung mit einem Benutzer

In der Theorie ist eine explizite Rechteverwaltung weder notwendig, noch sinnvoll. Es existiert nur ein Benutzer für das System, somit muss dieser alle Rechte besitzen, die das Datenbanksystem zur Verfügung stellt. Da es neben ihm keine weiteren Benutzer gibt, muss auch keinerlei Unterscheidung vorgenommen werden.
In der Praxis hängt die Einrichtung einer Rechteverwaltung vom Aufgabenbereich des Benutzers ab. Er mag zwar der einzige (aktive) Benutzer sein, jedoch besteht seine Aufgabe nur in der Datenpflege des Systems. Die Einrichtung des Datenbanksystems wurde durch einen Administrator ausgeführt. Damit der jetzige Benutzer nicht (aus Versehen) die Struktur des Datenbanksystems verändert, wurden ihm nur eingeschränkte Rechte gegeben. Insofern existieren eigentlich zwei Benutzer, der Administrator und der Anwender.

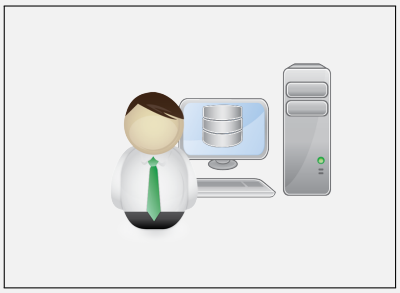


Abb. 4.1.: Einzelplatzanwendung mit einem Benutzer ohne Rechteverwaltung

Serverlösung mit mehreren Benutzern und unterschiedlichen Rechten

Wenn es Benutzer mit unterschiedlichen Rechten gibt, impliziert dies die Notwendigkeit einer Rechteverwaltung. Der Umfang der Rechte ergibt sich dabei aus der individuellen Aufgabenbeschreibung der Benutzer.
Es ist daher eine Vergabe von **individuellen Rechten** notwendig.

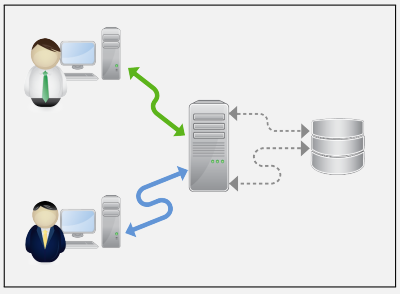


Abb. 4.2.: Serveranwendung für mehrere Benutzer mit unterschiedlichen Rechten

Serverlösung mit mehreren Benutzern und gleichen Rechten

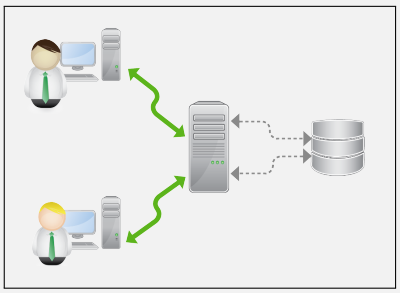


Abb. 4.3.: Serveranwendung für mehrere Benutzer mit gleichen Rechten

Grundsätzlich unterscheidet sich die Vergabe der Rechte in diesem Fall nicht von der Vorgehensweise des vorherigen Beispiels. Der Umfang der Rechte ergibt sich auch hier aus dem Aufgabenbereich der Benutzer.
Da der Umfang der Rechte für alle Benutzer gleich ausfällt, bietet sich die Erstellung einer Gruppe mit den notwendigen Rechten an. Anschließend werden die Benutzer einfach der entsprechenden Gruppe zugeordnet. Sofern eine Änderung der Rechte notwendig wird, muss dies nicht mehr bei jedem Benutzer einzeln erfolgen, sondern kann elegant über die Gruppe gepflegt werden. Die Möglichkeit zur Einrichtung solcher **Gruppenrechte** existiert in allen modernen Datenbanksystemen.

Die obigen Beispiele stellen nur eine Auswahl an möglichen praktischen Ausprägungen dar. So kann es natürlich auch Einzelplatzanwendungen mit mehreren Benutzern geben, oder auch Serversysteme auf denen die Benutzer zwar in Gruppen mit gleichartigen (Gruppen-) Rechten aufteilbar sind, trotzdem haben sie innerhalb einer Gruppe auch unterschiedliche, somit individuelle, Rechte.
Die Planung und Umsetzung der Rechteverwaltung muss dabei auf sämtlichen Ebenen des Anwendungssystems (Benutzeroberfläche, Verbindungsschnittstellen, Datenbanksystem) erfolgen, da gerade beim Einsatz verschiedener Komponenten eine Angriffsfläche an den Schnittstellen gegeben ist.

4.1. Rechteverwaltung im Datenbanksystem

Eine Rechteverwaltung auf der Ebene des Datenbanksystems ist (theoretisch) nur dann notwendig, wenn der oder die Benutzer direkt mit dem Datenbankserver und -system Verbindung aufnehmen wollen. Hierzu werden individuelle Zugänge erstellt (Abb. 4.4) und mit entsprechenden Rechte versehen. Ist ein individueller Zugang nicht notwendig, erhalten alle Benutzer einer Gruppe (Abb. 4.5) dieselben Zugangsdaten.

Diese Verfahren sind nur dort sinnvoll, wo die Benutzer direkt mit dem Datenbankserver kommunizieren, zum Beispiel in einem lokalen (Firmen-) Netzwerk. Der Benutzerkreis ist genau bekannt und alle Systeme, die Zugriff haben dürfen sind entsprechend vorkonfiguriert worden. Auch ein Zugriff von außerhalb durch VPN-Tunnel stellt kein Problem dar.

Soll das Datenbanksystem einer größeren und im Vorfeld nicht genau abgegrenzten Gruppe zur Verfügung gestellt werden, fehlen meist vorinstallierte Anwendungsprogramme auf den Rechnern der Benutzer. Diese Aufgabe wird häufig durch einen Webserver übernommen, der seinerseits die Verbindungen zum Datenbanksystem verwaltet. Auch hierbei kann zwischen der Verwaltung von individuellen (Abbildung 4.6) und Gruppenrechten (Abbildung 4.7) unterschieden werden.

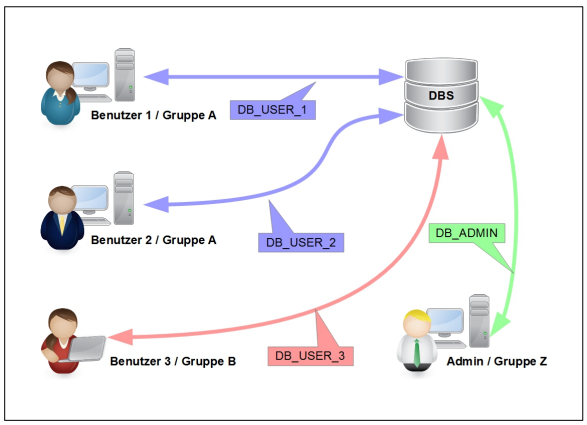


Abb. 4.4.: Direkter Datenbanksystem-Zugriff Variante 1: Benutzer erhalten eigenen Zugang mit ihren individuellen Rechten

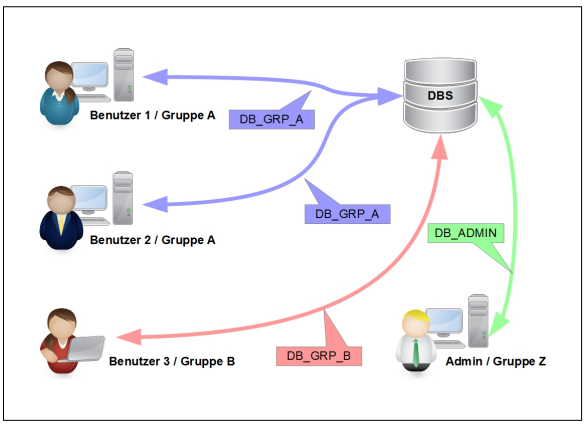


Abb. 4.5.: Direkter Datenbanksystem-Zugriff Variante 2: Benutzer erhalten gemeinsamen Zugang abhängig von ihrer Gruppenzugehörigkeit

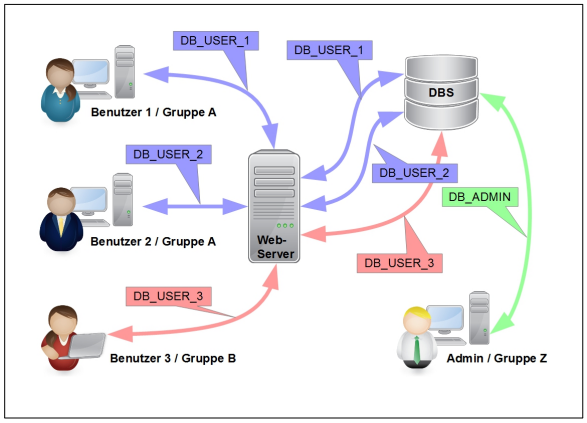


Abb. 4.6.: Indirekter Datenbanksystem-Zugriff Variante 1: Benutzer erhalten eigenen Zugang mit den individuellen Rechten, der Webserver stellt eine separate Verbindung zum Datenbanksystem her

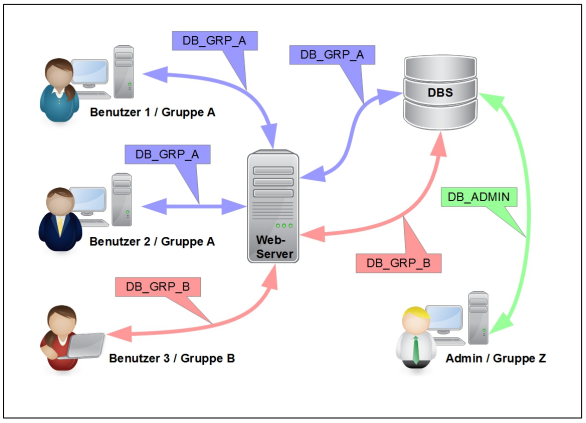


Abb. 4.7.: Indirekter Datenbanksystem-Zugriff Variante 2: Benutzer erhalten gemeinsamen Zugang abhängig von ihrer Gruppenzugehörigkeit, der Webserver erstellt eine Gruppenverbindung zum Datenbanksystem

Die Nachteile solcher, rein auf Datenbankseite, kontrollierten Rechteverwaltung sind der Zeitpunkt der Rechteprüfung und die (möglichen) Beschränkungen des Rechtesystems im Datenbanksystem.

4.1.1. Zeitpunkt der Rechteprüfung

Das vorgeschaltete Anwendungssystem (lokales Programm oder Webserver) hat keinerlei Informationen über die Rechte des Benutzers. Alle möglichen Funktionen sind daher sichtbar und erst bei der Ausführung wird durch eine Rückmeldung des Datenbanksystems klargestellt, ob das entsprechende Recht vorliegt. Für Fachanwender, die sich über ihre Rechte im Klaren sind stellt dies kein Problem dar, einem einfacher Endanwender sollten jedoch nur die ihm zustehenden Funktionalitäten angezeigt werden, um unnötiges Experimentieren zu vermeiden.

Beispiel 4.1: späte Rechteprüfung

*Eine Anzeige besteht aus den Menüpunkten **Datensatz abrufen** und **Datensatz anlegen** und ist für alle Benutzer sichtbar.*
*Ein Benutzer wählt den Menüpunkt **Datensatz anlegen** und gelangt in entsprechende Eingabemaske. Erst beim Versuch, die neuen Daten durch das Bestätigen der Maske anzulegen, erfolgt eine Prüfung durch das Datenbanksystem. Weil dieser Benutzer keine Rechte für das Anlegen von neuen Datensätzen hat, wird dieser Versuch abgewiesen.*
Sofern eine Rückmeldung erfolgt ist der Benutzer bestenfalls wegen der investierten Zeit genervt, ohne eine Rückmeldung vermutet er einen Fehler im System.
In jedem Fall verbringt er viel Zeit damit, die ihm zustehenden Funktionalitäten zu ermitteln.

Daneben kann es auch gewollt sein, dass den Endanwendern nicht alle Funktionalitäten und Möglichkeiten angezeigt werden sollen, um keine etwaige Angriffs- und Manipulationspunkte preiszugeben.

4.1.2. Beschränkungen im Datenbanksystem

Die gängigen SQL-Systeme ermöglichen eine Rechteverwaltung im Datenbanksystem auf Tabellen und deren Spalten, jedoch nicht (ohne Umweg) auf Zeilen. Dieses Manko kann zwar durch die Erstellung von Views auf Tabellen gelöst werden, bei einer entsprechend komplizierten Gemengelage an Rechten werden schnell Grenzen erreicht.

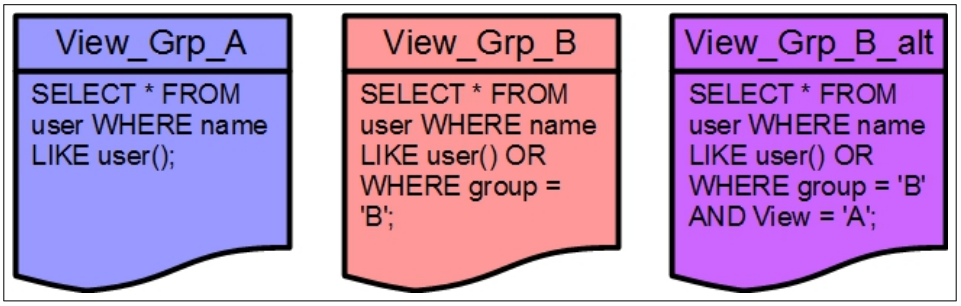


Abb. 4.8.: Viewübersicht für die Gruppen A und B (links und mitte), sowie für die Sonderrechte von Benutzer 4711 (rechts)

Obwohl die Rechteverwaltung eine Beschränkung auf einzelne Spalten einer Tabelle zulässt, ist die Funktionalität nur bedingt anwendbar. So kann in MySQL ein Benutzer mit teilweiser Zugriffsbeschränkung auf Spalten einer Tabelle keine allgemeine Anfrage in Form eines `SELECT * FROM tablename` durchführen, sondern muss die ihm zugewiesenen Spalten explizit aufrufen `SELECT row1, row2, row3 FROM tablename`. Um mögliche Änderungen zu einem späteren Zeitpunkt zu vereinfachen wird hier ebenfalls auf die Verwendung von Views zurückgegriffen.

Beispiel 4.2: Abfragebeschränkung auf Zeilen

Für ein Datenbanksystem existieren drei Benutzergruppen:

1. *Benutzer der Gruppe A dürfen nur ihre eigenen Daten aus der Tabelle USER abrufen,*
2. *Benutzer der Gruppe B hingegen dürfen ihre eigenen und die Daten der anderen Benutzer der Gruppe B abrufen,*
3. *Benutzer der Gruppe Z (Admin) dürfen alles abrufen.*

Bisher lässt sich dies noch recht einfach umsetzen, da für jede Benutzergruppe eine entsprechende View (Abb. 4.8, links und mitte) erstellt wird. Jetzt werden ein paar individuelle Ausnahmeregeln hinzugefügt:

1. *Benutzer 0815 gehört eigentlich zur Gruppe A, soll aber neben seinen Daten auch die von Benutzern der Gruppe B sehen können.*
2. *Benutzer 1138 gehört zwar zur Gruppe B, soll aber nur seine Daten sehen können.*

Auch dies kann noch mit den vorhandenen Views gelöst werden.
Soll aber der Benutzer 4711 nur die Benutzer der Gruppe B sehen können, die eben nicht alle Benutzer der Gruppe B sehen können (z.B. Benutzer 1138) muss eine weitere View (Abbildung 4.8, rechts) erstellt werden.
Sobald neben den Gruppenrechten auch abweichende individuelle Rechte, vergleichbar mit Sonderregeln, eingesetzt werden, wird die Anzahl der entsprechenden Views schnell sehr groß.

4.2. Rechteverwaltung in der Anwendungsoberfläche

Es gibt verschiedene Gründe zur Verwendung eines separaten (Web-) Servers zwischen den Systemen der Benutzer und dem Datenbankserver. Ein Grund kann die Verlagerung der Anwendungsoberfläche auf eben diesen Server sein. Es kann aber auch der Wunsch nach Trennung und Kontrolle der Kommunikation zwischen Benutzern und Datenbanksystem sein, vornehmlich um den Benutzern keinen direkten Zugang zu gewähren und dadurch Angriffe auf den Datenbankserver zu erschweren.
Ein weiterer Grund ist das (teilweise) Auslagern der Rechteverwaltung aus dem Datenbanksystem, sei es aus Performance-Gründen oder um ein verfeinertes und anwendungsbezogenes Rechtesystem zu installieren.
Sofern der Zugriff über ein zwischengeschaltetes System, wie zum Beispiel einen Webserver, erfolgt und dort bereits eine entsprechende Rechteverwaltung implementiert ist, könnte im Datenbanksystem darauf verzichtet werden. Als Zugangskennung würden dann die notwendigerweise vorhandenen Administrationsinformationen (Abbildung 4.9) dienen.

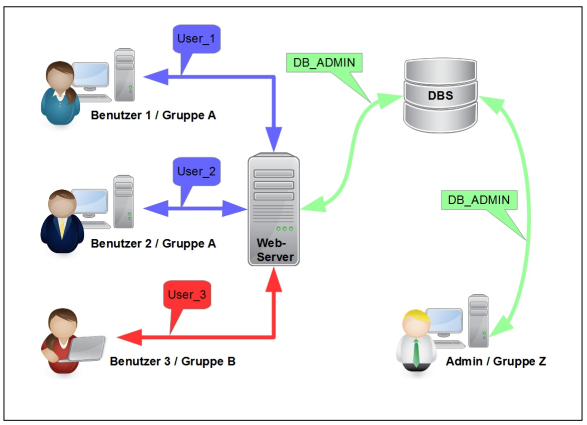


Abb. 4.9.: Der Webserver nutzt den Admin-Zugang als Verbindung zum Datenbanksystem

Die Kontrolle der Rechte in der Oberfläche ermöglicht u.a. auch die Einschränkung der Programmfunktionen. Die im Beispiel 4.1 auf Seite 15 genannten Menüpunkte ***Datensatz abrufen*** und ***Datensatz anlegen*** sind nur für die Benutzer sichtbar, die über die dafür notwendigen Rechte verfügen. Die Einschränkung auf bestimmte Zeilen und Spalten (siehe Beispiel 4.2) wird durch den Aufruf der entsprechender SELECT-Statements umgesetzt, wobei das Rechtesystem bestimmt, welches SELECT-Statement zur Anwendung kommt. Insofern entspricht dieses Verfahren dem Erzeugen von Views aus dem vorherigen Kapitel (Abbildung 4.8), mit dem Unterschied, dass nun bei einer Änderung die entsprechenden SELECT-Statements an verschiedenen Stellen im Programm-Code geändert werden muss.

Die weitaus größte Gefahr stellen fehlende oder unzureichende Prüfungen auf einzelnen Webseiten dar, sei es weil diese einfach nur vergessen wurden oder weil die Entwickler der Meinung waren, die Seite sei ohne vorherige Überprüfung gar nicht erreichbar. Bei jeder Seite, die eine Transaktion zum Datenbanksystem auslöst (gleich ob Abfrage oder Datenänderung), muss die notwendige Berechtigung des Aufrufenden geprüft werden.

Rechteverwaltung im Laravel-Framework:
Für die Benutzer müssen in einer Tabelle die allgemeinen und individuellen Rechte hinterlegt werden, dies kann von allgemeinen Gruppenrechten (*'isAdmin'*) bis zu individuellen Rechten (*'canDeleteUsers'*) gehen. Auch hier steigt mit der Komplexität der Rechtevermengung die Anzahl der Informationen (vergleichbar mit den Views auf DBS-Ebene).
Auf jeder Webseite und bei jeder Funktion und Aktion kann durch einfache Abfragen auf die notwendigen Rechte ('if (*isAdmin* ...') eine Vorselektion auf dem Webserver erfolgen. Der Benutzer auf der Client-Seite erhält nur den HTML-Seitenaufbau mit den für ihn zulässigen Elementen. Dadurch wird eine nachträgliche Manipulation des HTML-Codes erschwert, jedoch nicht gänzlich ausgeschlossen.

Um zu verhindern, dass bei einem Zugriff auf den Webserver die Admin-Zugangsdaten für das Datenbanksystem ausgelesen werden können, kann hierfür ein spezieller Datenbankbenutzer angelegt werden, der zwar über weitreichende Rechte in Bezug auf die Datenmanipulation selbst besitzt (DML-Anweisungen: *Insert*, *Update* und *Delete*), jedoch keinerlei Manipulation an der Datenbankstruktur (DDL-Anweisungen: *Create*, *Alter* und *Drop*) vornehmen kann.

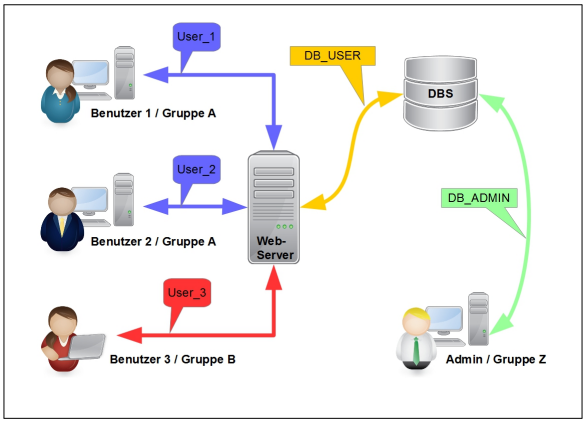


Abb. 4.10.: Der Webserver nutzt einen speziellen Datenbankzugang, der ausschließlich die Rechte aller Nutzergruppen besitzt

Vereinfacht ausgedrückt würde nur die Gesamtmenge aller Rechte erteilt, die den verschiedenen Benutzern oder Benutzergruppen zur Verfügung stehen.
Natürlich entspricht diese Beschränkung des Datenbankzugangs einer Rechteprüfung im Datenbanksystem, insofern liegt eine Überprüfung an beiden Enden vor und dies sogar mit einer gewissen Redundanz. Im nachfolgenden Abschnitt wird dieses Prinzip weiterverfolgt und ausgebaut.

4.3. Rechteverwaltung auf mehreren Ebenen

In der Praxis ist eine vereinfachte Rechteverwaltung im Datenbanksystem üblich, um eine Unterscheidung zwischen Endbenutzern und den Administratoren eines Datenbanksystems zu ermöglichen.
Da die Endbenutzer in vielen Fällen nicht direkt mit der Datenbank verbunden sind, sondern über eine Anwendungs- oder Weboberfläche Zugriff haben, wird für diese generellen Verbindungen ein Benutzerzugang im Datenbanksystem hinterlegt, der nur die notwendigen Rechte besitzt. Der Webserver ist daher nur über diesen einen Benutzerzugang mit dem Datenbanksystem verbunden (Abbildung 4.10), und alle Abfragen und Einträge werden darüber gesammelt übermittelt. Die dabei entstehende Redundanz kann durchaus gewollt sein und als zweistufiges Sicherheitssystem angesehen werden.

Um den Sicherheitsaspekt weiter zu steigern, kann der Zugang zum Datenbanksystem noch weiter gestaffelt werden. Sofern es verschiedene Gruppen von Endbenutzern gibt, können hierfür auch verschiedene Datenbankbenutzer angelegt werden. Der Webserver muss dann im Einzelfall erkennen, welcher Benutzergruppe der Anwender angehört und mit welchem Datenbankzugang (Abbildung 4.11) daraufhin gearbeitet wird.

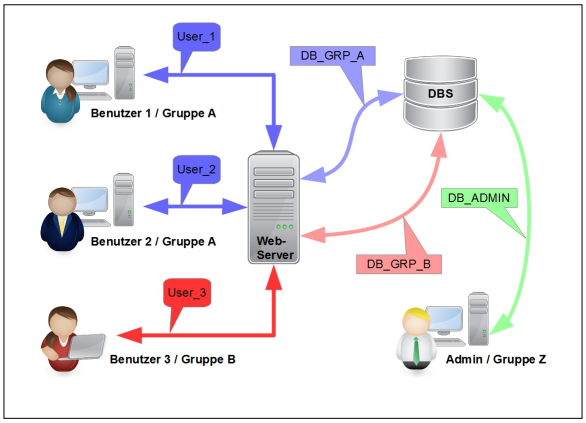


Abb. 4.11.: Benutzer 1 und 2 erhalten als Rechteinhaber der Gruppe A einen Zugriff über den Datenbankbenutzer *DB_GRP_A*, Benutzer 3, mit Rechten der Gruppe B, wird über den Datenbankbenutzer *DB_GRP_B* verbunden. Der Administrator des Systems greift direkt mit *DB_ADMIN* auf das System zu.

In einem weiteren Schritt kann sogar für jeden Benutzer ein eigener Zugang mit den entsprechenden, individuellen Rechten (Abbildung 4.12) erstellt werden.

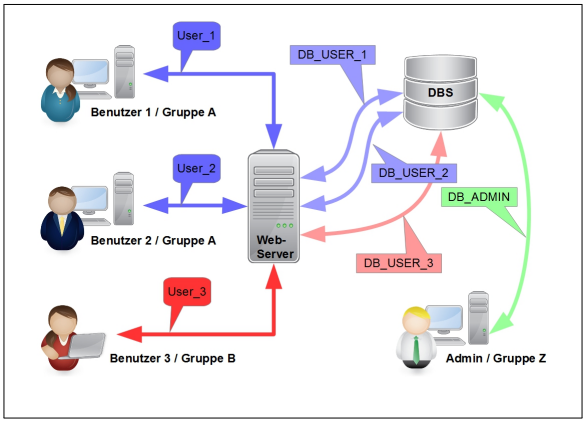


Abb. 4.12.: Der Webserver nutzt die User-Daten um einen individuellen Zugang zum Datenbanksystem zu erzeugen

Diese Redundanz der Rechteprüfung erhöht zwar mit jedem Schritt die Sicherheit in Bezug auf die Ausführung von Transaktionen, gleichzeitig muss die Konsistenz des Rechteumfangs an beiden Enden gewährleistet sein. Eine Einschränkung oder Erweiterung der Benutzerrechte muss sowohl im Datenbanksystem, wie auch in der Anwendungsoberfläche angepasst werden.
Wird dies nicht konsequent umgesetzt, hat ein Benutzer zwar in der Anwendungsoberfläche die entsprechenden Rechte, der Datenbankzugang verweigert aber den Zugriff. Umgekehrt können sogar Sicherheitslücken entstehen, wenn der Zugriff in der Anwendungsoberfläche zwar verweigert wird, die zugrunde liegende Datenbankverbindung dies aber durchaus zulassen würde. Schafft es der Benutzer, die Restriktion in der Anwendung zu umgehen, erhält er einen umfassenderen Zugriff, als ihm eigentlich zugestanden wurde.

5. Grenzen des relationalen Systems

In diesem Kapitel geht es um diese theoretischen und praktischen Grenzen relationaler Datenbanksysteme, aber auch um Beschränkungen im Praxiseinsatz.
Als Voraussetzung für diese Betrachtungen gilt die Regel, dass ein relationales Modell, bzw. System überhaupt für die anstehende Aufgabe geeignet ist. Wie aber kann dies für eine vorgesehene Aufgabe abgegrenzt werden?

Definition 5.1: Eignung für relationale Modelle

- *Relationale Datenbanken sind dort sinnvoll, wo die zu speichernden Daten einem klar erkennbaren Schemata folgen.*
- *Können die Daten in Tabellen mit aussagekräftigen Spaltennamen gepackt werden, gibt es ein relationales Modell.*

Stellt dies eine gültige und ausreichende Definition dar oder gibt es weitere Grenzen für relationale System und wenn ja, worin unterscheiden sie sich in Theorie und Praxis?
Die aufgestellten Beobachtungen und Erkenntnisse werden dabei in Bezug auf SQL-Systeme betrachtet, da diese Systeme einen Quasi-Standard im Bereich der Relationalen Datenbanksysteme stellen und oft (auch in Fachliteratur) als Synonym für relationale Modelle und Systeme angesehen werden.

5.1. Relationale Grenzen in der Theorie

> "In der Theorie gibt es praktisch keine Grenzen,..."

In diesem Abschnitt sollen kurz die theoretischen Grenzen eines relationalen Systems betrachtet werden. Die möglichen Einschränkungen durch die verfügbare Hardware wird dabei außen vorgelassen und in 5.3 *weitere Beschränkungen* genauer betrachtet.
Sofern die Definition 5.1 auf die Daten und abgeleitete Tabellenstrukturen anwendbar ist, gibt es ein relationales Modell, welches (zumindest auf dem Papier) keinerlei Grenzen kennt. Alle eingehenden Daten können in entsprechenden Tabellen abgelegt werden. Neue Informationen kommen entweder in bestehende Tabellen als neue Reihen bzw. Spalten oder erhalten gar eine eigene Tabelle.
Ab einem gewissen Komplexitätsgrad und einer großen Anzahl von Tabellen muss die Auswirkung jeder noch so kleinen Änderung genau erfasst werden, da durch mögliche Abhängigkeiten zwischen den Tabellen die Änderung auf viele Tabellen Einfluss hat.

Die Abbildungen 5.1 und 5.2 veranschaulichen die Veränderungen, die das Datenbankmodell im Verlauf des Praxisprojektes erfahren hat.

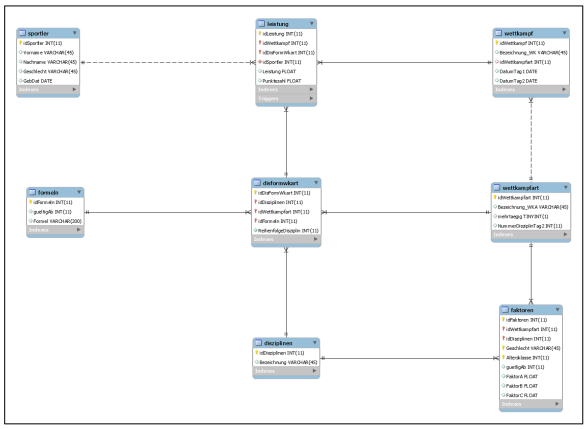


Abb. 5.1.: ER-Diagramm des Datenbankmodells zum Beginn des Praxisprojektes

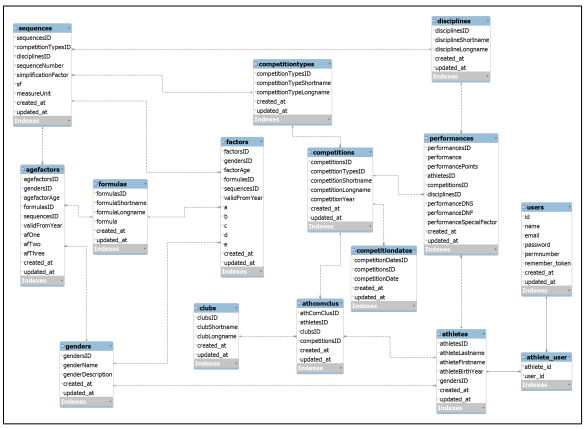


Abb. 5.2.: ER-Diagramm des Datenbankmodells zum Abschluss des Praxisprojektes

5.2. Relationale Grenzen in der Praxis

"...in der Praxis nur theoretisch nicht."

In der Praxis werden die Grenzen eines relationalen Datenbanksystems meist durch den Grad der Komplexität bestimmt.
Ein großer Online-Versandhändler, in diesem Beispiel *Ganges* genannt, vertreibt alle möglichen Arten von Artikeln. Um nun die entsprechenden weitergehenden Informationen zu einem Artikel zu speichern, seien es DVDs, Blue-Rays, Software oder Küchenausstattung, muss sehr große Artikel-Tabelle, wie in Abbildung 5.3, erstellt werden, die eine Menge an Spalten vorhält. Alternativ müssen für die Artikel-Gruppen eigene Tabellen mit den entsprechenden Beschreibungsspalten erstellt werden.

ID	Artikelname	Artikel-Art	Lauf-zeit	FSK	Betriebs-system	USK	Speicher-platz	Energie-klasse	Volu-men	Preis
1	Bambo	DVD	90 min	18	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	9,99
2	Rambi	Blue-Ray	85 min	16	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	4,99
3	Windumb X	Software	*NULL*	*NULL*	Windumb X	0	10 GB	*NULL*	*NULL*	199,99
4	Call of Modern Tomb Witcher	Software	*NULL*	*NULL*	Windumb X	18	40 GB	*NULL*	*NULL*	79,99
5	Kühlschrank	Küche	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	AAA	210 l	399,99

Abb. 5.3.: Eine **große** Artikeltabelle mit allen Daten. Sehr viele Datenfelder enthalten *NULL*-Werte, da sie nicht für die jeweiligen Artikel benötigt werden, und verbrauchen Speicherplatz. Von 50 Datenfeldern sind nur 27 mit Werten belegt.

In der Tabelle 5.3 sind sehr viele Datenfelder vorhanden deren Inhalt bei der Mehrzahl der Artikel keine Information enthält. Da aber auch solche Stellen Speicherplatz benötigen

wächst die notwendige Speichermenge mit jeder weiteren Artikelgruppe exponentiell an.

Beispiel 5.2: Unnötige Datenfelder

In einer Artikeltabelle haben die Artikel

- *3 gemeinsame Eigenschaften (somit 3 gemeinsam genutzte Spalten) und*
- *2 individuelle Eigenschaften je Artikelgruppe.*

Bei einer Anzahl von 5 Artikelgruppen und 4 Artikeln je Gruppe hat die Artikeltabelle insgesamt 260 Datenfelder, von denen jedoch 160 Felder NULL-Werte enthält (Eine Quote von 61,5%).
Bei einer Anzahl von 20 Artikelgruppen und 1.000 Artikeln je Gruppe sind es insgesamt 860.000 Datenfelder und 760.000 NULL-Werte (Eine Quote von 88,4%).

Die Menge an Datenfeldern und die Anzahl der NULL-Werte lässt sich mit den beiden nachfolgenden Formeln selbst berechnen:

- *Anzahl Datenfelder:* $A * G * (3 + 2 * G) <=> 3 * A * G + 2 * A * G^2$
- *Anzahl NULL-Werte:* $A * G * (2 * G - 2) <=> 2 * A * G^2 - 2 * A * G$

A = *Anzahl Artikel,* G = *Anzahl Artikelgruppen*

Ein alternativer Ansatz teilt die Informationen auf viele kleine Tabellen auf, mit einer Artikeltabelle, die nur die gemeinsamen Informationen (Name, Artikelart, Preis) und einen Verweis auf die Information in einer der Untertabellen enthält.
In Abbildung 5.4 ist die große Artikeltabelle in mehrere kleine Tabelle aufgeteilt worden, die durch Fremdschlüssel eine Referenzierung zueinander haben. Während in der großen Artikeltabelle von 50 Datenfeldern nur 27 mit Werten belegt waren, sind es nun 32 von 32 Datenfeldern. Die Menge an Informationen ist gestiegen (durch die Fremdschlüssel), trotzdem ist die Gesamtmenge an Datenfeldern gesunken (unnötige NULL-Werte fallen weg).

Die Komplexität des Systems nimmt dadurch zu, dass die Informationen auch noch länderspezifisch hinterlegt werden müssen oder viele Einzelinformationen durch Zuordnungstabellen eingebunden werden.
Die Verteilung der Informationen auf viele Einzeltabellen ist nicht nur unübersichtlich (zum Verständnis des Datenbankmodells), sondern führt auf lange Sicht zu Performance-Problemen, da der Aufruf eines Artikels das Zusammensuchen aus vielen einzelnen Tabellen zur Folge hat. Bei einer entsprechend großen Anfrage-Last kann dies zu teils langwierigen Antwortzeiten führen.[1]

[1] [18, S. 639-640]

Artikel-Tabelle

ID	Artikelname	Artikel-Art	Preis
1	Bambo	DVD	9,99
2	Rambi	Blue-Ray	4,99
3	Windumb X	Software	199,99
4	Call of Modern Tomb Witcher	Software	79,99
5	Kühlschrank	Küche	399,99

DVD-Tabelle

ID	Laufzeit	FSK	ID Artikel
1	90 min	18	1

Blue-Ray-Tabelle

ID	Laufzeit	FSK	ID Artikel
1	85 min	16	2

Software-Tabelle

ID	Betriebs-system	USK	Speicher-platz	ID Artikel
1	Windumb X	0	10 GB	3
2	Windumb X	18	40 GB	4

Küche-Tabelle

ID	Energie-klasse	Volumen	ID Artikel
1	A++	210 l	5

Abb. 5.4.: Mehrere **kleine** Einzeltabellen, die die Informationen zu einem Artikel verteilt aufbewahren. Mit jeder weiteren Artikel-Art entsteht eine zusätzliche Tabelle. Es existieren 32 Datenfelder, alle sind mit Werten belegt.

Ein weiteres Problem kann die nachträgliche Erweiterung der Funktionalität des Systems darstellen, wenn plötzlich Dateninformationen, die bisher nicht berücksichtigt wurden, ebenfalls abgespeichert werden sollen. Auch wenn theoretisch eine Erweiterung und Neustrukturierung des Modells möglich ist, so kann dies im realen Betrieb unter Umständen nur mit erheblichem Aufwand realisiert werden. Das Anhängen einer weiteren Spalte an eine Tabelle ist dabei noch eine leichte Übung, das Einfügen gänzlich neuer Tabellen, die gegebenenfalls auch Informationen aus bereits bestehenden Tabellen übernehmen, inklusive der entsprechenden Beziehungen kann ein gewaltiges Problem darstellen. Eine Migration der vorhandenen Daten inklusive entsprechender Testphasen nimmt viel Zeit in Anspruch.

5.3. weitere Beschränkungen

Neben den praktischen Grenzen, die das relationale Datenbanksystem von sich aus mitbringt, gibt es zwei weitere wesentliche Bereiche, die eine Beschränkung in der Umsetzung erzeugen. Einerseits sind dies Beschränkungen, die den Entwicklungsprozess bestimmen, andererseits treten Beschränkungen bei der Erweiterung des Systems für die Verarbeitung größerer Datenmengen und die Abwicklung steigender Anfragen auf.

5.3.1. Beschränkungen beim Entwicklungsprozess

Die Beschränkungen relationaler Datenbanksysteme bei der Entwicklung und Umsetzung eines Anwendungssystems können mitunter die größten Hürden darstellen.
Relationale Systeme werden anfangs durch Modelle repräsentiert, so dass zu Beginn ein entsprechendes Modell entwickelt wird, welches als Grundlage für die Konstruktion der tatsächlichen Datenbank dient.

Daraus ergibt sich das Problem, dass das Datenbanksystem in all seinen Einzelheiten vollständig entwickelt ist, bevor die ersten Daten gespeichert werden können. Es muss sogar genau überlegt werden, welche Daten im späteren Betrieb aufkommen können.
Dementsprechend muss bei der Entwicklung des Modells genau erörtert werden, welche Daten vorkommen und in welchem Kontext die Daten gespeichert werden sollen. Sollten im Nachgang Daten plötzlich anders oder in anderem Kontext, als ursprünglich gedacht, gespeichert werden, muss das ganze Modell neu entwickelt und gegebenenfalls das ganze System mit all seinen Daten neu zusammengestellt werden. Bei einem bereits laufenden System ist dies meist nur mit erheblichen Aufwand möglich, abhängig von der Komplexität des Systems.
Weitergehende Informationen und Anregungen für die Vorgehensweise zur Entwicklung von relationalen Datenbanksystemen finden sich in verschiedenen Literaturquellen[2,3].

Um die teils gravierenden Veränderungen bei einer nachträglichen Erweiterung des Systems zu verdeutlichen, werden hier beispielhaft die beiden ER-Diagramme aus Informatik- und Praxisprojekt in den Abbildungen 5.5 und 5.6 aufgeführt.

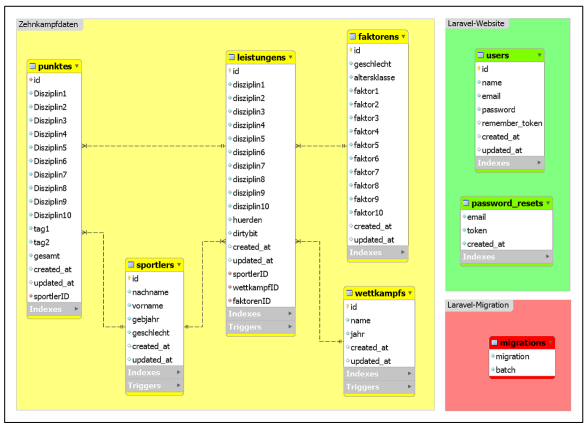


Abb. 5.5.: ER-Diagramm des Informatikprojektes, einfach strukturiert und schnell zu erfassen.

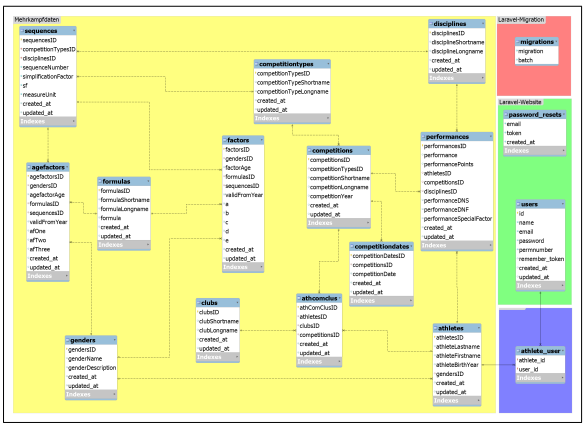


Abb. 5.6.: ER-Diagramm des Praxisprojektes, das Einbinden neuer Funktionen schlägt sich in einer größeren Anzahl an Tabellen nieder, die gesteigerte Anzahl an Verknüpfungen erschwert das Verständnis.

Obwohl das Praxisprojekt eine Erweiterung und Fortführung darstellt, musste ein komplett neues Modell realisiert werden, um den veränderten Anforderungen gerecht zu werden.

5.3.2. Beschränkungen bei nachträglicher Erweiterung / Skalierung

Relationale Datenbanken sind in ihrer Struktur und ihrem Aufbau leider nicht geeignet, um einen parallelen Betrieb auf mehreren Datenbankservern zu unterstützen. Eine Duplizität von Tabellen würde zu unterschiedlichen Informationen führen und ließe sich nur durch einen regelmäßigen Abgleich aller Datenbestände (z.B. bei einem Insert- oder Update-Vorgang) umgehen. Der dabei entstehende Datenverkehr und Prozessoraufwand würde vermutlich aber die gesteigerte Rechenkapazität des Parallelsystems wieder auffressen.

[2] [4, S. 65-194]
[3] [7, S. 129-277]

Diese Beschränkung der relationalen Systeme auf eine ausschließliche vertikale Skalierung, d.h. die Leistungssteigerung des einen Datenbankservers, macht relationale Systeme für sehr große Datenmengen (Big Data) ungeeignet.
Verschiedene Versuche zur Umgehung der Skalierungsbeschränkungen haben einige interessante Ansätze gebracht, wobei alle nachfolgend genannten in der Umsetzung andere Einschränkungen hinnehmen müssen.

5.3.2.1. VoltDB

VoltDB[4] ist eine In-Memory-Operational-Database (kurz IMOD). Bei IMODs werden sämtliche Informationen im RAM des Datenbankservers gespeichert. Dadurch wird eine minimale Zugriffszeit auf die Daten gewährleistet.
Genau genommen wird hierbei jedoch nicht die vertikale Skalierung umgangen, sondern der Leistungsgrad an die obere Grenze verschoben.

5.3.2.2. InfiniSQL

InfiniSQL[5] ist ein Ein-Mann-Projekt, das noch in der Entwicklung ist. Nach Aussagen des Entwicklers soll es voll horizontal skalierbar sein, ohne dabei das ACID-Prizip zu verletzten. Es existieren bisher noch keine großangelegten Testversuche zur Bestätigung der genannten Features.

5.3.2.3. CitusDB

CitusDB[6] verwendet das ORDBMS von PostgreSQL und wird auch von einigen (größeren) Firmen kommerziell genutzt. Die Daten werden dabei auf sogenannte Knoten verteilt, wodurch eine horizontale Skalierung ermöglicht wird. Allerdings kann bei dieser Umsetzung das ACID-Prinzip relationaler Systeme nicht mehr gewährleistet werden[7], womit es streng genommen zu den NoSQL-Systemen zählen müsste.

[4] [9]
[5] [8]
[6] [1]
[7] [19]

6. Pair-Programming

Das Pair-Programming ist eine Methodik der (kooperativen) Code-Entwicklung und ein Teilgebiet des Extreme Programming.[1] Es beschreibt eine wenig formalisierte Vorgehensweise zur Softwareentwicklung, bei der zwei Personen zusammen den Programm-Code entwickeln.
Eine Person übernimmt die Rolle des Programmierenden (*Driver*), der die entsprechenden Eingaben vornimmt. Die zweite Person sitzt als Beobachter (*Observer*) und Anleiter (*Navigator*) daneben.

Im ersten Abschnitt soll das Verfahren auf einer allgemeinen Ebene vorgestellt werden und die zugrunde liegenden Intentionen erläutert werden.
Im zweiten Abschnitt wird die praktische Anwendung des Pair-Programming anhand des umgesetzten Projektes betrachtet.

6.1. Pair-Programming in der Theorie

Die beiden Rollen sind nicht statisch vergeben, sondern sollen im Verlauf des Prozesses immer wieder gewechselt werden, um die Flexibilität der beteiligten Personen zu fördern und ein Verständnis für die Sichtweise und Argumente des Mitstreiters zu erhöhen. Der Vergleich mit den Personen in einem Fahrzeug und ihrer Funktion ist durchaus gewollt, da er den jeweiligen Aufgabenbereich darstellt.

6.1.1. Durchführung von Pair-Programming

Der *Driver* hat die "Kontrolle", da er die entsprechenden Code-Eingaben vornimmt, der *Observer* soll beobachten und durch Nachfragen und Hinweise die Aufmerksamkeit des *Drivers* hoch halten. Als *Navigator* kann er auch Anleitungen geben und hat dabei nicht so sehr die momentane Problemstellung, sondern den größeren Gesamtkontext vor Augen.
Anders als in einem Fahrzeug ist es gewollt, unterschiedliche Standpunkte und Sichtweisen zu besprechen und argumentativ zu unterlegen. Durch diesen Diskurs soll, ausgehend von verschiedenen Qualitäten der beteiligten Personen, das Gesamtergebnis besser ausfallen.
Neben dem Rollenwechsel beider beteiligten Personen innerhalb einer Aufgabe oder eines Projektes, soll die Zusammensetzung der Gruppenpaare immer wieder gewechselt werden.

[1] [20]

6.1.2. Vor- und Nachteile von Pair-Programming

Basierend auf einer von Williams und Kessler durchgeführten Studie[2] werden vier positive Effekte benannt:[3,4]

1. Pair-Pressure
   Durch die Zusammenarbeit sind beide Teilnehmer sehr viel aufmerksamer, als Teil der Gruppe fühlen sie sich viel stärker für den Erfolg oder Misserfolg mitverantwortlich. Umgangssprachlich möchte sie ihren Partner "nicht hängen lassen". Dabei wird Wert darauf gelegt, dass dieser (positive) Druck ein intrinsisches, also ein selbst auferlegtes, Verhalten ist und nicht durch äußere Einflüsse (extrinsisch) erzeugt wird.

2. Pair-Think
   Die Erfahrungen und Sichtweisen der Beteiligten erweitern den Pool an Möglichkeiten, der die Herangehensweise und Lösungsansätze für die Aufgabenstellung beschreibt. Dabei wird durch die Diskussion eine erste Qualitätsanalyse ausgeführt, bei der jeder seinen eigenen Lösungsansatz genauer überdenken muss.

3. Pair-Reviews
   Durch gegenseitige Beobachtung, Nachfrage und Analyse der Vorschläge werden bereits in der Entwicklung Fehlerquellen gefunden und behoben, das Ergebnis ist meist robuster und eleganter.

4. Pair-Learning
   Das Prinzip des "learning by doing" hat den generellen Nachteil, dass jeder im Laufe der Zeit einen Pool an Lösungsansätzen für Problemstellungen entwickelt, auf den später immer wieder zugegriffen wird. Neue Lösungsansätze werden nur selten oder bei unbekannten Problemstellungen verfolgt. Bei der Zusammenarbeit werden die Beteiligten zwangsläufig mit den Lösungsansätzen des Partners konfrontiert und finden gegebenenfalls neue Verfahren.

Bei den Studien wurden auch einige negative Einflüsse festgestellt. So gibt es gerade zu Beginn der Zusammenarbeit eine Zeitspanne, in der die Effektivität und die Vorteile des Pair-Programming noch nicht voll zur Geltung kommen. Dieser als *"Jelly-Time"* bezeichnete Zeitabschnitt muss erst vergehen, bevor die positiven Effekte greifen. Insofern sollte ein zu häufiges Wechseln der Gruppenzusammensetzung vermieden werden.
Auch die personelle Art der Zusammensetzung in den Paaren kann einen gravierenden Einfluss auf das Ergebnis haben. Die Bereitschaft zur Diskussion und Annahme eines anderen Standpunktes sind von essentieller Bedeutung für ein erfolgreiches Zusammenarbeiten.[5]
Zu guter Letzt spielt auch der Umfang des Projektes eine nicht unerhebliche Rolle. Aufgrund der notwendigen Einarbeitungszeit in der *Jelly-Time* muss das Projekt einen entsprechenden Umfang und Komplexität besitzen, um die positiven Effekte zur Geltung bringen zu können.

[2] [22]
[3] [21]
[4] [23]
[5] [16]

Inwiefern das Pair-Programming insgesamt einen signifikanten Unterschied ausmacht, konnte noch nicht abschließend festgestellt werden. Die Anzahl der Studien und der Umfang derselben scheint zumindest einen starken Einfluss zu haben. In einer Meta-Studie haben die Autoren Müller, Padberg und Tichy die zu diesem Zeitpunkt veröffentlichten Einzelstudien ausgewertet.[6] Sowohl in qualitativer und quantitativer Hinsicht lagen die dort aufgeführten Ergebnisse weit auseinander.

6.2. Pair-Programming in der Praxis

> „All I Really Need to Know about Pair Programming I Learned in Kindergarten“
>
> — Williams und Kessler

Das Eingangszitat, der Überschrift eines Fachartikels entnommen,[7] rezitiert, in abgewandelter Form, den Sachbuchtitel von Robert Fulghum aus dem Jahr 1988 und bringt die Anwendung von Pair-Programming auf den Punkt: ”Alles was für das Pair-Programming benötigt wird, wurde bereits im Kindergarten vermittelt.”

Gemäß dem Prinzip der ”guten Kinderstube”, war das Projekt nicht explizit von Beginn an nach den Vorgaben und Empfehlungen des Pair-Programming gestaltet. Trotzdem konnten die Autoren im Nachgang eine generelle Übereinstimmung zwischen ihrem Vorgehen und den Regeln des Pair-Programming feststellen.
Während in der anfänglichen Entwicklungsphase jeder seinen speziellen Fachbereich hatte, den er auch recht eigenverantwortlich voranbrachte, wurde in den späteren Ausarbeitungen und Korrekturschritten ein reger Diskurs betrieben. Dabei kam die Erkenntnis zu Tage, dass das fachliche Hintergrundwissen (oder das Fehlen desselben) den Entwicklungsprozess beschleunigen oder hemmen kann.
Die schnellen Anfangsergebnisse der Einzelarbeiten konnten danach erst langsam weiterentwickelt werden, da das jeweilige Fachwissen nachgeholt werden musste, um überhaupt eine Diskussion zu ermöglichen. Insofern sollten in großen Projekten die Partner denselben Fachschwerpunkt haben, der Wissensumfang kann dabei durchaus variieren.

Ein wichtiger Bestandteil ist die Kommunikation und Diskussion zwischen den Partnern, dadurch bedingt sollte die ”Chemie” stimmen. In der Praxis ist es manchmal schwierig, die Beiträge und Kritiken des Partners auf einer reinen Sachebene zu betrachten. Herrscht bereits im Vorfeld eine gewisse Spannung, ist eine Zusammenarbeit nur noch schwer umsetzbar.

Obwohl die gemeinschaftliche Arbeit das Hauptelement des Pair-Programming darstellt, war in einzelnen Abschnitten des Projektes eine klassische Arbeitsteilung durchaus vorteilhaft. Insbesondere in kleinen Teilaufgaben, bei denen vornehmlich das Ergebnis zählt, konnte die Arbeit durch eine einzelne Person schnell erledigt werden.

[6] [15]
[7] [21]

7. Erweiterte Datenbankfunktionen

In diesem Kapitel werden die Möglichkeiten moderner und aktiver Datenbanksysteme über die reine Datenhaltung und die damit verbundenen Funktionalitäten hinaus betrachtet. Im ersten Abschnitt wird kurz die Entwicklung und Veränderung der Datenbanksysteme und der Umfang ihrer Aufgaben beschrieben. Im zweiten Abschnitt werden die Möglichkeiten und verschiedenen Verteilungsformen kurz zusammengefasst. Im abschließenden dritten Abschnitt werden die möglichen Problematiken der unterschiedlichen Ansätze aufgezählt und an einem praktischen Fall aus der Projektentwicklung erläutert.

7.1. Die Entwicklung vom passiven zum aktiven Datenbanksystem

Ursprünglich waren Datenbanksysteme rein passiv ausgelegt und nur für das Abspeichern, Ausliefern und die Konsistenz (Prüfung von Fremdschlüsseln) der hinterlegten Daten zuständig. Obwohl einzelne Datenbanksysteme (insbesondere im Bereich der Forschung) schon vorher ein reaktiveres Verhalten ermöglichten, wurde erst 1999, mit der Einführung von Triggern in SQL3/SQL:99, das Prinzip der ECMA-Regeln[1] umgesetzt, welche um einiges älter[2] ist. Hierdurch wurde dem Datenbanksystem eine aktive Rolle zugewiesen, da nicht nur reine Konsistenzprüfungen (wie bei Fremdschlüsseln) durchgeführt wurden, sondern auch Datenmanipulation in Form von Löschungen, Änderungen und Ergänzungen möglich waren.
Zusätzlich zu Triggern bieten moderne Datenbanksysteme, neben der reinen Datenbanksprache zur Verwaltung, eine Script- oder Programmsprache, die das Erstellen von Prozeduren und Funktionen gestatten.

7.2. Die Aufteilung von Aufgaben

Durch die Erweiterung der Datenbanksysteme mit Triggern, Prozeduren, Funktionen und anderen Logikelementen lassen sich (vereinfacht) drei verschiedene Gruppen der Nutzung, hier aus Sicht des Datenbanksystems, darstellen:

1. **"über"-passives Verhalten**
   Die Verarbeitung, Veränderung und Überwachung von logischen Abhängigkeiten der Daten wird ausschließlich in der Anwendungssoftware (Programm, Weboberfläche) durchgeführt. Das Datenbanksystem hat lediglich die Aufgabe zur Datenhaltung, -annahme und -auslieferung. Die Ausprägung in dieser Gruppe kann derart auf die

[1] [4, S. 400-403]
[2] [3]

Spitze getrieben sein, dass selbst die Funktion eines Triggers oder eines Fremdschlüssels durch die Anwendungsseite erfolgt.

2. **"hyper"-aktives Verhalten**
   Die Verarbeitung, Veränderung und Überwachung von logischen Abhängigkeiten der Daten wird ausschließlich im Datenbanksystem durchgeführt. Die Anwendungssoftware übernimmt lediglich die Aufgabe eine Ein- und Ausgabemaske. Auf die Spitze getrieben würde selbst die Typenprüfung von Eingaben erst im Datenbanksystem erfolgen.

3. **"normal"-aktives Verhalten**
   Anwendungssoftware und Datenbanksystem teilen den Part des aktiven Verhaltens auf. Der Grad der Verteilung ist dabei variabel, es können durchaus auch Aufgaben von beiden Seiten übernommen werden (Redundanz). Welche Seite welchen Part einnimmt ist dabei von verschiedenen Faktoren, wie Einsatzgebiet, verwendete Anwendungs- und Datenbanksoftware und Systemaufbau, abhängig. So kann das Datenbanksystem mithilfe von Triggern und Fremdschlüsseln den inneren logischen Zusammenhang der Dateninhalte überwachen, während auf der Anwendungsseite die Validität von Informationen und Eingabeprüfungen durchgeführt wird.

Der Übergang von einer Form zur anderen ist dabei nicht ein Wechsel von Schwarz zu Weiß, sondern mehr ein feines Überfließen durch verschiedene Graustufen. Jede praktische Ausprägung lässt sich mehr oder weniger stark einer dieser drei Grundformen zuordnen.

7.3. Probleme der Aufgabenverteilung

Alle drei Gruppen haben gewisse grundlegende Vor- und Nachteile in ihrer Verwendung. Wie stark diese im Praxisbetrieb zutage treten, kann dabei stark variieren. Die nachfolgende Zusammenfassung stellt dabei kein allumfassendes Kompendium dar. Es kann durchaus weitere Punkte geben, die den Autoren in ihrer Aufstellung entgangen sind.

1. **"über"-passives Datenbanksystem**
   Ein passives System, davon ausgehend dass es zumindest über eine Fremdschlüsselverwaltung verfügt und auf einem vorliegenden ER-Modell basiert, ist schnell eingerichtet. Alle höheren Abhängigkeiten von Dateninformationen müssen von einer Anwendungssoftware geprüft und verwaltet werden.
   Wenn Benutzer über verschiedene Anwendungen mit dem Datenbanksystem interagieren, kann dieser Lösungsansatz ein Nachteil sein, da die notwendigen Prüfungssysteme in jeder Anwendung neu erstellt werden müssen. Darüber hinaus müssen Benutzer, die direkt an die Datenbank angebunden sind, bei entsprechenden Datenänderungen selbst die Prüfungen durchführen. Schlussendlich kann es für verschiedene Prüfungen notwendig sein, weitere Daten aus dem Datenbestand abzurufen, wodurch auch eine erhöhte Kommunikationsverkehr zwischen Anwendung und Datenbanksystem erzeugt wird.

2. **"hyper"-aktives Datenbanksystem**
   Sämtliche Verarbeitungsabfolgen in Bezug auf die Daten werden im Datenbanksystem

vollzogen, von außerhalb werden lediglich die 'Roh'-Daten geliefert. Der Kommunikationskanal (Anwendungsprogramm, Weboberfläche, direkter Zugriff) ist dabei egal und sämtliche logischen Zusammenhänge müssen nur im Datenbanksystem realisiert werden.

Obwohl aktive Datenbanksysteme heutzutage, neben dem Einsatz von Triggern, auch eine Möglichkeit zum Erstellen von Prozeduren und Funktionen bieten, sind sie weit entfernt vom Umfang vollwertiger Programmiersprachen. Dadurch ergeben sich zwangsläufig Einschränkungen in der Umsetzbarkeit von Anwendungsanforderungen. Diese Einschränkung kann durch Restriktionen des Datenbanksystems oder die Möglichkeiten der Programmiersprache des Datenbanksystems verursacht sein, letzten Endes ist die vornehmliche Aufgabe eines Datenbanksystems die Datenhaltung und nicht die Softwareentwicklung.
Nicht alle Verarbeitungswünsche lassen sich daher im Datenbanksystem gänzlich oder Teilen umsetzen, ein praktisches Beispiel folgt im Abschnitt 7.3.1 *Das Prepared-Statement und SQL-Injection*. Zusätzlich wird das Datenbankschema durch zu viele Erweiterungen mit mehr und mehr "Sonderregeln" durchsetzt, die alle aufeinander abgestimmt werden müssen und mit jeder neuen oder geänderten Funktionalität getestet werden müssen. Und obwohl Datenbanktrigger schon lange im SQL-Standard aufgenommen sind, ist ihre Umsetzung und Mächtigkeit von System zu System unterschiedlich ausgefallen.[3]

3. **"normal"-aktives Datenbanksystem**
   Im Datenbanksystem werden solche Aktionen gesteuert, die entweder direkt den Dateninhalt betreffen (Fremdschlüssel, Constraints) oder allgemeine und einfache Zusammenhänge prüfen sollen (Trigger).
   Komplizierte und schwer umsetzbare Funktionen werden in der Anwendungssoftware im echten Programm-Code hinterlegt, insbesondere dynamische Vorgänge.
   Die Mischform erbt gewissermaßen einige Nachteile beider Reinformen, so umgehen direkte Zugriffe auf das Datenbanksystem den Logikteil der Anwendungssoftware und bei verschiedenen Anwendungsoberflächen muss der entsprechende Logikanteil mehrfach umgesetzt werden. Einige Informationen zur Verarbeitung in der Anwendungssoftware müssen zusätzlich aus dem Datenbestand abgefragt werden, wodurch ein erhöhter Transferaufwand entsteht.
   Der Vorteil dieser Mischform ist jedoch der, dass jeder Teil (Datenbanksystem, Anwendungssoftware) seine individuellen Stärken ausspielen kann. Auch können dabei redundante Mechanismen (siehe Kapitel 3 *Mehrbenutzerbetrieb*) für mehr Sicherheit sorgen.
   Welche Seite welchen Part übernimmt muss außerdem genauestens dokumentiert werden und Übergänge müssen auf Sicherheitslücken geprüft werden.

Im Verlauf und der Umsetzung des Datenbank-Projektes musste die Entscheidung zur Aufgabenverteilung an verschiedenen Stellen getroffen werden. Die in Kapitel 4 beschriebene Verwaltung und Prüfung der Benutzerrechte war ein Punkt.

[3] [4, S. 420-429]

In diesem Abschnitt sollen die Abwägungen für oder gegen ein Verfahren anhand eines weiteren praktischen Beispiels aufgeführt werden.

7.3.1. Das Prepared-Statement und SQL-Injection

Zu Beginn der Projektarbeit war die Ausrichtung auf ein sehr aktives Datenbanksystem fokussiert. Da bereits im vorangegangenen Projekt die Berechnung der Leistungspunkte eines Sportlers durch das Datenbanksystem mittels Triggern gelöst worden war, schien es nur selbstverständlich, dies auch hier wieder zu versuchen. Da aber die Formel zur Berechnung nicht statisch für alle Disziplinen und Wettkämpfe Gültigkeit haben sollte, konnte sie nicht direkt im Trigger implementiert werden.
Der Lösungsansatz war die Hinterlegung der Formeln in einer eigenen Tabelle als SQL-Anweisung, die bei Bedarf über ein Prepared-Statement evaluiert werden konnte. Den Anstoß zur Evaluierung sollten verschiedene Trigger geben, die bei Änderungen der Grunddaten entsprechend ausgelöst werden sollten.

Im Nachfolgenden, stark verkürzten Beispiel, soll die Funktionsweise dieser Methodik veranschaulicht werden. Dabei sollen nur die entscheidenden Schritte genau betrachtet werden. Im Beispielsystem existieren Tabellen mit den entsprechenden Informationen für die Sportler, ihre Leistungen, Faktoren und Formeln. Die Relationen sind derart hergestellt, dass für die Leistung eines Sportlers die notwendige Formel und die Faktoren zur Berechnung erfragt werden können.
Für einen Sportler werden erbrachte Leistungen in der *leistung*-Tabelle abgelegt. Der Trigger *Punkteermittlung* (Code-Beispiel 7.1) überwacht die Eintragung von Leistungen in der *leistung*-Tabelle und wird gefeuert. Er sammelt die notwendigen Daten, wie die Formel und die Faktoren, und ruft seinerseits eine Prozedur *Punkteberechnung* auf und erhält nach Ausführung ein Ergebnis in der Variable `Punkte`, welches er abschließend in einem Datensatz ablegt.

```
CREATE TRIGGER ‘Punkteermittlung_AFTER_INSERT_on_Leistung‘ AFTER INSERT ON ‘leistung‘
 ↪  FOR EACH ROW
BEGIN
[...]
  DECLARE Leistung FLOAT;
  DECLARE Punkte INT;
  DECLARE Formel VARCHAR(100);
  DECLARE FaktorA FLOAT;
  DECLARE FaktorB FLOAT;
  DECLARE FaktorC FLOAT;
[...]  hier werden die notwendigen Daten (Leistung, Formel, FaktorA-C) ermittelt
  call Punkteberechnung(Leistung,Formel,FaktorA,FaktorB,FaktorC,Punkte);
[...]  das Ergebnis in Punkte wird abschließend in der Tabelle gespeichert
END
```

Code 7.1: Der Trigger *Punkteermittlung* wird ausgelöst, wenn eine neue Leistung in die *leistung*-Tabelle eingetragen wird und berechnet mit der Prozedur *Punkteberechnung* das entsprechende Ergebnis.

Die eigentliche Berechnung der Punkte wird in der Prozedur *Punkteberechnung* (Code-Beispiel 7.2) durchgeführt. Dabei werden die notwendigen Variablen als Session-Variablen einem Prepared-Statement zur Verfügung gestellt, welches einen String in Form einer vorbereitete SQL-Anfrage ausführt.
Beispielhaft steht der String

```
'@Punkte = SELECT @FaktorA * POW(@FaktorB - @Leistung, @FaktorC)'
```

für die mathematische Formel

$$@Punkte = @FaktorA * (@FaktorB - @Leistung)^{@FaktorC}$$

```
CREATE PROCEDURE `Punkteberechnung`(Leistung FLOAT, Formel VARCHAR(100), FaktorA FLOAT,
 ↪  FaktorB FLOAT, FaktorC Float, OUT Punkte FLOAT)
BEGIN
  SET @Formel = Formel;
  SET @FaktorA = FaktorA;
  SET @FaktorB = FaktorB;
  SET @FaktorC = FaktorC;
  SET @Leistung = Leistung;
  PREPARE stmt1 FROM @Formel;
  EXECUTE stmt1;
  SET Punkte = @Punkte;
  DEALLOCATE PREPARE stmt1;
END
```

Code 7.2: Die Prozedur *Punkteberechnung* führt den in *Formel* hinterlegten String als Prepared-Statement aus. Die lokalen Variablen werden vorher in Session-Variablen (gekennzeichet durch ein @) übergeben, um diese für das Prepared-Statement verfügbar zu machen.

7.3.1.1. Problemstellungen

Bei der Verwendung des Prepared-Statements ergeben sich zwei grundlegende Probleme, deren Auswirkungen teils gravierende Folgen nach sich ziehen kann.

1. MariaDB (genauso wie MySQL) erlaubt nicht die Verwendung von Prepared-Statements mit (gewollter) SQL-Injection in Triggern oder in, durch Trigger, aufgerufenen Prozeduren und Funktionen, wohingegen diese Möglichkeit, laut Dokumentation, in Oracle gegeben ist. Dieses uneinheitliches Verhalten der SQL-Datenbanksysteme birgt die Unsicherheit, dass durch einen Wechsel des Datenbanksystems oder durch ein Versionsupdate die Verwendung dieser Lösung aus dem Datenbanksystem heraus nicht mehr funktioniert.

2. Die SQL-Injection stellt eine Sicherheitslücke dar, denn, ob gewollt oder nicht, kann damit jede Art von SQL-Befehl ausgeführt werden. Da die Anlage der Formel durch die Benutzer erfolgt, könnte hier, ohne eine entsprechende Sicherheitsmaßnahme, ein SQL-Befehl zum Löschen aller Datensätze eingetragen werden.

7.3.1.2. Lösungsansätze

Die uneinheitliche Anwendung des Prepared-Statements in den verschiedenen SQL-Derivaten könnte nur dadurch umgangen werden, dass der Aufruf der Prozedur durch eine externe Quelle (in diesem Fall die Anwendungsoberfläche) erfolgt. Damit wäre jedoch ein Teil der Funktionalität (das Erkennen von Veränderungen und Auslösen der Punkteberechnung) aus dem Datenbanksystem ausgelagert.

Die Problemstellung der SQL-Injection kann nicht wirklich gänzlich gelöst werden, allenfalls können verschiedene Ansätze zur Minimierung der Sicherheitslücke verfolgt werden.
Die einfachste Abhilfe kann hierbei die Einschränkung des berechtigten Benutzerkreises für die Formelerfassung und -änderung auf besonders vertrauenswürdige Personen sein, abhängig von der Wichtigkeit und Bedeutung der hinterlegten Daten könnte dies jedoch nicht ausreichen. Auch kann es in Einzelfällen schwierig sein, den entsprechenden Kreis klein zu halten.
Alternativ könnte durch die Verwendung von Kontrollmechanismen der Inhalt auf möglichen Schad-Code geprüft werden. Abgesehen von der Problematik bei direktem Zugriff auf das Datenbanksystem können dies auch keine absolute Sicherheit garantieren, wie im folgenden Abschnitt gezeigt wird.

7.3.2. Die PHP-Funktion eval() und PHP-Injection

Da die Auswertung der Formel und anschließende Berechnung innerhalb des Datenbanksystems nicht ausreichend sicher gelöst werden kann, wird diese Aufgabe in die Anwendungsoberfläche, in diesem Fall in den PHP-Code des Webservers, ausgelagert.
Dabei wird die PHP-Funktion `eval()` verwendet, die einen übergebenen String als PHP-Code ausführt. Somit ist es möglich die gewünschte Formel in Form einer PHP-Anweisung als String im Datenbestand zu speichern und bei Notwendigkeit abzurufen und auszuführen. Das Ergebnis dieser Evaluierung wird entsprechend in den Datenbestand zurückgeschrieben. Im Gegensatz zur Lösung im vorhergehenden Abschnitt ist der notwendige PHP-Code recht kurz, wie im Beispiel 7.3 zu sehen. Zusätzlich müssten natürlich die notwendigen Logiken hinterlegt werden, die eine Veränderung erkennen und die damit einhergehende Neuberechnung auslösen. In Bezug auf die Anwendungsoberfläche ist dies jedoch schnell umgesetzt, da es in den vorhandenen Eingabe- und Änderungsmasken passiert. Einzig die direkte Eingabe ins Datenbanksystem würde keine Berechnung auslösen.

```php
<?php
 $wert;
  eval("$wert=4+10;");// $wert hat danach den wert 14
```

Code 7.3: Die beispielhafte Verwendung der PHP-Funktion `eval()`

Die dynamische Auswertung des Strings zur Laufzeit ist dem des Prepared-Statements in SQL nicht unähnlich. Dadurch ist das größte Sicherheitsproblem durchaus mit dem der SQL-Injektion vergleichbar.

7.3.2.1. Sicherheitsprobleme

Die PHP-Funktion `eval()` wird zur Laufzeit ohne weitere Überprüfung ausgeführt. Somit kann in dem auszuwertenden String jede Art von PHP-Code hinterlegt werden. Analog zum Abschnitt 7.3.1 wird die Problematik dieser Sicherheitslücke als *PHP-Injection* bezeichnet.[4] So wie bei der SQL-Injection unberechtigter Zugriff auf das Datenbanksystem möglich ist, kann durch PHP-Injection Zugriff auf Informationen des ausführenden Webservers erfolgen. Damit wäre sogar ein Umschreiben der Webseite möglich. Da die Webserver zusätzlich ihre Zugangsdaten zum Datenbankserver verwalten, können diese auch ausgelesen werden und der Benutzer könnte auf diesem Weg direkten Zugang zum Datenbankserver erhalten, gegebenenfalls mit umfassenderen Rechten als die ihm zugestandenen.

7.3.2.2. Lösungsansätze

Um die Sicherheitslücke der PHP-Injection zu schließen, können verschiedene Taktiken eingesetzt werden:

- Zugriffsbeschränkung auf vertrauenswürdigen Benutzerkreis
- vereinfachte Prüfung mittels
  - Erstellung und Verwendung einer *Whitelist*
  - Erstellung und Verwendung einer *Blacklist*
- umfassendere Prüfung mittels eines Parsers

Der Lösungsansatz der Zugriffsbeschränkung wurde bereits in Abschnitt 7.3.1 für die SQL-Injection beschrieben, die Verfahrensweisen bei der PHP-Injection sind exakt dieselben.

Eine Whitelist ist eine Liste mit erlaubten Zeichen oder Zeichenfolgen, die in dem auszuwertenden String vorkommen dürfen. Demgegenüber ist eine Blacklist eine Liste mit verbotenen Zeichen und Zeichenfolgen, deren Vorhandensein im String eine Ausführung unterbinden. Die Entscheidung für die Verwendung dieser Listen ist von verschiedenen Faktoren abhängig. Theoretisch wird die Liste eingesetzt, deren Umfang geringer ist und somit weniger Auswertungsarbeit verursacht. Im Beispieleinsatz für eine Formel werden die notwendigen Zeichen für Rechenoperationen (+, -, *, /, ^) und die Ziffern (0-9) in die Whitelist aufgenommen zusätzlich werden die Variablennamen, sofern im Vorfeld eindeutig definiert, als Zeichenfolge in der Whitelist aufgeführt. Obwohl die so erstellte Liste besser ist als eine ungeprüfte Ausführung, bleiben noch einige Probleme ungelöst.
Zwar werden durch die Whitelist die erlaubten Zeichen definiert, über ihr kontextuellen Zusammenhang werden jedoch keine Aussagen gemacht. Dies erlaubt die Eingabe von mathematisch unsinnigen, aber vollkommen zulässigen, "Formeln".
Der Beispielstring `'1++2++3+4*5**67******'` würde die Prüfung der Whitelist bestehen, aber einen Fehler in der Auswertung erzeugen. Ein weiteres Problem können mögliche Seiteneffekte der eingesetzten Programmiersprache sein, die ungewollt ausgelöst werden. Ein besonders eigenwilliges Beispiel ist die Programmiersprache JSFuck[5], die unter Verwendung

[4] [2]
[5] [10]

von sechs Zeichen vollständigen Javascript-Code erzeugt. Auch wenn den Autoren kein ähnliches Beispiel für PHP bekannt ist, schließt dies das Vorhandensein nicht gänzlich oder in Teilen aus.
Auch die ergänzende Verwendung einer Blacklist wird nicht alle Kombinationen vermeiden können und vermutlich nur die offensichtlichen Angriffspunkte abdecken.

Ein alternativer Ansatz ist die Auswertung des String mit Hilfe eines Parsers, der auch die Syntax prüft. Der Beispielstring `'1++2++3+4*5**67******'` würde dadurch aussortiert, und auch möglicher anderer Schadcode mit Nebeneffekten könnte abgewiesen werden. Inwieweit trotzdem noch Möglichkeiten zur PHP-Injection bestehen ist nur sehr schwer bestimmbar, da der auszuwertende String immer noch PHP-Code darstellt.

Anstatt den Parser zur Auswertung einer in PHP-Code geschriebenen Formel zu benutzen, bietet sich die direkte Auswertung und Evaluierung eines mathematischen Formelausdruckes an. Die Vorteile dieser Methode werden im nächsten Abschnitt genauer erläutert.

7.3.3. Ein Parser für mathematische Ausdrücke

Die Anwendung von Parsern auf mathematische Ausdrücke ist +ein umfassend behandeltes Themengebiet und auf theoretischer Ebene gut dokumentiertes Verfahren.[6,7] Ein großer Vorteil von mathematischen Ausdrücken ist, im Vergleich zu einer (Programmier-) Sprache, das klar strukturierte und relativ einfache Regelsystem. Dementsprechend ist ein Parser für mathematische Ausdrücke schnell umgesetzt.
In den folgenden Abschnitten werden auf die notwendigen Vorbereitungen, den Parse-Vorgang selbst und die Vorteile eines Parsers eingegangen. Die einzelnen Schritte des Parsens werden beispielhaft mit der Formel

$$(a + b * 10 + (4 * 10))$$

durchgeführt und kurz erläutert. Zu beachten ist, dass es verschiedene Parser-Konstruktionen gibt, die alle dieselbe Funktion erfüllen, in ihrer Durchführung jedoch unterschiedlich arbeiten. Der in diesem Projekt verwendete Parser basiert auf einem Github-Projekt[8], welcher von den Autoren, leicht modifiziert,[9] übernommen wurde.

7.3.3.1. Die lexikalische Analyse

In der lexikalischen Analyse wird die Formel in ihre einzelnen syntaktischen Bestandteile zerlegt,[10,11] im Fall von mathematischen Termen (siehe Abbildung 7.1) in atomare Objekte wie Klammern, Operatoren, Variablen und Zahlen.

[6] [12]
[7] [24]
[8] [5]
[9] [6]
[10] [12]
[11] [24]

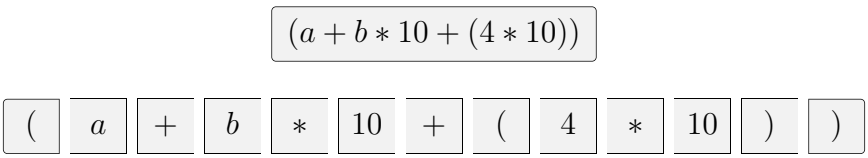


Abb. 7.1.: Die Auftrennung der Beispielformel in atomare Objekte

Ausschlaggebend für die Zerlegung sind Operatoren und Klammern, die die Zahlen und Variablen voneinander trennen. Nach der vollständigen Zerlegung des Terms kann die Auswertung durch den Parser erfolgen, alternativ könnte auch eine parallele Zerlegung und Auswertung durchgeführt werden.

7.3.3.2. Der Ablauf beim Parsen

Der Parser nimmt die in der lexikalischen Analyse erzeugten atomaren Objekte auf und erzeugt unter Verwendung des Regelwerkes einen Stack, wie in Abbildung 7.2 beispielhaft dargestellt. Während dieses Vorgangs werden die einzelnen Objekte überprüft und etwaige Fehler ausgewiesen. Nur wenn der gesamte Parse-Vorgang erfolgreich verläuft, wird der erzeugte Stack übergeben. Dieser kann nun rekursiv ausgewertet werden und liefert ein zulässiges Ergebnis.

Schritt	(Rest)-String	*Output*-Stack	*Operators*-Stack
1	(a + b * 10 + (4 * 10))		
2	a + b * 10 + (4 * 10))		(
3	+ b * 10 + (4 * 10))	a	(
4	b * 10 + (4 * 10))	a	+ (
5	* 10 + (4 * 10))	b a	+ (
6	10 + (4 * 10))	b a	* + (
7	+ (4 * 10))	10 b a	* + (
8	(4 * 10))	+ * 10 b a	+ (
9	4 * 10))	+ * 10 b a	(+ (
10	* 10))	4 + * 10 b a	(+ (
11	10))	4 + * 10 b a	* (+ (
12	))	10 4 + * 10 b a	* (+ (
13	)	* 10 4 + * 10 b a	+ (
14		+ * 10 4 + * 10 b a	

Abb. 7.2.: Hier wird die Stack-Erzeugung in *Output* durch den Parser in den einzelnen Schritten dargestellt, der *Operators*-Stack dient nur als Hilfskonstrukt. Im fertigen *Output*-Stack kann der Syntaxbaum (dargestellt in Abbildung 7.3) ausgelesen werden.

Die Erläuterung der einzelnen Schritte würde eine zu detaillierte Darstellung erfordern und den Umfang dieser Ausarbeitung sprengen. Interessierte Leser können sich aber anhand der Funktionen und Methoden des Github-Projektes[12] den Ablauf selbst erschließen.

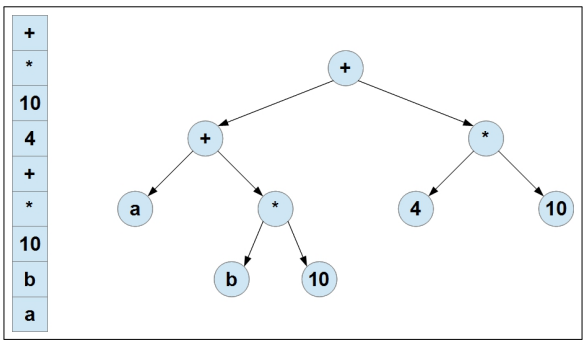


Abb. 7.3.: Basierend auf der Beispielformel $(a + b * 10 + (4 * 10))$ erzeugt der Parser einen Stack (links), der in Pre-Order-Schreibweise einen auswertbaren Syntaxbaum (rechts) ergibt. Die Reihenfolge ist dabei: Vaterknoten, rechter Kind- und linker Kindknoten. Operatoren sind immer die Wurzel eines neuen Teilbaums, Variablen und Zahlen liegen immer in den Blättern.

Der Stack selbst kann auch als Syntaxbaum (siehe Abbildung 7.3) interpretiert werden, aus dem sich die ursprüngliche Formel wieder ablesen lässt, wobei jeder Teilbaum in Klammern geschrieben werden sollte.

7.3.3.3. Vorteile des Parsens

Ein Parser interpretiert das ihm übergebene Objekt stets nach den Regeln der vorgegebenen Sprache, im Falle von mathematischen Ausdrücken sind dies die Rechenvorschriften. Mögliche Nebeneffekte können nur innerhalb dieses Regelwerkes existieren, Nebeneffekte in der Programmiersprache des Parsers sind unerheblich.
Durch dieses Konstrukt sind Sicherheitslücken insofern nur dann relevant, wie sie durch das Regelwerk erlaubt sind und einen Zugriff auf zugrundeliegende Schichten des Programms und seiner Umgebung erlauben. Außerdem kann ein Parser Sprachfehler erkennen und eine detaillierte Rückmeldung erzeugen.

[12] [6]

Anhang

A. Abbildungsverzeichnis

B. Programm-Code-Verzeichnis

C. Definitionen, Theoreme und Beispiele

D. Literatur

[3] Dittrich, Klaus R., Kotz, Angelika M. und Mülle, Jutta A. „An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases“. In: *SIGMOD Rec.* 15.3 (Sep. 1986), S. 22–36. ISSN: 0163-5808. DOI: 10.1145/15833.15836. URL: http://doi.acm.org/10.1145/15833.15836.

[4] Faeskorn-Woyke, Heide u. a. *Datenbanksysteme: Theorie und Praxis mit SQL2003, Oracle und MySQL*. IT - Informatik. OCLC: 180108625. München: Pearson Studium, 2007. 508 S. ISBN: 978-3-8273-7266-6.

[7] Geisler, Frank. *Datenbanken: Grundlagen und Design*. 5., aktualisierte und erw. Aufl. OCLC: 880390769. Heidelberg Hamburg: mitp, 2014. 550 S. ISBN: 978-3-8266-9707-4.

[11] Kleinschmidt, Peter und Rank, Christian. *Relationale Datenbanksysteme: eine praktische Einführung ; mit zahlreichen Beispielen und Übungsaufgaben ; [jetzt auf der Basis von PostgreSQL]*. 3., überarb. und erw. Aufl. OCLC: 249652289. Berlin: Springer, 2005. 271 S. ISBN: 978-3-540-22496-9.

[12] Kopp, Herbert. *Compilerbau: Grundlagen, Methoden, Werkzeuge*. Hanser-Studienbücher. OCLC: 74961403. München: Hanser, 1988. 288 S. ISBN: 978-3-446-15245-8.

[15] Müller, Matthias M., Padberg, Frank und Tichy, Walther F. „Ist XP etwas für mich? Empirische Studien zur Einschätzung von XP“. In: *Software Engineering 2005: Fachtagung des GI-Fachbereichs Softwaretechnik, 8.-11.3.2005 in Essen*. GI-Edition Proceedings 64. OCLC: 76720531. Bonn: Ges. für Informatik, 2005, S. 217–228. ISBN: 978-3-88579-393-9. URL: http://www.ipd.uka.de/Tichy/uploads/publikationen/80/se2005.pdf (besucht am 29. 09. 2016).

[16] Nosek, John T. „The Case for Collaborative Programming“. In: *Communications of the ACM* 41.3 (1. März 1998), S. 105–108. ISSN: 00010782. DOI: 10.1145/272287.272333. URL: http://portal.acm.org/citation.cfm?doid=272287.272333 (besucht am 27. 09. 2016).

[17] Saake, Gunter, Sattler, Kai-Uwe und Heuer, Andreas. *Datenbanken: Implementierungstechniken*. 3. Aufl. OCLC: 844941544. Heidelberg: mitp, 2011. 630 S. ISBN: 978-3-8266-9156-0.

[18] Saake, Gunter, Sattler, Kai-Uwe und Heuer, Andreas. *Datenbanken: Konzepte und Sprachen*. 4. Aufl. Biber-Buch. OCLC: 680676568. Heidelberg Hamburg: mitp, Verl.-Gruppe Hüthig, Jehle, Rehm, 2010. 783 S. ISBN: 978-3-8266-9057-0.

[20] Succi, Giancarlo und Marchesi, Michele. *Extreme Programming Examined*. The XP series. Boston: Addison-Wesley, 2001. 569 S. ISBN: 978-0-201-71040-3.

[21] Williams, Laurie A. und Kessler, Robert R. „All I Really Need to Know about Pair Programming I Learned in Kindergarten“. In: *Communications of the ACM* 43.5 (1. Mai 2000), S. 108–114. ISSN: 00010782. DOI: 10.1145/332833.332848. URL: http://portal.acm.org/citation.cfm?doid=332833.332848 (besucht am 27. 09. 2016).

[22] Williams, Laurie A. und Kessler, Robert R. *The Collaborative Software Process*. 1999. URL: http://www.cs.utah.edu/~lwilliam/Papers/ICSE.pdf (besucht am 29. 09. 2016).

[23] Williams, Laurie A. und Kessler, Robert R. *The Effects of "Pair-Pressure" and "Pair-Learning" on Software Engineering Education*. 2000. URL: http://www.cs.utah.edu/~lwilliam/Papers/CSEET.PDF (besucht am 29. 09. 2016).

[24] Wirth, Niklaus. *Compilerbau: eine Einführung*. 4., durchges. Aufl. Leitfäden der angewandten Mathematik und Mechanik 36. OCLC: 256130828. Stuttgart: Teubner, 1986. 118 S. ISBN: 978-3-519-32338-9.

E. Online-Quellen

[1] *Citus Data*. URL: https://www.citusdata.com/ (besucht am 15. 08. 2016).

[2] *Direct Dynamic Code Evaluation ('Eval Injection')*. URL: https://www.owasp.org/index.php/Direct_Dynamic_Code_Evaluation_('Eval_Injection') (besucht am 08. 09. 2016).

[5] Ferrara, Anthony. *Ircmaxell/Php-Math-Parser*. 21. Sep. 2011. URL: https://github.com/ircmaxell/php-math-parser (besucht am 05. 10. 2016).

[6] Frank, Matthias und Morady, Daniel. *Matthias-Frank/Php-Math-Parser*. 21. Juli 2016. URL: https://github.com/Matthias-Frank/php-math-parser (besucht am 05. 10. 2016).

[8] *InfiniSQL*. URL: http://www.infinisql.org/ (besucht am 15. 08. 2016).

[9] *In-Memory Operational Database, SQL and Scale-Out*. 29. Juli 2014. URL: https://voltdb.com/memory-operational-database-sql-and-scale-out-voltdb (besucht am 15. 08. 2016).

[10] *JSFuck - Write Any JavaScript with 6 Characters: []()!+*. URL: http://www.jsfuck.com/ (besucht am 14. 09. 2016).

[13] *Laravel - The PHP Framework For Web Artisans*. URL: https://laravel.com/ (besucht am 15. 09. 2016).

[14] *MariaDB | High Availability, Scalability and Performance beyond MySQL*. URL: https://mariadb.com/ (besucht am 15. 09. 2016).

[19] Stamper, Jason. *Citus Data Open-Sources Its Distributed PostgreSQL Analytics Database*. 21. Apr. 2016. URL: https://451research.com/report-short?entityId=88740&referrer=marketing (besucht am 18. 08. 2016).

F. Projektverteilung

Die nachfolgend aufgeführte Verteilung soll nur einen groben Überblick verschaffen. Abschnitte mit dem Vermerk ***gemeinsam*** sollen aufzeigen, das hier eine Arbeitsteilung vorliegt, die ungefähr einem Anteil um die 50% entspricht.

Projektabschnitt	**Matthias Frank**	**Daniel Morady**	
Grundstruktur Datenbank	✓	✗	
Erweiterung und Ergänzung Datenbank	✓	✓	***gemeinsam***
Grundaufbau Webseite (Framework und Darstellung)	✗	✓	
Kommunikation zur Datenbank	✗	✓	
Ergänzungen/Korrekturen an Webseite	✓	✓	***gemeinsam***
Test der Webseite	✓	✓	***gemeinsam***
- Sicherheitslücken	✓	✓	***gemeinsam***
- Funktionsaufrufe	✓	✓	***gemeinsam***
Test der DB-Funktionen	✓	✓	***gemeinsam***
Erstellung der Dokumentation	✓	✓	***gemeinsam***
- 1. Projektbeschreibung	✓	✓	***gemeinsam***
- 2. Motivation	✓	✗	
- 3. Mehrbenutzerbetrieb	✓	✓	***gemeinsam***
- 4. Rechteverwaltung	✓	✓	***gemeinsam***
- 5. Relationale Grenzen	✓	✗	
- 6. Pair-Programming	✗	✓	
- 7. Erweiterte Datenbankfunktionen	✓	✓	***gemeinsam***
- Anhang	✓	✓	***gemeinsam***