

Relationale Datenbanksysteme

Eine praktische Einführung

Dritte, überarbeitete und erweiterte Auflage

Peter Kleinschmidt · Christian Rank

Relationale Datenbanksysteme

Eine praktische Einführung

Mit zahlreichen Beispielen
und Übungsaufgaben

Dritte, überarbeitete und erweiterte Auflage

Mit 133 Abbildungen

 Springer

Prof. Dr. Peter Kleinschmidt
Universität Passau
Lehrstuhl für Wirtschaftsinformatik I
Innstraße 29
94032 Passau
kleinsch@winf.uni-passau.de

Dr. Christian Rank
Universität Passau
Rechenzentrum
Innstraße 33
94032 Passau
christian.rank@rz.uni-passau.de

ISBN 3-540-22496-3 Springer Berlin Heidelberg New York
ISBN 3-540-42413-X 2. Auflage Springer Berlin Heidelberg New York

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie;
detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

Dieses Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, des Vortrags, der Entnahme von Abbildungen und Tabellen, der Funksendung, der Mikroverfilmung oder der Vervielfältigung auf anderen Wegen und der Speicherung in Datenverarbeitungsanlagen, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Eine Vervielfältigung dieses Werkes oder von Teilen dieses Werkes ist auch im Einzelfall nur in den Grenzen der gesetzlichen Bestimmungen des Urheberrechtsgesetzes der Bundesrepublik Deutschland vom 9. September 1965 in der jeweils geltenden Fassung zulässig. Sie ist grundsätzlich vergütungspflichtig. Zuwiderhandlungen unterliegen den Strafbestimmungen des Urheberrechtsgesetzes.

Springer ist ein Unternehmen von Springer Science+Business Media
springer.de

© Springer-Verlag Berlin Heidelberg 1997, 2002, 2005
Printed in Germany

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Umschlaggestaltung: design & production GmbH, Heidelberg

SPIN 11303046

42/3130-5 4 3 2 1 0 – Gedruckt auf säurefreiem Papier

Vorwort

Die dateiorientierte Datenhaltung wird heute in betrieblichen Bereichen weitgehend abgelöst durch Datenbankmanagementsysteme (DBMS), vor allem durch die relational orientierten Systeme.

Die Literatur zu diesem Gebiet besteht zum einen aus stark grundlagenorientierten Werken, die die Prinzipien des Datenbank-Entwurfs und der Implementierung von DBMS behandeln. Zum anderen findet man sehr detailreiche Beschreibungen mit Manualcharakter, die die technischen Aspekte des Umgangs mit DBMS beinhalten. Das vorliegende Buch ist in der Mitte dieser beiden Extreme anzusiedeln.

Wir wollen den Leser durch Vermittlung der wichtigsten Techniken möglichst schnell in die Lage versetzen, eigene Datenbankapplikationen zu konzipieren und zu implementieren bzw. Verständnis für die Hintergründe bestehender Datenbankapplikationen zu gewinnen. Durch die weitgehend knappe Darstellung und Konzentration auf wesentliche Aspekte wollen wir dem Leser das umständliche Extrahieren des für ihn wichtigen Materials aus Manualen oder Online-Hilfen ersparen. Natürlich wird sich für spezielle Fragen der Umgang mit Manualen nicht vermeiden lassen.

Dieses Buch entstand aus einer Lehrveranstaltung über Datenbanken und Informationssysteme, die regelmäßig im Sommersemester für Studierende der Betriebs- und Volkswirtschaftslehre der Universität Passau im Vertiefungsgebiet "Wirtschaftsinformatik" stattfindet. Die Veranstaltung besteht aus einer Vorlesung mit zwei Wochenstunden und einem Praktikum mit vier Wochenstunden. Das Praktikum wird in Form von Tafelübungen und betreuten Übungen am Rechner veranstaltet. Deshalb ist das Buch geeignet als Begleitlektüre zu entsprechenden Lehrveranstaltungen an Universitäten und Fachhochschulen für wirtschaftswissenschaftliche Studiengänge im Schwerpunkt Wirtschaftsinformatik. Auch für Studierende mit Hauptfach Informatik ist es eine sinnvolle Ergänzung zu der meist wissenschaftlich weiterführenden Literatur.

Für Praktiker – vor allem solche, die sich mit dem zügigen Umdenken von dateiorientierten Anwendungssystemen zu datenbankorien-

tierten Informations- und Transaktionssystemen konfrontiert sehen – eignet es sich zum Selbststudium. Auch der Programmierer, der Datenbankkenntnisse – insbesondere in SQL – für Applikationsentwicklungssprachen benötigt, wird Nutzen aus dem Buch ziehen und sich schnell die notwendigen Kenntnisse aneignen. Technische Versiertheit im Umgang mit Informationstechnologie setzt ein “Learning by doing” anhand konkreter Beispiele voraus. Wir haben den Text deshalb stets mit kleinen Beispielen durchsetzt und bieten im Anhang eine Sammlung komplexerer Übungsbeispiele an. Diese orientieren sich alle an kleinen Datenbanken, die wir über das Internet verfügbar machen (siehe dazu Anhang C).

Das Buch beschäftigt sich ausschließlich mit dem relationalen Datenmodell. Ältere Modelle wie das Netzwerkmodell und das hierarchische Modell behandeln wir nicht, da sie keine größere Bedeutung mehr haben. Objektorientierte Konzepte finden allmählich Eingang in kommerzielle Datenbanksysteme – auch der aktuelle Standard SQL:2003 sowie sein Vorgänger SQL:1999 unterstützen diese Konzepte –, eine Behandlung würde jedoch den Rahmen dieses Buches sprengen. Da objektorientierte Modelle ebenfalls relationale Techniken verwenden, sind die Inhalte dieses Buches auch für derartige Modelle relevant.

Die Beispiele des Buches und die Sprachsyntax von SQL und der behandelten prozeduralen Erweiterung orientieren sich an dem DBMS PostgreSQL, da dieses lizenzkostenfrei für viele Plattformen verfügbar ist und nur relativ geringe Systemressourcen benötigt. Wenn möglich, bemühen wir uns allerdings um die Vermeidung PostgreSQL-spezifischer Elemente. Es hätte jedoch zu weit geführt, alle über den Standard hinausgehenden Syntaxelemente und Besonderheiten anderer Systeme aufzuführen. Die Erweiterungen von SQL oder die Gestaltung von graphischen Benutzeroberflächen sind bei vielen anderen Systemen im Prinzip verwandt, so dass auch für den Leser, der ein anderes DBMS einsetzt, die Lektüre sinnvoll ist. Da PostgreSQL frei verfügbar ist (“Open Source”), steht es dem Leser in jedem Fall offen, sich das System zu Übungszwecken zu installieren (siehe dazu Anhang B.1), ein normaler PC reicht hierzu aus.

Das erste Kapitel dieses Buches führt kurz in die Thematik der Daten-

banksysteme ein, grenzt sie von klassischen Dateisystemen ab, formuliert Forderungen an ein DBMS und seine Leistungsfähigkeit und stellt ihre allgemeine Architektur vor.

Im zweiten Kapitel werden das Relationenmodell und das Entity-Relationship-Modell beschrieben und die wichtigsten Basisoperationen auf Relationen vorgestellt. Diese Operationen werden im dritten Kapitel anhand von SQL als Datenmanipulationssprache konkret implementierbar.

Im vierten Kapitel wird der Datenbank-Entwurf über die Vermeidung von Anomalien durch Normalisierung behandelt und anschließend die Datenbankdefinition und die Formulierung von Integritätsbedingungen mit SQL eingeführt.

Das fünfte Kapitel behandelt die Aspekte eines Datenbankadministrators: Transaktionskonzepte, Probleme des Mehrbenutzerbetriebs und Zugriffsrechte sowie die Verfügbarkeit von Systeminformationen. Im sechsten Kapitel wird eine prozedurale Erweiterung von SQL mit ihren Kontrollstrukturen, dem Cursor-Konzept, der Fehlerbehandlung sowie Triggern und Funktionen beschrieben.

Für manche Applikationen reichen auch diese Erweiterungen nicht. Deshalb wird im siebten Kapitel darauf eingegangen, wie SQL-Sprachelemente in einer höheren Sprache verwendet werden können. Wir zeigen sowohl die Einbettung in C unter Verwendung von – mittlerweile in den SQL-Standard aufgenommenen – Embedded SQL-Konstrukten, als auch den Datenbankzugriff in der vor allem im Bereich der WWW-Programmierung weit verbreiteten Sprache Perl.

Für die Bereitstellung von Inhalten im World Wide Web, insbesondere bei der Realisierung von E-Commerce-Anwendungen wie z. B. Shopsystemen, sind Datenbanken mittlerweile unverzichtbar geworden. Daher erscheint es uns wichtig, in einem zusätzlichen Kapitel auf Möglichkeiten zur WWW-Integration von Datenbanken einzugehen.

Graphische Werkzeuge für Datenbankadministration und -design sind insbesondere bei komplexen Datenbankstrukturen äußerst nützlich. Wir stellen daher in einem separaten Kapitel einige dieser für PostgreSQL verfügbaren Werkzeuge vor. Die Konzepte dieser Administra-

tionswerkzeuge finden sich auch in entsprechenden Realisierungen für andere DBMS.

Literaturverzeichnis, Syntaxdiagramme zu Sprachelementen und ein Stichwortverzeichnis finden sich im Anhang. Dort haben wir außerdem eine Sammlung von umfangreichen Übungsaufgaben zusammengestellt. Diese Aufgaben sind so gestaltet, dass auch typische in der Praxis vorkommende Probleme behandelt werden.

Die weitgehende Anpassung der verwendeten SQL-Sprachkonstrukte sowie die Umstellung der Beispiele auf PostgreSQL in der vorliegenden Auflage machten eine grundlegende Überarbeitung des gesamten Textes erforderlich. Wir hoffen, dass dadurch ein noch größerer Kreis von Lesern von den in diesem Buch vorgestellten Inhalten profitieren kann. Wie schon die Manuskripte für die ersten beiden Auflagen wurde auch der vorliegende Text mit dem bewährten Textsatzsystem T_EX erstellt.

Wir bedanken uns vielmals für die intensive Unterstützung während der Schlussredaktion dieser Auflage bei Frau Julia Rank. Herrn Dr. Werner Müller, Frau Katharina Wetzel-Vandai und Frau Ruth Milewski vom Springer-Verlag danken wir herzlich für ihre konstruktive Kooperation.

Peter Kleinschmidt
Christian Rank

im Juli 2004

Inhaltsverzeichnis

1. Was sind Datenbanksysteme?	1
1.1. Unterschiede von DBMS zu klassischen Dateisystemen	1
1.2. Forderungen an ein DBMS	4
1.3. Abstraktionsebenen im DBMS	5
2. Formale Modelle für Datenbanken	7
2.1. Das Relationenmodell	7
2.2. Das Entity-Relationship-Modell	9
2.3. Vom Entity-Relationship- zum Relationenmodell	11
2.4. Wichtige Operationen auf Relationen (Tabellen)	12
2.5. Zusatzforderungen für relationale DBMS	19
3. Datenbankoperationen mit SQL	20
3.1. Anforderungen an einen Datenbank-Server	20
3.2. SQL-Standards	22
3.3. Sprachstruktur von SQL	24
3.4. Abrufen von Informationen mit SELECT	37
3.5. Virtuelle Tabellen (Views)	58
3.6. Datenmodifikationen	60
3.7. Ein- und Ausgabe von Datenbankinhalten	68
4. Datenbank-Entwurf	73
4.1. Anomalien in Datenbanken	73
4.2. Normalformen von Relationen	75
4.3. Dekomposition	82
4.4. Datenbankdefinition mit SQL	84
5. Datenbankbetrieb	96
5.1. Das Transaktionskonzept in Datenbanksystemen	96
5.2. Mehrbenutzerbetrieb	98
5.3. Transaktionen und Constraints	102
5.4. Datenschutz und Zugriffsrechte	104
5.5. Zugriff auf Datenbank-Metadaten	111
6. Modulare Erweiterungen von SQL	112
6.1. Benutzerdefinierte Funktionen	112
6.2. Funktionen mit SQL-Statements	114
6.3. PL/pgSQL	116
6.4. Trigger	135
7. Datenbankzugriff in höheren Programmiersprachen	143
7.1. Embedded SQL	143

7.2. Die DBI-Schnittstelle für Perl	149
7.3. Portabilität der Zugriffsverfahren	159
8. WWW-Integration von Datenbanken	162
8.1. Kommandozeilen- und graphische Benutzeroberflächen	162
8.2. Benutzeroberflächen im World Wide Web	165
8.3. Architektur einer WWW-Oberfläche mit CGI-Skripten	167
8.4. Sammeln von Eingaben mit Web-Formularen	168
8.5. Auswertung von Formulareingaben mit CGI-Skripten	171
8.6. Erzeugen von Formularen mit CGI-Skripten	175
8.7. Interaktion mit Web-Formularen	176
8.8. Effizienz von CGI-Skripten	185
8.9. CGI-Skripten im Mehrbenutzerbetrieb	188
8.10. Implementierung umfangreicher Anwendungen	192
9. Administrations- und Designwerkzeuge	194
9.1. Kommandozeilentools	194
9.2. Zugriff über WWW	195
9.3. Dedizierte graphische Tools	198
Literatur	202
A. Syntaxdiagramme	203
B. Software-Bezugsquellen	205
B.1. PostgreSQL	205
B.2. Linux	205
B.3. Cygwin für Windows	206
B.4. Perl und Perl-Module	206
B.5. Apache	206
B.6. Weitere relationale Open Source-Datenbanksysteme	207
C. WWW-Site für dieses Buch	208
D. Die Musterdatenbank FIRMA	209
E. Übungsaufgaben	212
E.1. Musterdatenbank THEATER	212
E.2. Musterdatenbank HOTEL	227
E.3. Musterdatenbank VHS	242
F. SQL-Kommandoreferenz	254
Abbildungsverzeichnis	255
Stichwortverzeichnis	260

1. Was sind Datenbanksysteme?

Die zunehmende Informationsflut in Industrie und Verwaltung erfordert im Zeitalter leistungsfähiger Rechner mächtige Programmsysteme, die die Speicherung und Verwaltung der Daten übernehmen.

Diese Aufgabe leisten heute meist Datenbanksysteme (genauer: Datenbankverwaltungssysteme, englisch: **D**atabase **M**anagement **S**ystems (DBMS)).

Ein DBMS setzt sich aus einer Kollektion von Daten, die auch Datenbasis oder Datenbank genannt wird, zusammen. Außerdem gehört zu einem DBMS eine Sammlung von Programmen zur Erzeugung und Verwaltung der Datenbank. Die Begriffe “Datenbank” und DBMS haben also verschiedene Bedeutung und sollten nicht verwechselt werden.

DBMS existieren für alle Plattformen vom PC bis zum Großrechner. Die Beispiele dieses Buches orientieren sich an dem DBMS PostgreSQL, das lizenzkostenfrei für viele Rechner- und Betriebssystemplattformen verfügbar ist (siehe dazu auch Anhang B.1).

1.1. Unterschiede von DBMS zu klassischen Dateisystemen

Informationssysteme bestanden früher meist aus einer Sammlung von Dateien, auf die mit Programmen zugegriffen wurde. Bei der Erstellung dieser Programme (häufig in COBOL geschrieben) sind jeweils eigene Operationen zum Lesen und Schreiben von Daten zu implementieren. Die Dateiverwaltung muss ebenfalls von diesen Programmen übernommen werden. Insbesondere macht es Probleme, innerhalb eines Mehrbenutzerbetriebs die Konsistenz der Daten sicherzustellen. So muss vom Programm z. B. sichergestellt werden, dass die Daten, die ein Benutzer A liest und als korrekt annimmt, erst dann von einem Benutzer B geändert werden können, wenn die Leseoperation von Benutzer A beendet ist. Der damit verbundene Programmieraufwand ist sehr hoch und für Programme, die ein DBMS nutzen, überflüssig, da dieses die Transaktionskontrolle und Dateiverwaltung übernimmt.

Bei der Datenspeicherung in isolierten Dateien werden identische Daten häufig mehrfach für verschiedene Anwendungen bereitgehalten, was zu erheblicher Redundanz führen kann. Außerdem ist sicherzustellen, dass Datenänderungen, die ein Anwendungsprogramm ausführt, auch bei den Kopien erfolgen, damit Inkonsistenzen vermieden werden.

Wir wollen dies an einem Beispiel illustrieren: Ein Versicherungsunternehmen bietet Haftpflicht- und Rechtsschutzversicherungen an. Zur Verwaltung der Haftpflicht- bzw. Rechtsschutzversicherten existieren jeweils separate Anwendungsprogramme, die u. a. eigene Adressdateien für ihren Versichertenstamm verwenden. Die Marketing-Abteilung verwendet mit ihrem eigenen Programm diese Adressdateien für Werbeaktionen. Abb. 1 zeigt schematisch den Zugriff der Programme auf diese Daten.

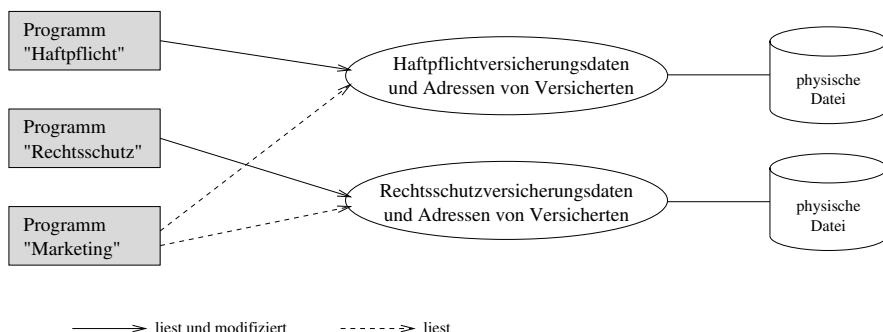


Abb. 1: Zugriff auf Daten im klassischen Dateisystem

Ändert ein Mitarbeiter der Haftpflichtabteilung die Adressdaten eines Haftpflichtversicherten, der auch rechtsschutzversichert ist, so ist evtl. dafür Sorge zu tragen, dass auch bei den Rechtsschutzdaten die Adresse geändert wird. Stimmen die Adressdaten nicht überein, so hat die Marketing-Abteilung Zugriff auf zwei verschiedene Adressen desselben Kunden. Die Adressen unterscheiden sich evtl. nur durch kleine orthographische Merkmale. Dies führt z. B. dazu, dass der Kunde zwei Werbesendungen erhält. (Jeder Postbezieher hat vermutlich diesen Effekt schon erlebt.)

Im DBMS erhalten die verschiedenen Programme keinen Zugriff auf die physischen (d. h. die auf der Festplatte abgelegten) Dateien. Die Daten befinden sich in nicht-redundanter Form in der physischen Datenbank, die durch das DBMS vor dem Anwender gekapselt wird.

Für die verschiedenen Anwendungsprogramme stellt das DBMS die benötigten logischen Dateien zur Verfügung und übernimmt die konsistente Verknüpfung der logischen Dateien mit den physischen Datenbeständen (Abb. 2).

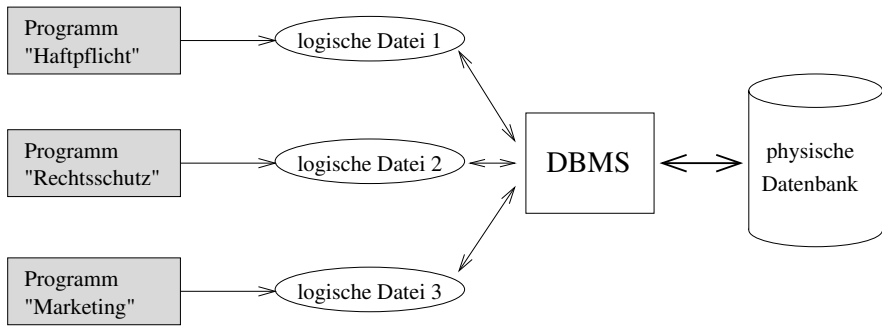


Abb. 2: Zugriff auf Daten durch ein DBMS

Ändert das Haftpflchtprogramm eine Adresse in der logischen Datei 1, so ändert sich diese Adresse automatisch auch in der logischen Datei 2 in derselben Weise. Physisch ist sie nur einmal in der Datenbank enthalten und die Marketing-Abteilung greift in der logischen Datei 3 auf ein und dieselbe Adresse zu.

Fassen wir die Nachteile der Dateisysteme zusammen:

- Doppelte Datenhaltung,
- Inkonsistenz durch unkontrollierbare Änderungen,
- hoher Programmieraufwand für Lese- und Schreiboperationen.

Als Vorteile der DBMS ergeben sich:

- Daten werden nur einmal gespeichert,
- Konsistenzprüfung und Datensicherheit werden vom DBMS gesteuert,
- weniger Programmieraufwand, da die Datenbankzugriffe vom DBMS organisiert werden.

Als weitere Unterschiede zwischen den beiden Vorgehensweisen sind zu nennen:

- Die Datenbank enthält Informationen über die Struktur der Datenbank selbst (Metadaten).
- Die Programmier Ebene und die Datenebene sind bei DBMS-Applikationen klarer getrennt.
- Der Benutzer einer Datenbank muss nicht wissen, wie und wo die Daten gespeichert sind (Datenabstraktion).

Das Modell einer Datenbank-Umgebung ist in Abb. 3 schematisch dargestellt.

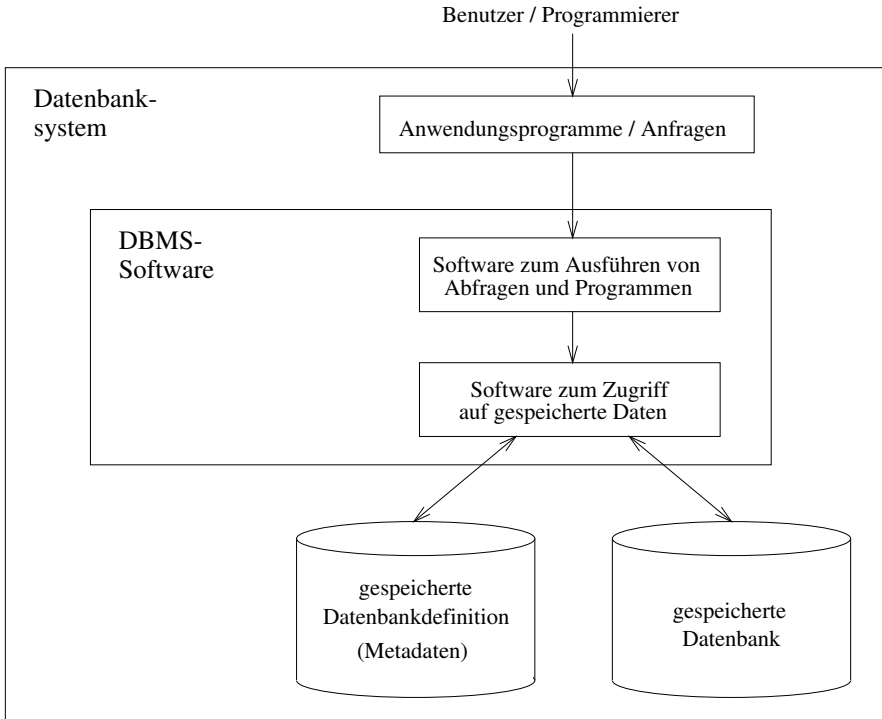


Abb. 3: Vereinfachtes Modell einer Datenbank-Umgebung

1.2. Forderungen an ein DBMS

Damit ein DBMS wirklich die genannten Vorteile bietet, sind die folgenden Forderungen zu stellen:

- Daten müssen manipulierbar sein.
- weitgehende Redundanzfreiheit (jede Information wird nur einmal gespeichert); kontrollierbare Redundanz ist jedoch manchmal notwendig und unvermeidbar.
- Universelle Verwendbarkeit, d. h. ein DBMS sollte für verschiedene Anwendungsbereiche eingesetzt werden können.
- Unabhängigkeit vom zugreifenden Programm, d. h. eine Information, die von einem Programm erzeugt wurde, kann auch von einem anderen Programm wieder gelesen werden.
- Konfigurationsunabhängigkeit des DBMS von Hardware- und Softwareumgebung sowie Netztopologie.
- Funktionale Integration: semantische Datenzusammenhänge sollen so dargestellt werden, dass sie transparent und nutzbar sind.

- Strukturflexibilität: Die Struktur der Daten sollte veränderbar sein (man denke etwa an die Umstellung von Postleitzahl- oder Währungssystemen).
- Mehrbenutzerbetrieb: gleichzeitiger Zugriff mehrerer Benutzer auf die Datenbank.
- Zugangssicherungs- und Datenschutzaufgaben sollen vom DBMS unterstützt werden.
- Gewährleistung von Datenintegrität: Die Daten sollen vollständig und semantisch korrekt sein.
- Datensicherheit: Sicherheitsspeicherung (Backups) und Rekonstruktionsverfahren (Rollforward) sollten vom DBMS unterstützt werden.

Wie bei allen komplexen Softwareprodukten werden diese Forderungen nicht sämtlich von allen DBMS erfüllt. Jedoch kommen ihnen viele Produkte schon recht nahe.

Weitere Beurteilungskriterien für die Qualität eines DBMS können für Kaufentscheidungen eine wichtige Rolle spielen:

- Verfügbarkeit für verschiedene Rechnerplattformen,
- Reaktionszeiten auf Abfragen oder Datenmanipulationen (hierzu gibt es als Richtlinie standardisierte Benchmarktests),
- Ressourcennutzung,
- Nationalsprachenunterstützung,
- verfügbare Ergänzungstools z. B. zur Dateneingabe, Abfrageausführung, Berichtserstellung, Oberflächenerstellung, Datenbank-Entwurf, Anwendungsentwicklung, Navigation in komplexen Datenbanken, Unterstützung der Systemadministration.

1.3. Abstraktionsebenen im DBMS

Um die Unabhängigkeit von Daten und Programmen zu gewährleisten, erweist sich die Verwendung eines Drei-Ebenen-Modells für Datenbankarchitekturen als hilfreich (Abb. 4).

Zweck dieses Modells ist die Trennung von Datenbankapplikationen und der (physischen) Datenbank selbst. Die einzelnen Ebenen sind wie folgt charakterisiert:

- Interne Ebene: Diese beschreibt die physikalische Struktur der Datenbank, z. B. aus welchen Dateien sie besteht. Auf die interne Ebene besteht normalerweise selbst für den Datenbankadministrator kein direkter Zugriff, sie wird allein vom DBMS verwaltet.
- Konzeptionelle Ebene: Diese beschreibt die logische Struktur der Datenbank, also etwa Tabellen, Integritätsbedingungen, Datenmo-

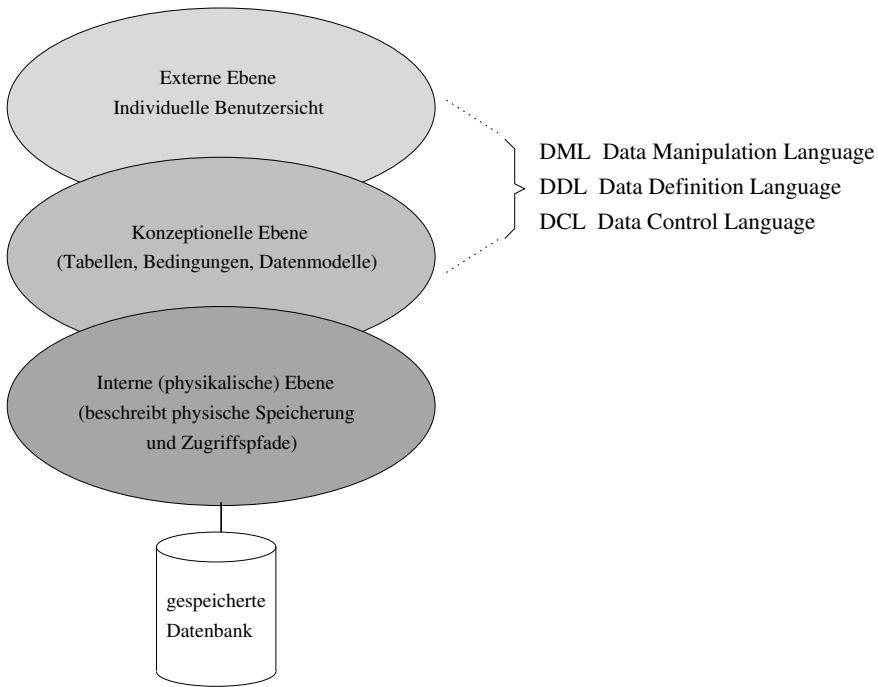


Abb. 4: Drei-Ebenen-Modell einer DBMS-Architektur

delle etc. Auf diese Ebene kann der Datenbankadministrator oder -Entwickler über eine Abfragesprache wie SQL zugreifen.

- **Externe Ebene:** Hier werden individuelle Benutzersichten auf die Datenbank definiert; insbesondere kann die zugreifbare Information für den jeweiligen Benutzer individuell festgelegt werden. Der Zugriff auf die externe Ebene erfolgt entweder in der Abfragesprache oder über graphische Benutzeroberflächen bzw. Datenbankapplikationen.

Im Rahmen dieses Buches spielen nur die konzeptionelle und die externe Ebene eine Rolle. Eine genaue Beschreibung auch der physikalischen Ebene findet sich etwa in [ElNa2004] und [KeEi2004].

2. Formale Modelle für Datenbanken

Die meisten heute eingesetzten DBMS basieren auf dem relationalen Modell, das durch E. F. Codd in dem Artikel [Co1970] begründet wurde. Das relationale Modell beruht auf dem Prinzip, dass alle Informationen in Tabellen (Relationen) abgelegt werden können. In diesem Abschnitt gehen wir auf die mathematischen Grundlagen dieses Modells ein.

2.1. Das Relationenmodell

Seien M_1, \dots, M_n beliebige Mengen (z. B. ganze Zahlen, gebrochene Zahlen, Zeichenketten, Datumsangaben).

Die Menge $M_1 \times \dots \times M_n$ bezeichnet das **kartesische Produkt** von M_1, \dots, M_n . Es besteht aus den geordneten (n -)Tupeln (m_1, \dots, m_n) , wobei $m_i \in M_i$, $1 \leq i \leq n$.

Eine **Relation** R ist eine Teilmenge

$$R \subseteq M_1 \times \dots \times M_n.$$

Die Indizes $1, \dots, n$ heißen **Attribute** der Relation; die Relation hat dann die Attributmenge $A = \{1, \dots, n\}$. Ist $(m_1, \dots, m_n) \in R$, so heißt m_i Attributwert zum Attribut i .

Statt Indizes verwendet man der besseren Übersicht wegen Namen als Attribute von Relationen.

Beispiel 2.1.: Sei M_1 die Menge aller Zeichenketten (Strings) und sei $M_2 = M_3 = M_4$ die Menge der ganzen Zahlen.

Die Relation $\text{MITARB} \subseteq M_1 \times M_2 \times M_3 \times M_4$ soll die Namen, Telefonnummern, Zimmernummern und Personalnummern der Mitarbeiter einer Firma darstellen.

Hier sei

$$\begin{aligned} \text{MITARB} = \{ & ('Müller', 4321, 130, 32077), \\ & ('Schulze', 3241, 217, 17317), \\ & ('Meier', 4938, 222, 22512), \\ & ('Schmid', 4411, 104, 21314), \\ & ('Huber', 3241, 217, 15991) \}. \end{aligned}$$

Wir bezeichnen das Attribut 1 mit NAME,
2 mit TELNR,
3 mit ZINR,
4 mit PNR.

Diese Relation kann in Tabellenform durch eine Tabelle wie in Abb. 5 dargestellt werden.

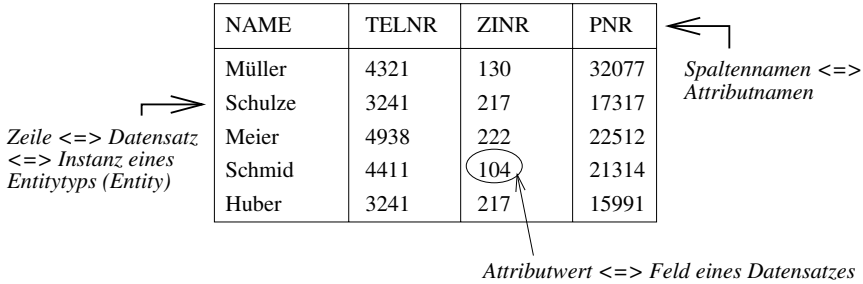


Abb. 5: Tabellendarstellung einer Relation

Wir schreiben eine Relation mit benannten Attributen (“Relationenschema”) als

MITARB(NAME,TELNR,ZINR,PNR)

Attributmenge ist hier also {NAME,TELNR,ZINR,PNR}.

Sei $R(A_1, \dots, A_n)$ ein Relationenschema und seien $X, Y \subseteq \{A_1, \dots, A_n\}$. Y ist genau dann **funktional abhängig** von X (in Zeichen $X \rightarrow Y$), wenn gleiche Attributwerte für die Komponenten von X auch gleiche Attributwerte für die Komponenten von Y erzwingen (d. h. die Attributwerte für die Komponenten von X bestimmen eindeutig die Attributwerte für die Komponenten von Y).

Y heißt genau dann **voll funktional abhängig** von X (i. Z. $X \mapsto Y$), wenn $X \rightarrow Y$ gilt, aber für jede echte Teilmenge $Z \subset X$ nicht $Z \rightarrow Y$ gilt.

Beispiel 2.2.: In der Relation MITARB gelten u. a. folgende Abhängigkeiten (bzw. Nicht-Abhängigkeiten):

- $\{PNR, ZINR\} \rightarrow \{TELNR\}$
- $\{ZINR\} \rightarrow \{TELNR\}$
- $\{ZINR\} \mapsto \{TELNR\}$
- $\{PNR, ZINR\} \not\rightarrow \{TELNR\}$

$$\begin{aligned}\{\text{TELNR}, \text{ZINR}\} &\not\rightarrow \{\text{NAME}\} \\ \{\text{ZINR}\} &\not\rightarrow \{\text{NAME}\}\end{aligned}$$

□

Funktionale Abhängigkeiten werden in der Praxis nicht durch Analyse der konkreten Relation (die sich ja ändern kann), sondern durch die beabsichtigte Interpretation der Attribute festgelegt. So gilt etwa bei der konkreten Relation MITARB die Abhängigkeit $\{\text{NAME}\} \rightarrow \{\text{PNR}\}$, die jedoch dann nicht mehr gegeben ist, wenn zwei Mitarbeiter mit gleichem Nachnamen in der Firma beschäftigt sind. Wir werden in Abschnitt 4.2 nochmals genauer auf funktionale Abhängigkeiten eingehen.

Eine Teilmenge $X \subseteq \{A_1, \dots, A_n\}$ heißt **Superschlüssel** für R , falls gilt

$$X \rightarrow \{A_1, \dots, A_n\}$$

d. h. durch einen Superschlüssel sind die Werte aller Attribute eindeutig bestimmt, also das Element der Relation eindeutig identifiziert.

X heißt **Schlüssel**, falls gilt

$$X \mapsto \{A_1, \dots, A_n\}$$

d. h. X ist minimaler Superschlüssel. Das ist gleichbedeutend damit, dass kein Attribut aus X entfernt werden kann, ohne dass die Schlüsseleigenschaft verloren geht.

Man beachte, dass jeder Schlüssel gleichzeitig auch Superschlüssel ist!

Ein **Primärschlüssel** ist *ein* beim Datenbank-Entwurf ausgezeichnete Schlüssel. Grundsätzlich kann jeder Schlüssel als Primärschlüssel ausgewählt werden. Im konkreten Fall wird man denjenigen Schlüssel verwenden, der am effizientesten eingesetzt werden kann, z. B. den Schlüssel mit den wenigsten Attributen.

Wir werden künftig in Relations- und Tabellendarstellungen die Attribute des Primärschlüssels unterstreichen.

Attribute einer Relation, die Teil eines Schlüssels sind, spielen in der Datenbanktheorie eine besondere Rolle. Wir geben ihnen daher eine spezielle Bezeichnung: Ein Attribut A einer Relation R heißt **prim**, falls A Teil (irgend)eines Schlüssels ist. Ansonsten heißt A nichtprim.

2.2. Das Entity-Relationship-Modell

Hier betrachtet man *Entities* und ihre Beziehung zueinander.

Als **Entity** bezeichnet man dabei ein existierendes Objekt, das von anderen Objekten unterscheidbar ist. Ein **Entitytyp** stellt eine Kollektion (Objekttyp) von Entities mit gleichen Merkmalen dar. Eine **Relationship** (Beziehung) ist die Assoziation mehrerer (hier: zweier) Entities nach bestimmten Gesichtspunkten. Ein **Relationship-Typ** ist schließlich ein Objekttyp von Relationships.

Die sogenannte **Komplexität** von Relationship-Typen legt man über eine Klasseneinteilung fest. Seien E_1, E_2 Entitytypen und $R \subseteq E_1 \times E_2$ Relationship-Typ. Dann gibt es folgende Klassen von Relationship-Typen (Abb. 6):

- 1:1-Beziehung: Jeder Entity in E_1 wird durch R höchstens eine Entity in E_2 zugeordnet und umgekehrt.
- 1:n-Beziehung: Jeder Entity in E_2 wird durch R höchstens eine Entity in E_1 zugeordnet (keine Beschränkung in der anderen Richtung).
- n:1-Beziehung: Jeder Entity in E_1 wird durch R höchstens eine Entity in E_2 zugeordnet (keine Beschränkung in der anderen Richtung).
- m:n-Beziehung: Beliebige Zuordnung von Entities in E_1 zu Entities in E_2 möglich.

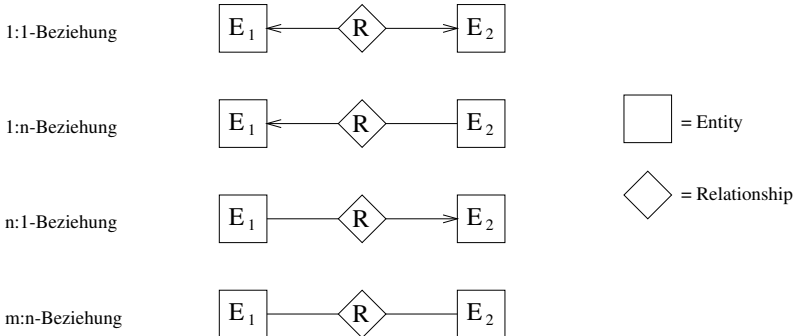


Abb. 6: Darstellung von Relationship-Typen in Entity-Relationship-Diagrammen

Beispiel 2.3.: Der Personalbestand einer Firma soll zusammen mit den Abteilungen der Firma und den Projekten, die bearbeitet werden, in einer Datenbank verwaltet werden. Die Objekttypen werden mit

den Beziehungen in einem Entity-Relationship-Diagramm dargestellt (Abb. 7).

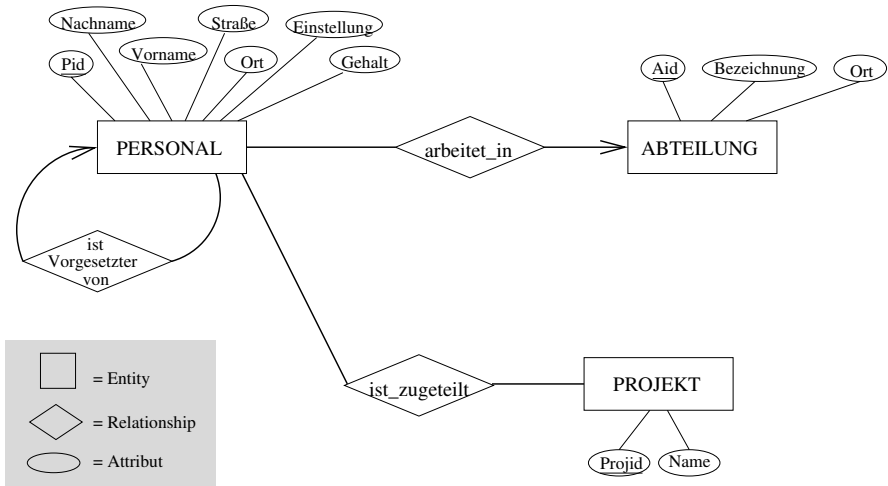


Abb. 7: Beispiel für eine durch ein Entity-Relationship-Diagramm dargestellte Datenbank

2.3. Vom Entity-Relationship- zum Relationenmodell

Jeder Entitytyp kann direkt in ein Relationenschema überführt werden, wie das folgende Beispiel zeigt.

Beispiel 2.4.: In unserer kleinen Firmendatenbank haben wir demnach folgende Relationenschemata:

PERSONAL(Pid, Nachname, Vorname, Straße, Ort
Einstellung, Gehalt)
ABTEILUNG(Aid, Bezeichnung, Ort)
PROJEKT(Projid, Name)



Jeder Relationship-Typ wird zu einem Relationenschema, in dem nur die Primärschlüssel als Attribute ("Fremdattribute") verwendet werden (bei gleichen Attributnamen muss man geeignete Umbenennungen verwenden).

Beispiel 2.5.: In der Firmendatenbank gilt:

ist_vorgesetzter_von(vorges_id,pid)
 arbeitet_in(pid,aid)
 ist_zugeteilt(pid,projid)

□

Die entstandenen Relationenschemata können noch optimiert werden: Steht etwa der Entitytyp E_1 bezüglich eines Relationship-Typs R in n:1-Beziehung mit dem Entitytyp E_2 , so können E_1 und R zu einem Relationenschema zusammengefasst werden, wobei gemeinsame Schlüsselattribute identifiziert werden.

Beispiel 2.6.: In der Firmendatenbank ergeben sich durch diese Optimierung folgende Relationenschemata:

PERSONAL(Pid,Nachname,Vorname,Straße,Ort, Einstellung,
 Gehalt,Vorges_Id,Aid)
 ABTEILUNG(Aid,Bezeichnung,Ort)
 PROJEKT(Projid,Name)
 ZUORDNUNG(Pid,Projid)

□

Optimierungen in dieser Form sind jedoch nicht immer sinnvoll, da sie zu Anomalien führen können (siehe Abschnitt 4.1).

2.4. Wichtige Operationen auf Relationen (Tabellen)

Wir betrachten die Operationen an dem Datenbank-Beispiel des letzten Abschnitts; die Relationen sollen konkret wie in Abb. 8 aussehen.

Operationen auf Relationen liefern wieder neue Relationen. Die Attributnamen der neuen Relationen können entweder aus den alten Relationen übernommen werden oder müssen neu vergeben werden.

Formal schreibt man

$$R(A_1, \dots, A_n) \leftarrow operation,$$

um eine durch *operation* entstandene Relation mit dem Namen R und den Attributnamen A_1, \dots, A_n zu benennen.

PERSONAL

pid	nachname	vorname	strasse	ort	einstellung	gehalt	vorges_id	aid
128	Meyer	Markus	Hilblestr. 17	München	19.01.1994	4327.50	107	8
205	Huber	Hilde	Passauer Str. 2a	Augsburg	27.05.1991	4995.05	57	5
107	Schmidt	Steffi	Münchner Str. 7	Freising	02.11.1990	5722.00	350	8
411	Frisch	Friedrich	Dachauer Str. 22	München	14.09.1995	4520.67	107	8
57	Klement	Karl	Kirchfeldstr. 3	Bad Tölz	04.10.1990	6011.44	350	5
350	Berger	Bernhard	Grünaustr. 11	München	28.05.1993	8748.92		1

ABTEILUNG			PROJEKT		ZUORDNUNG	
aid	bezeichnung	ort	projid	name	pid	projid
1	Betriebsleitung	München	3	Druckauftrag Fa. Karl	128	3
5	Außendienst	München	8	Beratung Fa. Seidl	411	11
8	Produktion	Olching	11	Werbung Fa. Rieger	107	11
					411	3
					205	8

Abb. 8: Konkretes Beispiel einer kleinen Datenbank (Musterdatenbank FIRMA)

2.4.1. Selektionsoperationen

Auswahl (Select):

Dies ist die Auswahl von Datensätzen einer Relation, die bestimmte Bedingungen erfüllen, i. Z. $\sigma_{\text{bedingung}}(R)$.

Beispiel 2.7.: Die Auswahl aller Datensätze der Relation PERSONAL, deren Attributwert des Attributes Aid gleich 5 ist, geschieht mit:

$$\begin{aligned} \sigma_{Aid=5}(PERSONAL) = \\ \{ (205, 'Huber', 'Hilde', 'Passauer Str. 2a', 'Augsburg', \\ '27.05.1991', 4995.05, 57, 5), \\ (57, 'Klement', 'Karl', 'Kirchfeldstr. 3', 'Bad Tölz', \\ '04.10.1990', 6011.44, 350, 5) \} \end{aligned}$$



Projektion:

Darunter verstehen wir die Auswahl bestimmter Spalten einer Tabelle (= Werte bestimmter Attribute einer Relation), i. Z. $\pi_{\text{attribut_liste}}(R)$.

Beispiel 2.8.: Auswahl der Spalten 'Ort' und 'Aid' aus der Relation

PERSONAL:

$$\begin{aligned} \pi_{Ort,Aid}(PERSONAL) = \\ \{('München',8), ('Augsburg',5), ('Freising',8), \\ ('Bad Tölz',5), ('München',1)\} \end{aligned}$$

□

Man beachte: Ergeben sich bei der Projektion identische Datensätze, so werden die Duplikate entfernt (eine Relation kann keine identischen Elemente enthalten)!

Vereinigung:

Damit ist die Zusammenfassung von zwei Relationen zu einer Relation bezeichnet. Hierbei müssen die Attributwerte jeweils paarweise derselben Grundmenge entstammen. In die Ergebnisrelation werden die Datensätze aufgenommen, die in mindestens einer der beiden Relationen vorkommen. I. Z. $R_1 \cup R_2$.

Beispiel 2.9.: Liste der Orte, die Wohnorte von Mitarbeitern oder Standorte von Abteilungen sind:

$$\begin{aligned} \pi_{Ort}(PERSONAL) \cup \pi_{Ort}(ABTEILUNG) = \\ \{('Augsburg'), ('Bad Tölz'), ('Olching'), ('Freising'), \\ ('München')\} \end{aligned}$$

□

Wie bei der Projektion werden Duplikate entfernt!

Durchschnitt:

Auch hier werden zwei Relationen zu einer Relation zusammengefasst. Die Attributwerte müssen jeweils paarweise derselben Grundmenge entstammen. In die Ergebnisrelation werden die Datensätze aufgenommen, die in beiden Relationen vorkommen. I. Z. $R_1 \cap R_2$.

Beispiel 2.10.: Liste der Orte, die Wohnorte von Mitarbeitern und Abteilungsstandorte sind:

$$\pi_{Ort}(PERSONAL) \cap \pi_{Ort}(ABTEILUNG) = \{('München')\}$$

□

Differenz:

Dies ist der dritte Operator, der zwei Relationen zu einer Relation zusammenfasst. Die Attributwerte müssen jeweils paarweise dersel-

ben Grundmenge entstammen. In die Ergebnisrelation werden die Datensätze aufgenommen, die in der ersten, aber nicht in der zweiten Relation vorkommen. I. Z. $R_1 \setminus R_2$.

Beispiel 2.11.: Liste der Abteilungsstandorte, die nicht gleichzeitig Wohnorte von Mitarbeitern sind:

$$\pi_{Ort}(ABTEILUNG) \setminus \pi_{Ort}(PERSONAL) = \{('Olching')\}$$

□

Kartesisches Produkt:

Auch dieser Operator fasst zwei Relationen zu einer zusammen. Jeder Datensatz der ersten Relation wird mit jedem Datensatz der zweiten Relation kombiniert:

$$R_1 \times R_2 :=$$

$$\{(m_1, \dots, m_k, n_1, \dots, n_l) \mid (m_1, \dots, m_k) \in R_1, (n_1, \dots, n_l) \in R_2\}$$

Verbindung (Join):

Dies ist die Zusammenfassung von zwei Relationen bezüglich bestimmter Beziehungen von Attributwerten der einen zu Attributwerten der zweiten Relation:

$$R_1 \bowtie_{\text{bedingung}} R_2 := \sigma_{\text{bedingung}}(R_1 \times R_2)$$

Sind in der Bedingung auftretende Attributnamen bei beiden Relationen gleich, so stellt man dem Attributnamen den Relationsnamen voran, damit der Name eindeutig wird, also

$$\text{relationsname.attributname}$$

Beispiel 2.12.: Verbindung der Tabelle PERSONAL mit der Tabelle ABTEILUNG, wobei der Attributwert von Abt_Id in PERSONAL mit dem Attributwert Id in ABTEILUNG übereinstimmen soll:

$$PERSONAL \bowtie_{PERSONAL.Aid=ABTEILUNG.Aid} ABTEILUNG$$

Dieser Join wird wie in Abb. 9 gezeigt konstruiert und liefert die in Abb. 10 dargestellte Ergebnistabelle. (Aus Gründen der Übersichtlichkeit weggelassene Spalten sind durch ... gekennzeichnet.)

□

Joins mit Gleichheitsbedingungen tauchen am häufigsten auf und werden als **Equi-Joins** bezeichnet. (Im Prinzip sind jedoch beliebige Bedingungen möglich.)

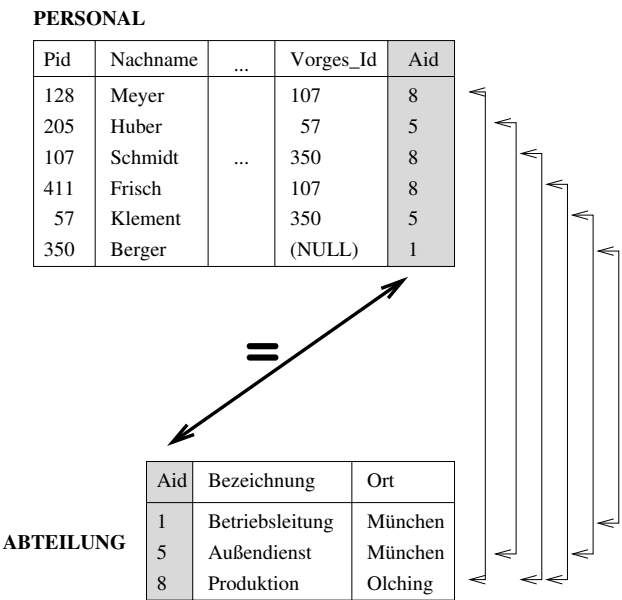


Abb. 9: Konstruktion eines Equi-Join

Pid	Nachname	...	PERSONAL.Aid	ABTEILUNG.Aid	Bezeichnung	ABTEILUNG.Ort
128	Meyer		8	8	Produktion	Olching
205	Huber		5	5	Außendienst	München
107	Schmidt	...	8	8	Produktion	Olching
411	Frisch		8	8	Produktion	Olching
57	Klement		5	5	Außendienst	München
350	Berger		1	1	Betriebsleitung	München

Abb. 10: Ergebnis eines Equi-Join

Nachteilig bei Equi-Joins ist, dass die verbundenen Attribute zweimal in der Ergebnistabelle auftauchen (von der ersten und der zweiten Relation). Daher schaltet man einem Equi-Join oft eine Projektion nach, die die duplizierten Attribute entfernt. Dies wird als **Natural-Join** bezeichnet.

Beim sog. **Self-Join** wird eine Relation mit sich selbst verbunden. Damit in der Join-Bedingung zwischen der ersten und zweiten Relation unterschieden werden kann, muss die betreffende Relation eine identische “Kopie” besitzen, die unter einem anderen Namen angesprochen werden kann.

Beispiel 2.13.: Sei PERSONAL auch unter dem Namen PERSO-

NAL2 ansprechbar. Es soll nun ein Join mit der Bedingung $PERSONAL.Vorges_Id = PERSONAL2.Pid$ durchgeführt werden. Die entstehende Relation soll auf $PERSONAL.Nachname$ und $PERSONAL2.Nachname$ projiziert werden (Abb. 11).

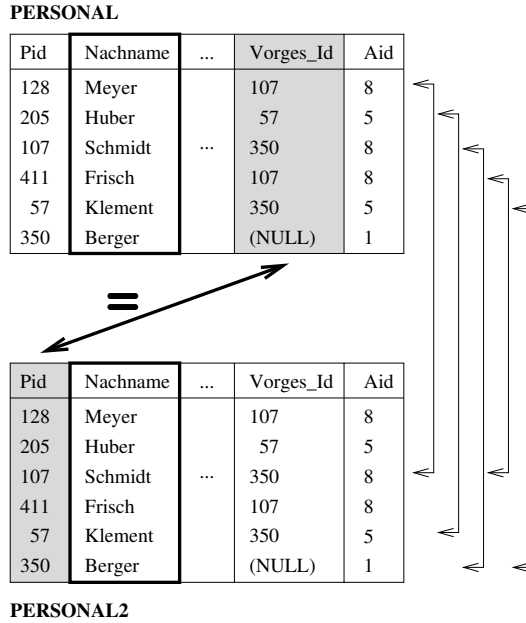


Abb. 11: Konstruktion eines Self-Join

$$\begin{aligned}
 & \pi_{PERSONAL.Nachname, PERSONAL2.Nachname} (\\
 & PERSONAL \bowtie_{PERSONAL.Vorges_Id = PERSONAL2.Pid} \\
 & PERSONAL2) = \\
 & \{ ('Meyer', 'Schmidt'), ('Huber', 'Klement'), ('Schmidt', 'Berger'), \\
 & ('Frisch', 'Schmidt'), ('Klement', 'Berger') \}
 \end{aligned}$$

Man beachte, dass die Ergebnisrelation kein Tupel mit erster Komponente 'Berger' enthält. Dies ist durch den NULL-Wert des Attributes $Vorges_Id$ bedingt. Ein NULL-Attributwert erfüllt niemals einen herkömmlichen logischen Vergleich.

□

2.4.2. Aggregationsoperationen

Es gibt eine Anzahl von Aggregationsfunktionen, die auf alle oder einen gewissen Teil der Datensätze einer Tabelle angewendet werden:

- COUNT: Zählen von Datensätzen bzw. Elementen
- SUM: Aufsummieren von numerischen Werten
- MINIMUM: Minimumsberechnung einer Reihe von numerischen Werten
- MAXIMUM: Maximumsberechnung einer Reihe von numerischen Werten
- AVERAGE: Berechnung des Durchschnitts (arithmetischen Mittels) einer Reihe von numerischen Werten.

Diese Funktionen werden jeweils auf Mengen von Datensätzen angewendet, bei denen bestimmte Attributwerte gleich sind. I. Z.

$$attribute_zur_gruppierung \mathcal{F}_{funktion} attribut, \dots(R)$$

Die Ergebnisrelation enthält die Werte der bei \mathcal{F} angegebenen Gruppierungsattribute sowie der angegebenen Funktionen, wobei für jede Menge von Datensätzen in der Ausgangsrelation, deren Attributwerte bezüglich *attribut_zur_gruppierung* übereinstimmen, ein eigener Datensatz angelegt wird.

Beispiel 2.14.: Es soll festgestellt werden, wie viele Angestellte in jeder Abteilung arbeiten (Abb. 12):

$$Aid \mathcal{F}_{COUNT} Pid(PERSONAL) = \{(8,3), (5,2), (1,1)\}$$

□

Tabelle PERSONAL

Pid	Nachname	...	Vorges_Id	Aid	group	count
128	Meyer		107	8	←	3
205	Huber		57	5		
107	Schmidt	...	350	8		
411	Frisch		107	8	←	2
57	Klement		350	5		
350	Berger		(NULL)	1	←	1

Abb. 12: Anwendung von Aggregationsoperationen

Beispiel 2.15.: Die höchste Personalnummer (Id) soll ermittelt werden:

$$\mathcal{F}_{MAXIMUM} Pid(PERSONAL) = \{(411)\}$$

Hier wurden keine Gruppierungsattribute angegeben, da sich die Funktion auf die gesamte Tabelle beziehen soll.



2.5. Zusatzforderungen für relationale DBMS

Zusätzlich zu den in Abschnitt 1.2 aufgelisteten Forderungen an allgemeine Datenbanksysteme gibt es für relationale Systeme weitergehende Forderungen. Auf die wichtigsten gehen wir im folgenden kurz ein, für eine vollständige Darstellung sei der Leser etwa auf [Da2004] verwiesen.

- Informationsregel: Alle Informationen werden nur auf eine Weise, nämlich durch Attributwerte von Datensätzen (Zeilen) in Tabellen dargestellt.
- Zugriffsgarantie: Auf jeden Wert in der Datenbank kann durch Angabe des Tabellennamens, des Spaltennamens und des Primärschlüsselwertes zugegriffen werden.
- Nicht bekannte Information: Es gibt einen besonderen Attributwert NULL, der nicht vorhandene oder unbekannte Informationen darstellt. Dieser Wert kann von allen anderen Werten, die in der Datenbank gespeichert werden können, unterschieden werden.
- Datenbankkatalog: Die Beschreibungen aller vorhandenen Datenbankobjekte stehen wie Benutzerdaten in Tabellenform zur Verfügung.
- Datenbank-Sprache: Es gibt eine Sprache, die sowohl interaktiv als auch in Applikationen verwendet werden kann und DDL-, DML- und DCL-Operationen (siehe Abb. 4 auf S. 6 sowie Abschnitt 3.3.2) unterstützt.
- Mengenorientierung: Insert-, Update-, Delete-Befehle auf Tabellen arbeiten mengenorientiert (z. B. sollten zwei identische Datensätze nicht vorkommen).
- Integritätsunabhängigkeit: Integritätsregeln werden im Datenbankkatalog abgelegt und in Datenbank-Sprache formuliert.

3. Datenbankoperationen mit SQL

Wir werden uns nun ausführlich mit SQL, der Standard-Datenbanksprache für relationale Datenbanksysteme, befassen. In einem relationalen Datenbanksystem sind alle Informationen in Tabellen gespeichert und Ergebnisse von Abfragen werden in Tabellenform geliefert.

Wie im letzten Abschnitt ausgeführt, sind Tabellen eine Darstellungsform von Relationen. Es gibt jedoch einen feinen Unterschied zwischen Tabellen in Datenbanken und Relationen als mathematischem Modell: Während Relationen nicht zwei identische Elemente enthalten können (eine Relation ist ja eine Menge), können in einer Tabelle durchaus mehrere gleiche Datensätze stehen. Falls erforderlich, kann man jedoch in Datenbanksystemen sicherstellen, dass auch eine Tabelle keine zwei gleichen Datensätze enthält.

3.1. Anforderungen an einen Datenbank-Server

Der Begriff “Datenbank-Server” wird oft in zwei verschiedenen Bedeutungen verwendet: Zum einen ist damit die Software gemeint, die die Datenbankfunktionen bereitstellt, zum anderen die Rechananlage (Hardware), auf der diese Software läuft. Wir verwenden den Begriff “Datenbank-Server” in der ersten Bedeutung und charakterisieren die Hardware, auf der die Datenbank-Serversoftware läuft, durch den Begriff “Server host”.

Die Sprache SQL bildet dann eine Schnittstelle zwischen dem Datenbank-Anwender und dem Datenbank-Server, die die Kommunikation mit dem Server auf verhältnismäßig hohem Abstraktionsniveau ermöglicht.

Wichtige Anforderungen an einen im Produktivbetrieb eingesetzten Datenbank-Server sind unter anderem:

- Steuerung von großen Datenbanken und entsprechende effiziente Speicherverwaltung
- Zugriff für viele gleichzeitige Datenbank-Benutzer
- Leistungsfähige Transaktionsverarbeitung
- Hohe Verfügbarkeit
- Realisierung von Industriestandards, z. B. SQL:1999, SQL:2003
- Konfigurierbare Sicherheit
- Überwachung der Datenbankintegrität

- Netzwerkfähigkeit und Unterstützung von Client/Server-Umgebungen
- Unterstützung verteilter Datenbanksysteme
- Portabilität, d. h. Einsetzbarkeit auf verschiedenen Plattformen

Der prinzipielle Aufbau eines Datenbank-Servers und der Zugriff darauf ist schematisch in Abb. 13 dargestellt.

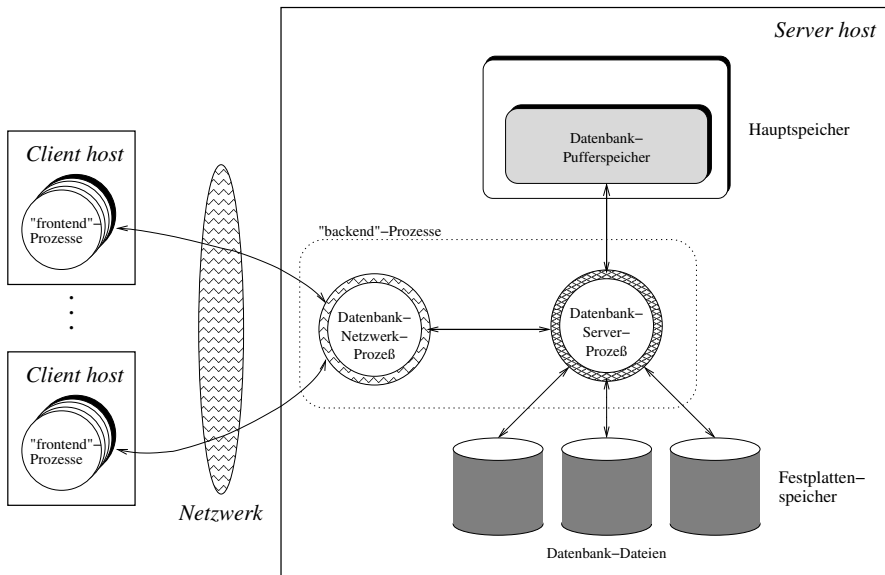


Abb. 13: Schematischer Aufbau eines Datenbank-Servers

Aufgrund der hohen Komplexität einer Datenbank-Verwaltung besteht ein DBMS meist aus mehreren Prozessen, die jeweils eine bestimmte Aufgabe bearbeiten. Die Prozesse lassen sich grob klassifizieren in:

- Netzwerkprozesse, die Benutzerzugriffe auf dem *Client host* an den eigentlichen Datenbank-Server weiterleiten und die Antworten des Servers wieder an die Benutzerprozesse zurückliefern.
- Serverprozesse auf dem *Server host*, die den Zugriff auf die physikalische Ebene der Datenbank regeln. Insbesondere obliegt den Serverprozessen eine effiziente Verwaltung des im Hauptspeicher befindlichen Datenbank-Pufferspeichers und des Festplattenspeichers.

Server- und Netzwerkprozesse müssen allerdings nicht notwendigerweise verschiedene Programme sein.

Die im vorhergehenden Abschnitt eingeführten Tabellen (Relationen) als Mittel zur Verwaltung von Daten werden durch den Datenbank-Server praktisch realisiert. Der Zugriff auf die Tabellen sowie die Durchführung der im letzten Abschnitt beschriebenen Operationen auf Tabellen erfolgt mit der Datenbank-Abfragesprache SQL.

3.2. SQL-Standards

SQL ist die Abkürzung für “Structured Query Language” (“Strukturierte Abfragesprache”). Die Sprache wurde Mitte der 70er Jahre in den IBM-Forschungslabors entworfen. Eine erste Standardisierung erfolgte im Jahr 1986 durch den Standard SQL-86 (dieser wird manchmal auch als SQL-87-Standard bezeichnet, da die Publikation erst Anfang 1987 erfolgte).

Es folgten die Standards SQL-89 und SQL-92; letzterer ist z. B. in [MeSi1993] ausführlich beschrieben. Der Standard, an dem sich viele aktuelle SQL-Implementierungen orientieren, ist Ende 1999 in Kraft getreten und wird daher als SQL:1999 bezeichnet.¹ SQL:1999 ist sehr umfangreich und in fünf Teile gegliedert, die jeweils in einem eigenen Standardisierungsdokument beschrieben werden ([SQL1999]).

SQL:1999 versucht insbesondere der Tatsache Rechnung zu tragen, dass SQL keine isolierte Abfragesprache ist, sondern mit der “Außenwelt” – etwa über sog. Host-Programmiersprachen wie z. B. C oder C++ – kommunizieren muss. Die Hersteller von Datenbankprodukten hatten in dieser Hinsicht schon länger eigene (herstellerspezifische) Lösungen bereitgestellt; der Standard versucht nun, einen allgemeingültigen Rahmen zu definieren, auch um eine gewisse Unabhängigkeit von Host-Applikationen und verwendeten Datenbanksystemen zu erreichen.

Die Grundlage von SQL:1999 ist *Core SQL*, das die Teile des Standards beinhaltet, die von einer SQL:1999-konformen Implementierung mindestens erfüllt werden müssen.

¹ Zwei an der Erarbeitung des neuen Standard beteiligte Autoren begründen in [EiMe1999] diese Namensgebung damit, dass bei Beibehaltung der bisherigen Namenskonvention – also hier SQL-99 – ein “Jahr 2000-Problem” in der folgenden Version des Standards auftreten würde, die dann beispielsweise SQL-03 heißen würde. Trotzdem findet man in der Literatur auch die Bezeichnung SQL-99 für den aktuellen Standard.

Eine wesentliche Neuerung in SQL:1999 ist die Unterstützung für objektorientierte Konzepte – es gibt zusammengesetzte benutzerdefinierte Typen und Methoden, die auf diesen Typen arbeiten. Auch das Konzept der Vererbung wird unterstützt. Im Rahmen dieses Buches werden wir aus Gründen des Umfangs nicht auf die objektorientierten Konzepte von SQL eingehen.

Der Vorgängerstandard SQL-92 sah drei Konformitätsebenen für Implementierungen vor: *Entry* als Grundvoraussetzung für SQL-92-Konformität, *Intermediate* und *Full*. Hingegen heißen Bestandteile von SQL:1999, die über Core SQL hinausgehen, *Features* und werden z. T. in *Packages* zusammengefasst. SQL-Implementierungen können dann zusätzlich zur SQL-Standardkonformität auch die Konformität zu gewissen Features oder Packages erklären. Die verschiedenen Features sind in den Anhängen der SQL-Standardisierungsdokumente [SQL1999] aufgeführt.

Seit Dezember 2003 gibt es den Standard SQL:2003 ([SQL2003]). Dieser ist im Wesentlichen eine fehlerbereinigte² Erweiterung von SQL:1999. Neu hinzugekommen sind insbesondere Festlegungen zu folgenden Themen:

- Zugriff auf externe (nicht in der SQL-Datenbank enthaltene) Daten mit SQL,
- Einbindung von SQL-Konstrukten in Java-Programme,
- Aufrufen von Java-Methoden in SQL sowie die Benutzung von Java-Klassen als SQL-Datentypen,
- Erzeugung und Manipulation von XML-Dokumenten in SQL.

Alle diese Erweiterungen sind optional und müssen von einer SQL:2003-konformen Implementierung nicht erfüllt werden. Im Rahmen dieses Buches, das ja eine Einführung in relationale Datenbanksysteme darstellen soll, werden wir auf diese Erweiterungen nicht weiter eingehen.

Wir werden uns in den folgenden Abschnitten zunächst ausschließlich mit SQL als Abfragesprache beschäftigen. Auf die Anbindung an höhere Programmiersprachen gehen wir im letzten Teil dieses Buches ein.

Die Ausführungen zu SQL orientieren sich an der SQL-Implementierung des Datenbanksystems PostgreSQL in der zur Zeit der

² Die “Technical Corrigenda” zu SQL:1999 umfassen mehrere hundert Seiten.

Manuskripterstellung aktuellen Version 7.4.2. Diese Version erfüllt in großen Teilen den Core-SQL:1999-Standard und bietet darüber hinaus zahlreiche in SQL:1999 definierte Features (siehe dazu Appendix D von [Pg2004]). Dieses Buch konzentriert sich auf die für den Entwurf und die Verwendung von relationalen Datenbanken wichtigsten SQL-Konstrukte, die in vielen SQL:1999/2003-konformen Implementierungen verfügbar sein sollten. Es soll weder den kompletten Sprachumfang von PostgreSQL noch den von (Core) SQL:1999/2003 behandeln. Hier sei der Leser auf die entsprechenden Standardisierungsdokumente bzw. die Dokumentation zu PostgreSQL [Pg2004] verwiesen.

Soweit in diesem Text PostgreSQL-spezifische Konstrukte verwendet werden, werden wir an der betreffenden Stelle deutlich auf diesen Umstand hinweisen.

3.3. Sprachstruktur von SQL

SQL ist eine nichtprozedurale Programmiersprache, d. h. in SQL beschreibt der Anwender, *was* er will, nicht *wie* es gemacht werden soll. Dies ist eine große Erleichterung gegenüber prozeduralen Programmiersprachen (PASCAL, C) und macht SQL schnell erlernbar. Alle Operationen auf der Datenbank werden mit SQL-Anweisungen (englisch: “statements”) durchgeführt.

Trotz der leichten Erlernbarkeit ist SQL kein Werkzeug, mit dem der Endbenutzer auf die Datenbank zugreift. SQL ist für Datenbankdesigner und Anwendungsentwickler gedacht; für den Endbenutzer gibt es komfortable Benutzerschnittstellen, z. B. graphische Tools oder Web-Interfaces. Vor allem letztere erlangen immer größere Bedeutung, so dass wir auf diese im letzten Teil dieses Buches ausführlicher eingehen.

3.3.1. Einführende Beispiele für SQL-Statements

Um vorweg einen Eindruck von SQL zu vermitteln, stellen wir im folgenden einige der bei der Beschreibung der Datenbankoperationen in Abschnitt 2.4 gegebenen Beispiele hier in SQL-Syntax vor. Wir beschränken uns jetzt noch auf reine Leseoperationen, die Informationen aus der Datenbank extrahieren, jedoch nichts an der Datenbank ändern. Solche Leseoperationen werden in SQL mit dem SELECT-Statement vorgenommen.

Das folgende und alle weiteren Beispiele im Textteil dieses Buches beziehen sich auf die Datenbank FIRMA aus Abb. 8; die zur Erzeugung dieser Datenbank notwendigen SQL-Befehle finden sich im Anhang D.

Vorweg sehen wir uns kurz die Struktur eines SELECT-Statements an; die Bezeichnungen für die einzelnen Teile eines SELECT-Statements sind in Abb. 14 dargestellt.

```
SELECT aid, COUNT(pid) FROM personal
      SELECT-Liste           Tabellen-Liste

WHERE pid < 400 AND pid > 100
      WHERE-Klausel

GROUP BY aid;
      GROUP BY-Klausel
```

Abb. 14: Bestandteile eines einfachen SELECT-Statements

Wir haben hier folgende Bestandteile vorliegen:

- SELECT-Liste: Dabei handelt es sich um Ausdrücke (im einfachsten Fall sind das Namen von Attributen), deren Werte in der Ergebnistabelle erscheinen sollen. Die Select-Liste wird für jeden Datensatz der Ergebnistabelle neu ausgewertet.
- Tabellen-Liste: Diese besteht aus den Namen der Tabellen, die in der SELECT-Abfrage benutzt werden, evtl. verknüpft durch JOIN-Operationen.
- WHERE-Klausel (optional): Nur diejenigen Datensätze, die die angegebenen Bedingungen erfüllen, werden für den Aufbau der Ergebnistabelle verwendet.
- GROUP BY-Klausel (optional): Hiermit werden Aggregationsoperationen ermöglicht (siehe Abschnitt 3.4.3).

Das in Abb. 14 angegebene SELECT-Statement liefert eine Liste von Abteilungsnummern und für jede Abteilung jeweils die Anzahl der Mitarbeiter mit Personalnummern zwischen 100 (ausschließlich) und 400 (ausschließlich), die in der betreffenden Abteilung arbeiten.

Die in Abschnitt 2.4 angeführten Beispiele für Datenbankoperationen lauten in SQL-Syntax wie folgt:

- Auswahl (Select):
`SELECT * FROM personal WHERE aid = 5;`
- Projektion:
`SELECT ort, aid FROM personal;`
- Vereinigung:
`SELECT ort FROM personal
UNION
SELECT ort FROM abteilung;`
- Kartesisches Produkt:
`SELECT * FROM abteilung, projekt;`
oder auch
`SELECT * FROM abteilung CROSS JOIN projekt;`
- Equi-Join:
`SELECT * FROM personal JOIN abteilung USING (aid);`
wobei hier auch gleich die Spalte `aid` nur einmal in der Ergebnistabelle auftritt.
- Self-Join:
`SELECT r1.nachname, r2.nachname
FROM personal r1 JOIN personal r2
ON r1.vorges_id = r2.pid;`
- Aggregation: Angestellte in jeder Abteilung:
`SELECT aid, COUNT(pid) FROM personal
GROUP BY aid;`
- Aggregation: Höchste Personalnummer:
`SELECT MAX(pid) FROM personal;`

3.3.2. Kategorien von SQL-Anweisungen

SQL-Anweisungen fallen in eine der folgenden Kategorien:

- DML (Data Manipulation Language)-Anweisungen
Hierunter versteht man Anweisungen, die Daten (aus Tabellen) extrahieren (`SELECT`), vorhandene Daten ändern (`UPDATE`), neue Daten hinzufügen (`INSERT`) und Daten löschen (`DELETE`).
- DDL (Data Definition Language)-Anweisungen
Darunter fällt das Erzeugen, Ändern und Löschen von Datenbankobjekten (z. B. Tabellen) (`CREATE`, `ALTER`, `DROP`) sowie die Erteilung oder das Entziehen von Zugriffsrechten (`GRANT`, `REVOKE`).

- DCL (Data Control Language)-Anweisungen
Diese Anweisungen bestimmen, dass Änderungen an den Daten endgültig übernommen werden (COMMIT) oder widerrufen werden (ROLLBACK).
- Die weiteren Anweisungskategorien “Sitzungskontrolle”, “Systemkontrolle” und “Embedded SQL-Statements” sind für uns jetzt nicht interessant.

3.3.3. Grundlegende Sprachkonstrukte

Eine Datenbank besteht aus **Objekten**. Für uns interessante Objekte sind

- Tabellen
- Virtuelle Tabellen (“Views”) (Abschnitt 3.5)
- Indizes (Abschnitt 4.4.4)
- Benutzerdefinierte Funktionen (Abschnitt 6.1)
- Trigger (Abschnitt 6.4)

Manche Objekte sind noch weiter unterteilt, z. B. bestehen Tabellen und Views aus Spalten.

Objekte werden zur Unterscheidung von anderen Objekten mit Namen bezeichnet. Namen von Objekten und Objektteilen dürfen bis zu 18 Zeichen lang sein.³ Sie dürfen aus alphanumerischen Zeichen sowie dem Zeichen `_` (“underscore”) zusammengesetzt sein. Es wird nicht zwischen Groß- und Kleinschreibung unterschieden, alle Großbuchstaben werden intern von PostgreSQL in Kleinbuchstaben umgewandelt.⁴

Schlüsselwörter, die eine spezielle Bedeutung in SQL haben, dürfen nicht als Namen verwendet werden.

³ Dies ist die Mindestforderung des SQL-Standards. Tatsächlich steht es einem DBMS frei, eine Maximallänge von bis zu 128 Zeichen für einen Objektnamen zuzulassen; bei PostgreSQL ist die Maximallänge 63 Zeichen.

⁴ Der SQL-Standard schreibt gerade die umgekehrte Umwandlungsrichtung (Klein- in Großbuchstaben) vor. Solange man nicht von der sowohl in PostgreSQL als auch im SQL-Standard vorgesehenen Möglichkeit Gebrauch macht, die automatische Umwandlung durch Einschließen des Objektnamens in doppelte Anführungszeichen zu unterdrücken, sollten sich hier keine Kompatibilitätsprobleme ergeben.

Jedes Objekt in der Datenbank ist einem **Schema** zugeordnet. Innerhalb eines Schemas müssen Objektnamen eindeutig sein. In Mehrbenutzersystemen ist es üblich, für jeden Benutzer ein eigenes Schema zu definieren (analog einem “Heimatverzeichnis” bei Betriebssystemen).

Die Syntax (Schreibweise) für die Bezeichnung von Objekten bzw. Objektteilen ist in Abb. 15 dargestellt.

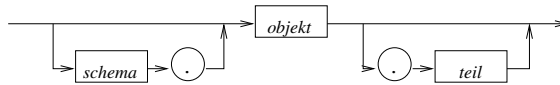


Abb. 15: Syntax für die Bezeichnung von Objekten und Objektteilen

Hierbei bedeutet

schema Name des Schemas. Wird er weggelassen, bezieht sich die Angabe auf das eigene Schema (dessen Name mit dem Benutzernamen in der Datenbank übereinstimmt). Zugriff auf Objekte in einem anderem Schema als dem eigenen ist nur mit entsprechenden Privilegien oder Zugriffsrechten möglich (siehe Abschnitt 5.4).

objekt Name des Objekts (z. B. einer Tabelle).

teil Name eines Teils des Objekts, z. B. einer Spalte in einer Tabelle.

Literale sind numerische Konstanten oder Textkonstanten.

Beispiele für numerische Literale sind:

1433

-45

34.67

-227.09

-4.5e3

6.112e-5

In den ersten beiden Beispielen werden ganze Zahlen dargestellt. Zahlen dieses Typs bezeichnet man als **Integerzahlen**. Die letzten beiden Beispiele stellen Zahlen in wissenschaftlicher Notation dar (bekannt vom Taschenrechner) und entsprechen $-4.5 \cdot 10^3 = -4500$ bzw. $6.112 \cdot 10^{-5} = 0.00006112$.

Beispiele für Textliterale sind:

'Vorlesung'

```
'Ein ganzer Satz.'  
'Ich hab''s!'
```

Merke bei Textliteralen:

- Textliterals sind in einfache Anführungszeichen eingeschlossen. Es können stattdessen keine doppelten Anführungszeichen verwendet werden!
- Innerhalb von Textliteralen ist Groß- und Kleinschreibung signifikant.
- Soll ein einfaches Anführungszeichen innerhalb eines Textliterals stehen, so muss es zweimal hintereinander geschrieben werden.

Wir werden in Abschnitt 3.7 nochmals auf Literale für die Angabe von Werten für spezielle Datentypen (insbesondere Datums- und Zeitangaben) zurückkommen.

3.3.4. Datentypen

Attribute von Tabellen müssen einem festen Datentyp zugeordnet werden.

Für uns interessant sind folgende SQL-Datentypen:

INTEGER	(Äquivalente Bezeichnung: INT.) Ganze Zahlen im Intervall von -2^{31} bis $2^{31} - 1$.
BIGINT	Ganze Zahlen im Intervall von -2^{63} bis $2^{63} - 1$. (Der SQL-Standard spricht hier nur davon, dass BIGINT einen mindestens ebenso großen Darstellungsbereich hat wie INTEGER.)
NUMERIC(p, s)	Festpunktzahlen mit höchstens p signifikanten Stellen (d. h. die Anzahl der gültigen Stellen vor und hinter dem Dezimalpunkt) und s Stellen hinter dem Dezimalpunkt ($p \geq 1, s \geq 0$). In PostgreSQL können bis zu 1000 signifikante Stellen gespeichert werden.
NUMERIC(p)	entspricht NUMERIC($p, 0$).
REAL	Gleitpunktzahlen mit einfacher Genauigkeit. Darstellungsbereich und Genauigkeit sind implementierungsabhängig (sowohl vom DBMS als auch von der Systemplattform, auf dem das DBMS läuft); auf den meisten PostgreSQL-Plattformen liegt der absolutbetragsmäßige Darstellungsbereich zwischen 10^{-37} und 10^{+37} bei einer Genauigkeit (Anzahl der signifikanten Stellen) von 6 Dezimalstellen.

DOUBLE PRECISION

Gleitpunktzahlen mit doppelter Genauigkeit. Auch hier sind Darstellungsbereich und Genauigkeit wie bei **REAL** implementierungsabhängig. Auf den meisten PostgreSQL-Plattformen liegt der absolutbetragsmäßige Darstellungsbereich zwischen 10^{-307} und 10^{308} bei einer Genauigkeit von mindestens 15 Dezimalstellen.

CHARACTER(s) (Äquivalente Bezeichnung: **CHAR**.) Zeichenketten mit fester Länge (Anzahl der Zeichen) von *s*. Kürzere Zeichenketten werden bei Speicherung in diesem Datentyp am Ende mit Leerzeichen auf die Länge *s* aufgefüllt.

CHARACTER VARYING(s)

(Äquivalente Bezeichnung: **VARCHAR**.) Zeichenketten variabler Länge mit einer Maximallänge von *s* Zeichen.

TEXT Zeichenketten beliebiger Länge. Dieser Typ ist nicht im SQL-Standard definiert.

DATE Datumsangaben.

TIME Uhrzeitangaben.

TIMESTAMP Zeitpunktangaben (Datum und Uhrzeit).

INTERVAL Zeitdauer. Im SQL-Standard heißt dieser Typ **INTERVAL SECOND**.

BYTEA Beliebige Binärdaten. Im SQL-Standard gibt es einen ähnlichen Datentyp **BINARY LARGE OBJECT**.

In SQL ist es möglich, Wertelisten (in Verbindung mit entsprechenden Vergleichsoperatoren, siehe unten) anzugeben; dazu listet man die betreffenden Werte durch Kommata getrennt auf und setzt das Ganze in runde Klammern, etwa

(**'Buchhaltung', 'Vertrieb', 'Einkauf'**)

Der spezielle Wert **NULL** dient zur Darstellung von unbekannten Werten in Tabellenspalten. **NULL** gehört keinem der oben beschriebenen Datentypen an. **NULL** ist insbesondere nicht zu verwechseln mit der Zahlenkonstante 0, die einen konkreten Wert, eben die Zahl 'Null' darstellt, während **NULL** einen unbekannten oder nicht verfügbaren Wert darstellt. **NULL** wird in Ausdrücken gesondert behandelt (siehe unten).

3.3.5. Ausdrücke

Diese können ähnlich wie bei anderen Programmiersprachen gewohnt gebildet werden. Für arithmetische Ausdrücke gibt es die Opera-

toren $+$, $-$, $*$, $/$. Die Binderegeln entsprechen der gewohnten “Punkt-vor-Strich”-Regel, d. h. $*$ und $/$ binden stärker als $+$ und $-$. Ein (Teil-)Ausdruck mit mehreren Operatoren gleich starker Bindung wird von links nach rechts ausgewertet. Um die Binderegeln zu umgehen, kann man Klammern setzen. Für Zeichenkettenausdrücke gibt es nur einen Operator, den Konkatenationsoperator $||$.

Logische Ausdrücke entstehen durch Vergleichsoperationen oder die Verknüpfung von weiteren logischen Ausdrücken (mit AND, OR oder NOT, siehe unten). SQL kennt folgende Vergleichsoperatoren: $=$, $<>$, $<$, $>$, $<=$, $>=$. Diese arbeiten bei Zahlen wie gewohnt. Bei Datums- und Zeitangaben wird ein Wert d_1 genau dann größer als ein Wert d_2 angesehen, wenn d_1 einen späteren Zeitpunkt als d_2 angibt.

Vergleiche zwischen Zeichenketten werden abhängig von den Datentypen der Operanden durchgeführt:

- Hat mindestens ein Operand den Datentyp `CHARACTER(s)`, so werden Leerzeichen am Ende eines der beiden Operanden ignoriert. Im übrigen wird ein lexikographischer Vergleich (s. u.) durchgeführt. Durch die Nichtbeachtung von Leerzeichen am Ende von Operanden werden zwei Operanden auch als gleich angesehen, wenn sie am rechten Ende eine unterschiedliche Anzahl von Leerzeichen besitzen.
- Sind beide Operanden vom Datentyp `CHARACTER VARYING(s)` oder Textlitterale, so wird sofort ein lexikographischer Vergleich durchgeführt. Leerzeichen am Ende der Operanden werden beachtet.

Lexikographischer Vergleich bedeutet, dass die Zeichenketten von links her zeichenweise verglichen werden, bis ein Unterschied gefunden wird. Die Zeichenkette, die an der betreffenden Stelle das größere Zeichen enthält, wird als insgesamt größer angesehen. Stimmt die Zeichenkette s_1 zeichenweise mit der Zeichenkette s_2 überein, ist jedoch s_1 früher zu Ende als s_2 , so ist s_1 kleiner als s_2 . Für den Vergleich von Zeichen gilt (\square markiert ein Leerzeichen)

$$' \square ' < '0' < \dots < '9' < 'A' < \dots < 'Z' < 'a' < \dots < 'z'.$$

Diese Beziehung gilt allerdings nur dann, wenn der Grundzeichensatz der Datenbank der Standard-ASCII-Zeichensatz (ohne Umlaute) ist. Bei anderen (nationalen) Zeichensätzen gelten andere Vergleichsbeziehungen.

Beispiel 3.1.: Hat s_1 den Wert `'ab□'` (am Ende steht ein Leerzeichen) und s_2 den Wert `'ab'` (ohne Leerzeichen am Ende), so erhält man je nach den Datentypen von s_1 und s_2 unterschiedliche Vergleichsergebnisse, die in Abb. 16 dargestellt sind. Um Überraschungen zu

vermeiden, ist es am besten, die Typen `CHARACTER` und `CHARACTER VARYING` bei Vergleichen nicht zu mischen.

□

<i>Datentyp s_1</i>	<i>Datentyp s_2</i>	<i>Ergebnis von $s_1 > s_2$</i>
CHARACTER	CHARACTER	falsch (denn $s_1 = s_2$ ist wahr)
CHARACTER	CH. VARYING	falsch (denn $s_1 = s_2$ ist wahr)
CH. VARYING	CHARACTER	falsch (denn $s_1 = s_2$ ist wahr)
CH. VARYING	CH. VARYING	wahr

Abb. 16: Zeichenkettenvergleiche mit verschiedenen Datentypen

Für Zeichenketten gibt es außerdem den besonderen Vergleichsoperator `LIKE`, der Vergleiche mit sog. Patterns ermöglicht. Ein Pattern ist eine Zeichenkette, in der die Zeichen ‘%’ und ‘_’ eine Sonderbedeutung haben: ‘%’ steht für eine beliebig lange (auch leere) Folge von beliebigen Zeichen, ‘_’ steht für ein beliebiges Zeichen. Die Syntax für einen Vergleich mit `LIKE` ist

zeichenkette `LIKE` *pattern*

Der `LIKE`-Operator liefert genau dann das Ergebnis ‘wahr’, wenn die Zeichenkette auf das angegebene Pattern passt.

Beispiel 3.2.: Die Ergebnisse von verschiedenen Vergleichen mit Patterns kann man Abb. 17 entnehmen.

□

<i>Ausdruck</i>	<i>Ergebnis</i>
<code>'Meier' LIKE 'M%'</code>	wahr
<code>'WMeier' LIKE 'M%'</code>	falsch
<code>'Meier' LIKE 'M_%'</code>	wahr
<code>'Mei' LIKE 'M_%'</code>	wahr
<code>'Me' LIKE 'M_%'</code>	falsch

Abb. 17: Ergebnisse von Vergleichen mit Patterns

Zusätzlich gibt es noch folgende Listenoperatoren, die mit Listen arbeiten und logische Werte liefern:

x `IN` *liste* Testet, ob x in der *liste* vorkommt.

x `NOT IN` *liste* Testet, ob x nicht in der *liste* vorkommt.

x `= ANY` *liste* Testet, ob x mit irgendeinem Element der *liste* übereinstimmt. Statt `=` kann man hier auch einen der anderen Vergleichsoperatoren verwenden. Der Operator `= ANY` entspricht dem Operator `IN`.

$x = \text{ALL } \textit{liste}$ Testet, ob x mit allen Elementen der Liste übereinstimmt. Statt $=$ kann man hier auch einen der anderen Vergleichsoperatoren verwenden.

Logische Ausdrücke lassen sich wie in anderen Programmiersprachen mit NOT, AND, OR kombinieren. Hierbei bindet NOT stärker als AND, das wiederum stärker als OR bindet. Klammern können gesetzt werden, um diese Binderegeln zu umgehen oder um den Ausdruck übersichtlicher zu gestalten.

Alle Ausdrücke, die einen NULL-Wert enthalten, evaluieren in jedem Fall zum Wert NULL. Dies gilt auch bei logischen Ausdrücken; da NULL als besonderer Wert weder wahr noch falsch ist, kann man *nicht* mit Ausdrücken der Form

$x = \text{NULL}$

testen, ob x gleich dem Wert NULL ist, da der Gesamtausdruck in jedem Fall wieder zu NULL evaluiert. Um zu testen, ob ein Wert NULL ist oder nicht, benötigt man daher die folgenden speziellen NULL-Operatoren:

$x \text{ IS NULL}$ Testet, ob x der NULL-Wert ist.

$x \text{ IS NOT NULL}$ Testet, ob x nicht der NULL-Wert ist.

Diese beiden Operatoren sind von der Regel, dass Ausdrücke, die NULL-Werte enthalten, zu NULL evaluieren, ausgenommen.

Auch mit Werten der Datentypen DATE, TIME, TIMESTAMP und INTERVAL können arithmetische Operationen durchgeführt werden; in Abb. 18 sind die wichtigsten Möglichkeiten aufgeführt.

Operand 1	Operator	Operand 2	Ergebnis (SQL-Std.)	Ergebnis (PostgreSQL)
DATE	+, −	INTEGER	<i>nicht zulässig</i>	<u>DATE</u>
DATE	+, −	INTERVAL	TIMESTAMP	TIMESTAMP
TIMESTAMP	+, −	INTERVAL	TIMESTAMP	TIMESTAMP
TIME	+, −	INTERVAL	TIME	TIME
INTERVAL	+, −	INTERVAL	INTERVAL	INTERVAL
DATE	−	DATE	INTERVAL	<u>INTEGER</u>
TIMESTAMP	−	TIMESTAMP	INTERVAL	INTERVAL
INTERVAL	*, /	INTEGER	INTERVAL	INTERVAL

Abb. 18: Arithmetik mit Datums- und Zeitangaben

Die Syntax von Ausdrücken ist nochmals in Abb. 19 überblicksmäßig zusammengefasst. Alle syntaktischen Elemente außer *funktion* wur-

den bereits eingeführt. Funktionen behandeln wir im folgenden Abschnitt 3.3.6.

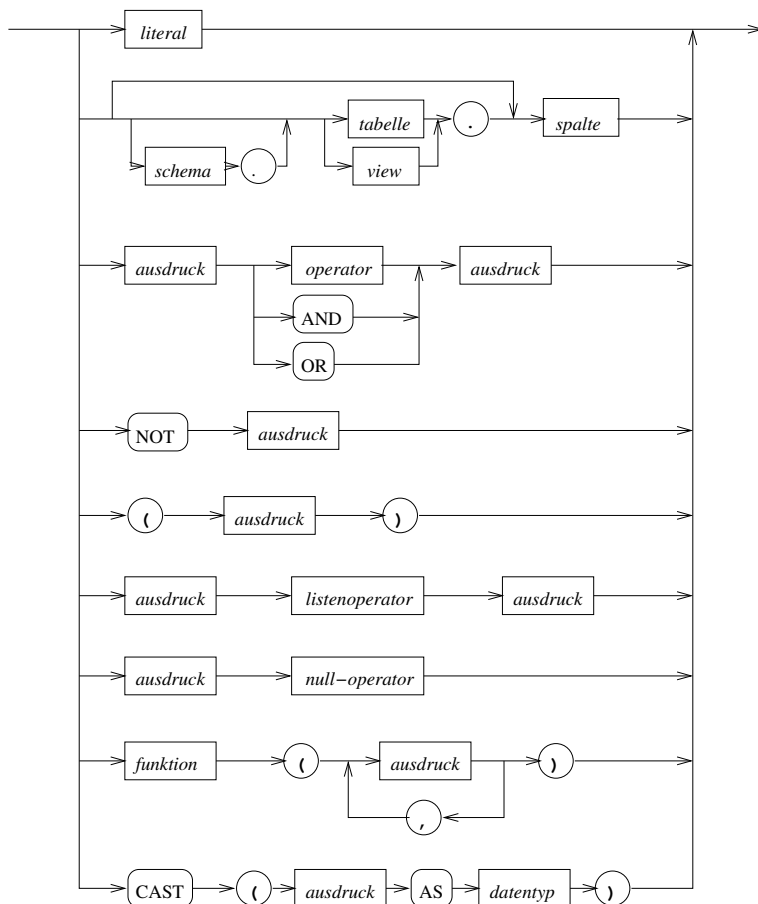


Abb. 19: Syntax für Ausdrücke

3.3.6. Funktionen

SQL bietet für die Auswertung von Ergebnissen einer Datenbankabfrage zwei Klassen von Funktionen an:

- *Skalare Funktionen*, die auf einem Resultatwert (z. B. auch dem Wert eines Attributs) arbeiten. Dieser Typ von Funktionen kann in SELECT-Listen und WHERE-Klauseln verwendet werden.
- *Aggregationsfunktionen*, die für eine Gruppe von Resultatwerten (einer Gruppe von Tabellenzeilen) einen Wert liefern. Dieser Typ von

Funktionen kann in SELECT-Listen und HAVING-Klauseln (s. u.) verwendet werden.

Skalare numerische Funktionen sind unter anderem:

ABS(n)	Liefert den Absolutbetrag von n .
POW(m, n)	Liefert die n -te Potenz von m , d. h. m^n . Im SQL-Standard heißt die Potenzfunktion POWER.
ROUND(n)	Rundet n auf einen ganzzahligen Wert. Nicht im Core-SQL-Standard enthalten.
ROUND(n, m)	Rundet n auf die m -te Stelle hinter dem Dezimalpunkt. Nicht im Core-SQL-Standard enthalten.

Skalare Zeichen(ketten)funktionen (Zeichenkettenargumente sind vom Typ CHARACTER, CHARACTER VARYING oder TEXT) sind unter anderem:

LOWER(s)	Wandelt alle Großbuchstaben in s in Kleinbuchstaben um.
SUBSTRING(s FROM m)	Liefert die letzten Zeichen von s ab dem m -ten Zeichen.
SUBSTRING(s FROM m FOR n)	Liefert die letzten Zeichen von s ab dem m -ten Zeichen, wobei die resultierende Zeichenkette nach n Zeichen abgeschnitten wird.
UPPER(s)	Wandelt alle Kleinbuchstaben in s in Großbuchstaben um.
LPAD(s, n)	Ist s kürzer als n Zeichen, wird s von links mit Leerzeichen aufgefüllt, so dass die resultierende Zeichenkette die Länge n hat. Ist s länger als n Zeichen, so wird s auf n Zeichen verkürzt, indem die Zeichen am Ende von s weggelassen werden. Nicht im Core-SQL-Standard enthalten.
RPAD(s, n)	Wie LPAD, jedoch wird von rechts mit Leerzeichen aufgefüllt bzw. werden Zeichen am Anfang von s weggelassen. Nicht im Core-SQL-Standard enthalten.
CHAR_LENGTH(s)	Liefert die Anzahl der Zeichen in s .

Skalare Funktionen, die Datums- bzw. Zeitangaben liefern, sind beispielsweise:

CURRENT_DATE	Liefert das aktuelle Datum.
--------------	-----------------------------

CURRENT_TIME	Liefert die aktuelle Uhrzeit mindestens sekunden genau.
CURRENT_TIMESTAMP	Liefert das aktuelle Datum und die aktuelle Uhrzeit mindestens sekunden genau.

Eine spezielle skalare Funktion ist:

COALESCE(e_1, e_2, \dots, e_n)	Liefert den Wert des ersten Ausdrucks e_i , der nicht zu NULL evaluiert, d. h.
------------------------------------	--

$$\text{COALESCE}(e_1, e_2, \dots, e_n) = e_i$$

genau dann, wenn e_j IS NULL für alle $j < i$ und e_i IS NOT NULL. Evaluieren alle e_j zu NULL, so evaluiert auch COALESCE zu NULL.

Aggregationsfunktionen, auch Gruppenfunktionen genannt, sind (hier ist e ein Ausdruck, der eine oder mehrere Tabellenspalten enthält):

AVG(e)	Liefert den Durchschnittswert aller Werte von e .
COUNT(e)	Liefert die Anzahl der Werte von e , wo e nicht NULL wird. Wird für e ein * angegeben, werden alle Werte (auch NULL-Werte) gezählt.
MAX(e)	Liefert das Maximum aller Werte von e .
MIN(e)	Liefert das Minimum aller Werte von e .
SUM(e)	Liefert die Summe aller Werte von e .

NULL-Werte werden in Aggregationsfunktionen – außer bei COUNT(*) – nicht in Berechnungen einbezogen.

Die Aggregationsfunktionen existieren auch in den Formen

- AVG(DISTINCT e),
- COUNT(DISTINCT e),
- MAX(DISTINCT e),
- MIN(DISTINCT e), und
- SUM(DISTINCT e).

Bei Verwendung einer dieser Formen werden mehrfach vorkommende Werte von e nur einmal verwendet (siehe dazu auch die Erläuterung von DISTINCT auf Seite 38).

3.3.7. Datentypkonversion

Wird in einem Ausdruck ein anderer Datentyp benötigt, als tatsächlich verwendet wird, so erfolgt automatisch eine Typkonversion in den benötigten Datentyp, soweit dies Sinn macht. Diesen Vorgang bezeichnet man als **implizite Typkonversion**.

Nicht sinnvoll ist etwa der Ausdruck

`1201 + 'Martin'`

hingegen wird im Ausdruck

`1201 + '23'`

die Zeichenkette '23' in die Zahl 23 umgewandelt und damit die Addition ausgeführt.

Auch der Ausdruck

`12 || '34'`

ist sinnvoll; hier wird die Zahl 12 in die Zeichenkette '12' umgewandelt und dann die Verkettungsoperation ausgeführt, die das Ergebnis '1234' liefert.

Die implizite Typkonversion ist vor allem auch wichtig bei der Verarbeitung von Datums- und Zeitangaben, da diese als Zeichenketten ein- und ausgegeben werden und somit eine Umwandlung in das interne Darstellungsformat erforderlich ist. Wir gehen in Abschnitt 3.7 genauer darauf ein.

Implizite Typkonversionen werden vom Datenbanksystem stets nur dann vorgenommen, wenn das Ergebnis das logisch zu erwartende ist (keine "Überraschungen").

In Fällen, wo keine implizite Typkonversion möglich ist, aber eine Konversion gewünscht wird bzw. erforderlich ist, muss sie mit dem speziellen **CAST**-Operator vorgenommen werden:

CAST (*e AS t*)

konvertiert den Typ des Ausdrucks *e* in den Datentyp *t*, wenn dies sinnvoll möglich ist.

Spezielle Konversionsfunktionen, die beim Umwandeln von eingegebenen Daten in Datentypen der Datenbank sowie beim Ausgeben von Werten der Datenbank zur Anwendung kommen, stellen wir in Abschnitt 3.7 vor.

3.4. Abrufen von Informationen mit **SELECT**

Das wichtigste SQL-Konstrukt ist zunächst das **SELECT**-Statement, das es gestattet, Informationen aus den Tabellen der Datenbank abzurufen.

3.4.1. Die Syntax des **SELECT**-Statements

In Abb. 20 ist die formale Syntax des **SELECT**-Statements als Syntaxdiagramm dargestellt. Das vollständige **SELECT**-Statement ist noch

etwas komplexer als die hier dargestellte Version; wir beschränken uns zunächst auf die am häufigsten benötigten Teile und verfeinern das Syntaxdiagramm später bei Bedarf entsprechend.

Die folgenden Konstrukte sind dabei neu:

DISTINCT	Es werden keine gleichen Datensätze mehrfach in die Ergebnistabelle übernommen. Ist dies nicht oder ist
ALL	angegeben, werden auch mehrfach vorkommende gleiche Datensätze mehrfach in die Ergebnistabelle übernommen.
*	bezeichnet alle Spalten einer Tabelle (wenn mit einem Tabellennamen qualifiziert) bzw. alle Spalten der Ergebnistabelle (wenn nicht mit einem Tabellennamen qualifiziert).
<i>s_alias</i>	ist der Name, den die betreffende Spalte in der Ergebnistabelle erhalten soll. Wird hier nichts angegeben, so ist der Spaltenname gleich dem Ausdruck.
<i>t_alias</i>	ist der Name, unter dem die betreffende Tabelle in dieser Abfrage angesprochen werden soll. Muss bei Self-Joins verwendet werden.
<i>join_klausel</i>	siehe Abschnitt 3.4.5. Eine <i>join_klausel</i> kann auch einfach ein Name einer Tabelle sein.
<i>bedingung</i>	ist ein logischer Ausdruck. Ein Datensatz wird dann in die Ergebnistabelle übernommen, wenn die Bedingung erfüllt ist.
GROUP BY, HAVING	siehe Abschnitt 3.4.3.
ORDER BY	bewirkt die Sortierung der Ergebnistabelle nach den Werten der angegebenen Ausdrücke bzw. Positionen in der Ergebnistabelle. Siehe auch Abschnitt 3.4.4.
<i>position</i>	Nummer der Spalte in der Ergebnistabelle, die für die Sortierung verwendet wird. (Die Spalten werden hierbei bei 1 beginnend durchnumeriert.)
UNION, UNION ALL, INTERSECT, EXCEPT	siehe folgenden Abschnitt 3.4.2.

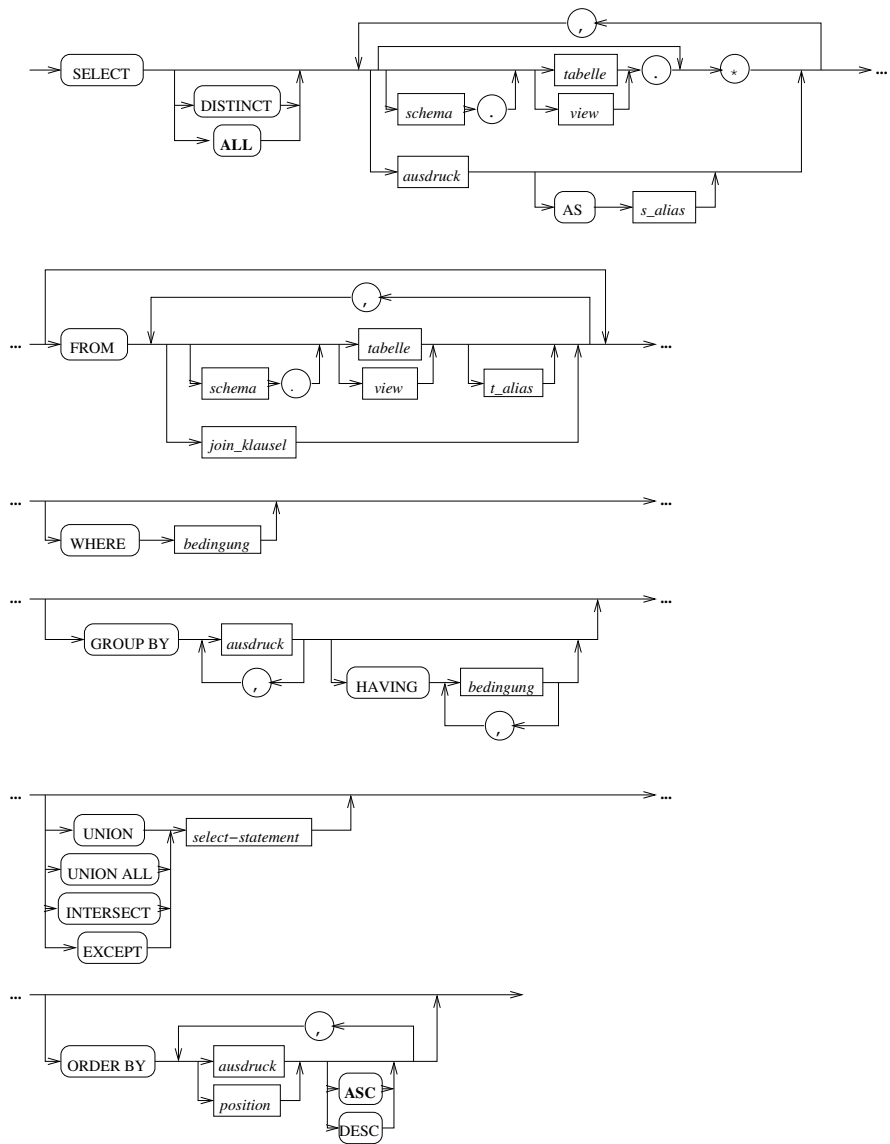


Abb. 20: Syntax des SELECT-Statements

ASC aufsteigende Sortierung (Standardeinstellung).

DESC absteigende Sortierung.

Für jeden der selektierten Datensätze werden die Werte der Spalten bzw. Ausdrücke in der SELECT-Liste in die Ergebnistabelle, die dann am Bildschirm dargestellt wird, übernommen.

Die FROM-Klausel kann weggelassen werden, wenn sich das SELECT-Statement nicht auf Tabellen stützt, z. B. liefert

```
SELECT CURRENT_TIMESTAMP;
```

das aktuelle Datum und die aktuelle Uhrzeit.

Einfache Beispiele für SELECT-Statements hatten wir bereits in Abschnitt 3.3.1 gegeben; wir werden im folgenden genauer auf die etwas komplexeren Konstrukte eingehen.

3.4.2. Mengenoperationen mit Abfragen

Zwei oder auch mehrere Abfragen können mit den Mengenoperatoren UNION, UNION ALL, INTERSECT, EXCEPT zu einer Abfrage verknüpft werden. Die Anzahl der Elemente in der SELECT-Liste sowie ihre Datentypen müssen bei den Einzelabfragen jeweils übereinstimmen. Die Ergebnistabelle einer Abfrage der Form

abfrage₁ mengenoperator abfrage₂

sieht je nach verwendetem Mengenoperator wie folgt aus:

UNION Die Ergebnistabelle besteht aus den Zeilen der Ergebnistabellen von *abfrage₁* und *abfrage₂*, wobei mehrfach vorkommende Zeilen entfernt werden.

UNION ALL wie bei UNION, aber doppelte Zeilen bleiben erhalten.

INTERSECT Die Ergebnistabelle besteht aus den Zeilen, die sowohl in der Ergebnistabelle von *abfrage₁* als auch in der von *abfrage₂* auftreten.

EXCEPT Die Ergebnistabelle enthält alle Zeilen der Ergebnistabelle von *abfrage₁*, die nicht in der Ergebnistabelle von *abfrage₂* vorkommen.

Beispiel 3.3.: Abteilungsstandorte, an denen keine Mitarbeiter wohnen:

```
SELECT ort FROM abteilung
EXCEPT
```

```
SELECT ort FROM personal;
```



Eine in einer Abfrage mit Mengenoperatoren vorkommende **ORDER BY**-Klausel bezieht sich stets auf die gesamte Abfrage und kann daher nur am Schluss der Abfrage verwendet werden.

3.4.3. Aggregation und Selektion mit **GROUP BY**- und **HAVING**-Klauseln

Eine **GROUP BY**-Klausel dient zur Zusammenfassung (Aggregation) von selektierten Datensätzen. Für jede Gruppe von Datensätzen wird ein Datensatz in der Ergebnistabelle produziert (soweit nicht durch **HAVING** weiter eingeschränkt), wobei jeweils die Datensätze gruppiert werden, für die die Ausdrücke in der **GROUP BY**-Klausel gleiche Werte haben.

SELECT-Statements mit **GROUP BY**-Klausel können in der **SELECT**-Liste nur folgende Typen von Ausdrücken bzw. Tabellenspalten enthalten:

- Konstanten,
- Aggregationsfunktionen,
- in der **GROUP BY**-Klausel vorkommende Ausdrücke und Spalten,
- aus den obigen Typen zusammengesetzte Ausdrücke, die für jeden Datensatz einer Gruppe den gleichen Wert haben.

Beispiel 3.4.: Folgendes SQL-Statement ist demnach *inkorrekt*:

```
SELECT nachname FROM personal GROUP BY ort;
```

Hier würden Datensätze mit gleichem Wert des Attributs **ort** zu einer Gruppe zusammengefasst. Das Attribut **nachname** kann aber innerhalb einer Gruppe durchaus unterschiedliche Werte haben, so dass hier kein definierter Wert zur Verfügung steht.



Es gilt also: Jedes in der **SELECT**-Liste vorkommende Attribut, das nicht Argument einer Gruppenfunktion ist, muss auch in der **GROUP BY**-Klausel vorkommen.

Mit der **HAVING**-Klausel werden die Gruppen von Datensätzen selektiert, für die ein Datensatz in der Ergebnistabelle produziert werden soll. Ohne eine **HAVING**-Klausel wird für jede Gruppe von Datensätzen ein Datensatz in der Ergebnistabelle produziert.

Beispiel 3.5.: Liste der Projektbezeichnungen der Projekte mit mindestens zwei zugeordneten Mitarbeitern (wir verwenden die **JOIN**-

Syntax hier zunächst informell, da diese erst in Abschnitt 3.4.5 eingeführt wird):

```
SELECT name FROM projekt JOIN zuordnung USING (projid)
GROUP BY name
HAVING COUNT(pid) >= 2;
```

Betrachten wir zum Verständnis dieses Beispiels erst die Ergebnistabelle des Join-Anteils der Abfrage, also:

```
SELECT * FROM projekt JOIN zuordnung USING(projid);
```

und sehen dann, was durch GROUP BY und HAVING damit passiert (Abb. 21).



			Gruppierung durch GROUP BY	count(pid)	durch HAVING selektiert
Projid	Name	Pid			
3	Druckauftrag Fa. Karl	128]	2	ja
3	Druckauftrag Fa. Karl	411			
8	Beratung Fa. Seidl	205]	1	nein
11	Werbung Fa. Rieger	411]	2	ja
11	Werbung Fa. Rieger	107			

Abb. 21: Auswertung einer SELECT-Abfrage mit GROUP BY- und HAVING-Klausel

Wird in der SELECT-Liste eine Aggregationsfunktion verwendet, ohne dass das SELECT-Statement eine GROUP BY-Klausel enthält, so werden implizit *alle* selektierten Datensätze als eine Gruppe aufgefasst.

Beispiel 3.6.:

```
SELECT COUNT(*) FROM projekt;
```

liefert die Anzahl der Zeilen (Datensätze) in der Tabelle projekt.



Gruppenfunktionen lassen sich nicht vernesten, d. h. auf das Ergebnis einer Gruppenfunktion kann nicht einfach eine weitere Gruppenfunktion angewendet werden. (Eine solche Funktionalität wäre wünschenswert, wenn wir in unserer Beispieldatenbank etwa die Zahl der Mitarbeiter in der mitarbeiterstärksten Abteilung bestimmen wollten.) Man kann jedoch den Effekt einer Vernestung unter Zuhilfenahme einer View (siehe Abschnitt 3.5) erreichen.

3.4.4. Sortierung der Ergebnistabelle

Bei *SELECT*-Anweisungen gibt es grundsätzlich keine Garantie dafür, dass die Datensätze der Ergebnistabelle in irgendeiner bestimmten Reihenfolge angezeigt werden.

Jedoch kann durch die zusätzliche Angabe einer *ORDER BY*-Klausel eine definierte Reihenfolge hergestellt werden: Die Ergebnistabelle wird dann anhand der bei *ORDER BY* angegebenen Ausdrücke, die für jeden selektierten Datensatz berechnet werden, aufsteigend (*ASC*, Standardeinstellung) oder absteigend (*DESC*) sortiert. Hierbei werden zunächst nur die Werte des ersten Ausdrucks berücksichtigt, bei gleichen Werten auch die Werte des zweiten Ausdrucks usw.

Statt eines Ausdrucks kann man auch eine ganze Zahl angeben; diese steht dann für den an der entsprechenden Position in der *SELECT*-Liste stehenden Ausdruck.

Beispiel 3.7.: Anzeigen der Tabelle *personal* sortiert nach Personalnummern:

```
SELECT * FROM personal
ORDER BY pid;
```



Beispiel 3.8.: Anzeige von Wohnorten und Personalnummern, sortiert zunächst nach Wohnort (aufsteigend) und dann nach Personalnummer (absteigend):

```
SELECT ort, pid FROM personal
ORDER BY 1, 2 DESC;
```



3.4.5. Joins

Joins dienen bekanntlich dazu, zwei oder mehr Tabellen nach gewissen Kriterien zu verknüpfen. Diese Verknüpfungskriterien kann man stets als Bedingungen in einer *WHERE*-Klausel angeben. Der große Nachteil an dieser Vorgehensweise ist jedoch, dass dadurch die Join-Bedingungen nicht explizit als solche zu erkennen sind (die Abfrage kann ja auch noch andere Bedingungen enthalten).

Daher gibt es im *SQL*-Standard die Möglichkeit, Joins in einer Abfrage mit einer *JOIN*-Klausel explizit zu spezifizieren.⁵ Die Syntax einer sol-

⁵ Im *SQL*:2003-Standard ist dies als Feature F401 (“Extended joined table”) bezeichnet.

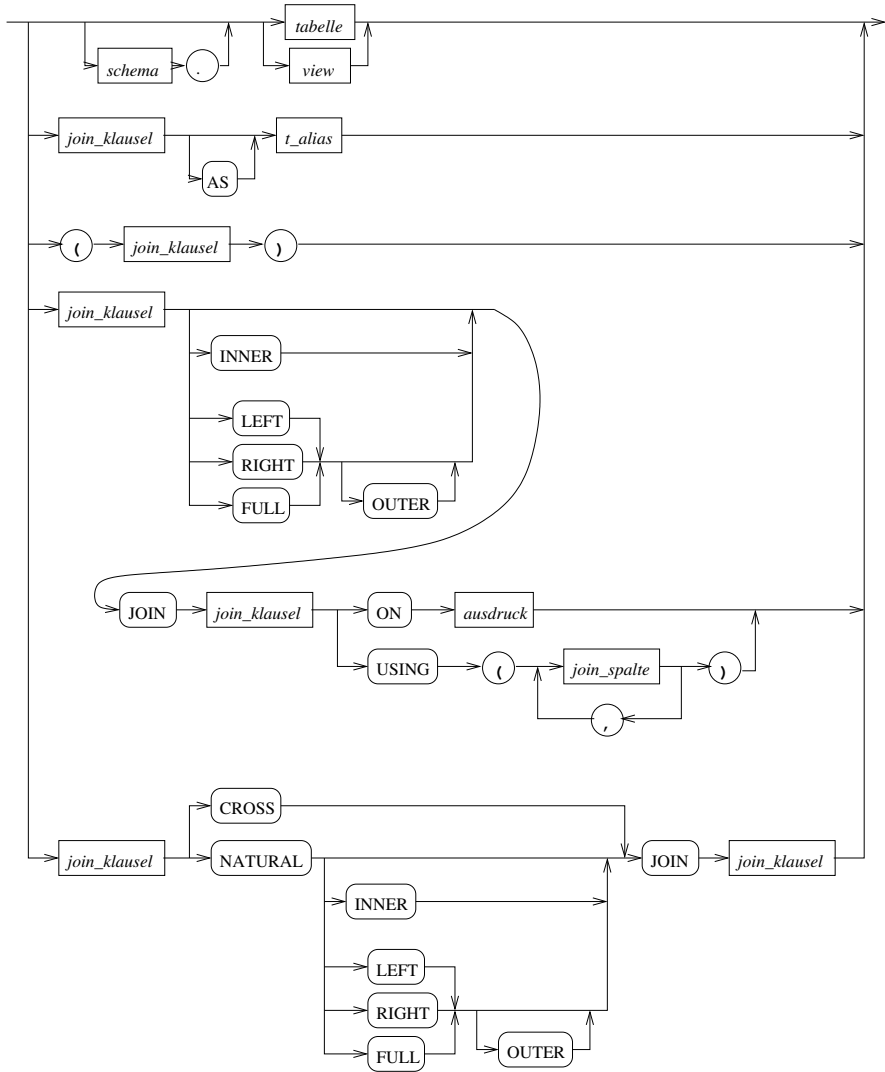


Abb. 22: Syntax einer JOIN-Klausel

chen JOIN-Klausel ist in Abb. 22 dargestellt.

Grundsätzlich liefert eine JOIN-Klausel eine Tabelle, auf der dann die SELECT-Abfrage – bzw. weitere Joins – operieren. Der triviale Fall einer JOIN-Klausel ist die bloße Angabe eines Tabellennamens; das Ergebnis einer solchen JOIN-Klausel ist natürlich die Tabelle selbst. Durch Angabe eines `t_alias` kann die Ergebnistabelle mit einem eigenen Namen versehen werden, unter dem sie in den übrigen Teilen der

SELECT-Abfrage angesprochen wird.

JOIN-Klauseln können geklammert werden, um bei Joins mit mehr als zwei Tabellen die Abarbeitungsreihenfolge festzulegen bzw. klarzumachen. Joins werden grundsätzlich jeweils zwischen zwei Tabellen ausgeführt; die betreffende Ergebnistabelle wird dann ggf. in einem weiteren Join verwendet. Sind *JOIN*-Klauseln nicht geklammert, erfolgt die Abarbeitung grundsätzlich von links nach rechts.

Ein *CROSS JOIN* ist nichts anderes als das kartesische Produkt von zwei Tabellen (siehe Abschnitt 2.4.1); statt Tabellen mit *CROSS JOIN* in der *FROM*-Liste zu verknüpfen, kann man sie alternativ auch durch Kommata trennen.

Beispiel 3.9.: Die Abfragen

```
SELECT * FROM personal, abteilung;  
und
```

```
SELECT * FROM personal CROSS JOIN abteilung;  
liefern das gleiche (nicht besonders sinnvolle) Ergebnis.
```

□

Wir wollen uns nun den weniger trivialen Komponenten einer *JOIN*-Klausel zuwenden.

Inner Joins entsprechen exakt den in Abschnitt 2.4.1 besprochenen Verbindungen von Tabellen, d. h. aus dem kartesischen Produkt von zwei Tabellen werden die Datensätze selektiert, die eine spezifische Join-Bedingung erfüllen.

Diese Bedingung kann explizit durch einen SQL-Ausdruck, der ein logisches Ergebnis liefert, angegeben werden; eine entsprechende *JOIN*-Klausel hat dann das Format

$$T_1 \text{ INNER JOIN } T_2 \text{ ON } \textit{bedingung}$$

Das Schlüsselwort *INNER* kann in *JOIN*-Klauseln übrigens stets weggelassen werden; man kann also ebenso gut schreiben

$$T_1 \text{ JOIN } T_2 \text{ ON } \textit{bedingung}$$

Wir werden im folgenden immer diese verkürzte Form verwenden.

Die Angabe einer Join-Bedingung in einem *Equi-Join* kann allerdings in wesentlich kürzerer Form erfolgen, wenn die Spalten, die in der Join-Bedingung vorkommen, in beiden Tabellen den gleichen Namen haben: Nehmen wir an, dass T_1 und T_2 jeweils Tabellenspalten mit den Namen s_1, s_2, \dots, s_n enthalten. Dann liefert die *JOIN*-Klausel

$$T_1 \text{ JOIN } T_2 \text{ USING } (s_1, s_2, \dots, s_n)$$

die gleiche Ergebnistabelle wie

T_1 JOIN T_2 ON

$T_1.s_1 = T_2.s_1$ AND $T_1.s_2 = T_2.s_2$ AND ... $T_1.s_n = T_2.s_n$

allerdings mit dem – sehr erwünschten – Unterschied, dass bei der USING-Form die Spalten s_1, s_2, \dots, s_n nur einmal in der Ergebnistabelle vorkommen, während bei der ON-Form diese Spalten jeweils doppelt vertreten sind, da sie ja in T_1 und T_2 vorkommen. Um von der komfortablen USING-Form zu profitieren, muss man allerdings schon beim Datenbank-Entwurf darauf achten, dass Attribute (Spalten), die in einem Equi-Join verwendet werden sollen, jeweils den gleichen Namen haben.

Natural Inner Joins sind ein Sonderfall der Inner Joins in der USING-Form: Hier werden *alle* in T_1 und T_2 gleichen Spaltennamen zur Konstruktion des Equi-Join verwendet. Sind also s_1, s_2, \dots, s_n die Namen *aller* Spalten, die sowohl in T_1 als auch in T_2 vorkommen, so entspricht

T_1 NATURAL JOIN T_2

der Klausel

T_1 JOIN T_2 USING (s_1, s_2, \dots, s_n)

Beispiel 3.10.: Wir wollen einen Equi-Join der Tabellen PERSONAL und ZUORDNUNG über das gemeinsame Attribut pid konstruieren. Die entsprechende Verbindung der beiden Tabellen ist in Abb. 23 dargestellt. (Wir haben in die Darstellung der PERSONAL-Tabelle aus Übersichtsgründen nur die Spalten pid und nachname aufgenommen.)

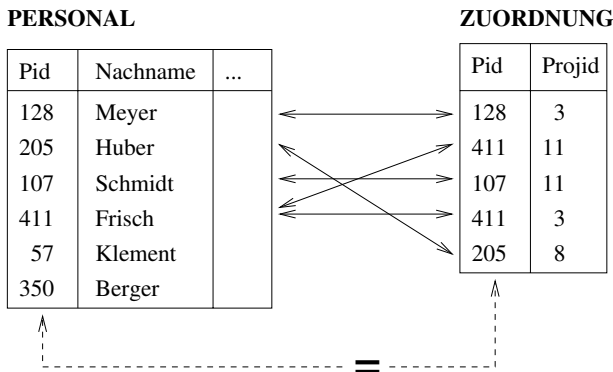


Abb. 23: Konstruktion eines Equi-Join der Tabellen PERSONAL und ZUORDNUNG

Mit einer expliziten Join-Bedingung lautet die Abfrage⁶

```
SELECT * FROM personal JOIN zuordnung
      ON personal.pid = zuordnung.pid;
```

und liefert das in Abb. 24 dargestellte Ergebnis.

PERSONAL Pid	Nachname	...	ZUORDNUNG Pid	Projid
128	Meyer		128	3
411	Frisch		411	11
107	Schmidt		107	11
411	Frisch		411	3
205	Huber		205	8

Abb. 24: Ergebnis des Equi-Join mit expliziter Join-Bedingung

Unter Ausnutzung der gleichen Spaltennamen kann man aber die Abfrage auch so formulieren:

```
SELECT * FROM personal JOIN zuordnung USING (pid);
```

Man erhält das in Abb. 25 gezeigte Ergebnis, das sich von dem vorhergehenden Resultat nur durch die nicht mehr doppelt vorhandene Spalte pid unterscheidet.

Pid	Nachname	...	Projid
128	Meyer		3
411	Frisch		11
107	Schmidt		11
411	Frisch		3
205	Huber		8

Abb. 25: Ergebnis des Equi-Join
mit USING bzw. NATURAL JOIN

Da die für den Equi-Join verwendeten Spalten die einzigen mit gleichem Namen in PERSONAL und ZUORDNUNG sind, können wir die Abfrage äquivalent als Natural-Join schreiben:

⁶ In einem Datenbanksystem, das keine JOIN-Klauseln kennt, müsste man die Abfrage mit einer gewöhnlichen WHERE-Klausel durchführen:

```
SELECT * FROM personal, zuordnung
WHERE personal.pid = zuordnung.pid;
```



```
SELECT * FROM personal NATURAL JOIN zuordnung;
```

Man sieht hier sehr deutlich, dass sich entsprechende Abfragen bei Ausnutzung der durch die Tabellenstruktur festgelegten Gegebenheiten wesentlich kürzer und prägnanter formulieren lassen. □

Das folgende Beispiel zeigt die Konstruktion eines Natural-Join mit drei Tabellen.

Beispiel 3.11.: Es soll eine Liste erzeugt werden, die die Zuordnung von Mitarbeiternamen zu Projektnamen darstellt. Jede Zeile der Ergebnistabelle soll also den Nachnamen eines Mitarbeiters und den Namen eines Projekts, dem dieser Mitarbeiter zugeordnet ist, enthalten. Da hier die Tabellen jeweils über die gleichen Spaltennamen verknüpft werden können, lautet die Abfrage einfach:

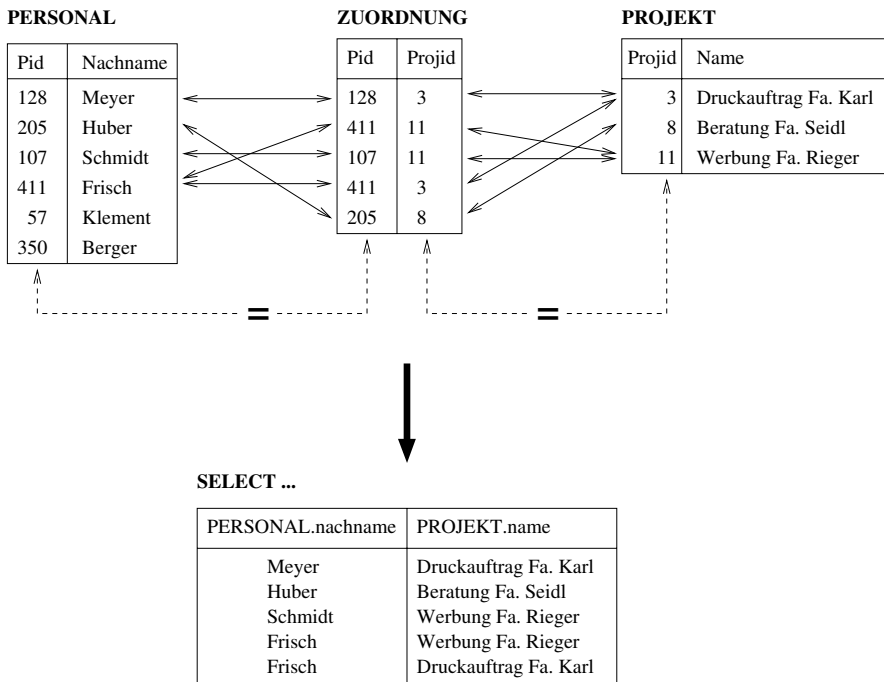
```
SELECT personal.nachname, projekt.name
FROM personal NATURAL JOIN zuordnung
      NATURAL JOIN projekt;
```

Wir haben in dieser Abfrage in der Select-Liste die Spaltennamen mit den Tabellennamen qualifiziert, um klarzumachen, woher die Attribute stammen. Da die Spaltennamen jedoch hier eindeutig sind, hätten wir auch nur die Spaltennamen angeben können. Die Konstruktion des Join ist mit den relevanten Spalten der beteiligten Tabellen in Abb. 26 gezeigt.

Obwohl die Syntax der obigen Abfrage den Eindruck erweckt, dass die drei Tabellen gleichzeitig miteinander verknüpft werden (das Resultat der gesamten Abfrage entspricht auch wirklich dieser Vorstellung), wird bei der Ausführung der Abfrage tatsächlich erst PERSONAL mit ZUORDNUNG und dann das Resultat mit PROJEKT verknüpft, da die Joins, wenn sie nicht geklammert sind, von links nach rechts abgearbeitet werden. Wollen wir erreichen, dass zunächst ZUORDNUNG mit PROJEKT und das Resultat dann mit PERSONAL verknüpft wird, müssen wir Klammern setzen:

```
SELECT personal.nachname, projekt.name
FROM personal NATURAL JOIN
      (zuordnung NATURAL JOIN projekt);
```

Das Ergebnis ist aber dasselbe wie bei der ungeklammerten JOIN-Klausel. Das ist nicht weiter verwunderlich, da es bei Inner Joins für das Ergebnis nicht auf die Ausführungsreihenfolge ankommt (diese kann aber wohl Auswirkungen auf die Effizienz der Abfrage haben). □

**Abb. 26:** Konstruktion eines Equi-Join aus drei Tabellen

Outer-Joins sind eine Erweiterung der bisher behandelten Inner Joins. Sie liefern zusätzlich noch die Zeilen einer Tabelle, die nicht mit einer Zeile der anderen Tabelle verknüpft werden können. Dadurch erreicht man, dass Zeilen einer Tabelle, die bei einem Inner Join nicht in die Ergebnistabelle aufgenommen würden, da sie keinen “Join-Partner” in der anderen Tabelle besitzen, doch in der Ergebnistabelle erscheinen.

Je nachdem, welche der beiden Tabellen vollständig in der Ergebnistabelle erscheinen soll, gibt es drei Ausprägungen eines Outer Join:

T_1 LEFT OUTER JOIN T_2

Es wird zunächst ein Inner Join ausgeführt. Für alle Datensätze von T_1 , die mangels eines Join-Partners in T_2 nicht in der Ergebnistabelle des Inner Join aufscheinen, wird ein zusätzlicher Datensatz erzeugt, der aus der betreffenden Zeile von T_1 und NULL-Werten für die Spalten von T_2 besteht. Die Ergebnistabelle enthält also in jedem Fall *alle* Datensätze von T_1 (der *linken* Tabelle).

T_1 RIGHT OUTER JOIN T_2

Es wird zunächst ein Inner Join ausgeführt. Für alle Datensätze von

T_2 , die mangels eines Join-Partners in T_1 nicht in der Ergebnistabelle des Inner Join aufscheinen, wird ein zusätzlicher Datensatz erzeugt, der aus der betreffenden Zeile von T_2 und NULL-Werten für die Spalten von T_1 besteht. Die Ergebnistabelle enthält also in jedem Fall *alle* Datensätze von T_2 (der *rechten* Tabelle).

T_1 FULL OUTER JOIN T_2

Dies ist im Prinzip die Kombination der beiden oberen Outer Joins: Es wird zunächst ein Inner Join ausgeführt. Für alle Datensätze von T_1 , die mangels eines Join-Partners in T_2 nicht in der Ergebnistabelle des Inner Join aufscheinen, wird ein zusätzlicher Datensatz erzeugt, der aus der betreffenden Zeile von T_1 und NULL-Werten für die Spalten von T_2 besteht. Außerdem wird für alle Datensätze von T_2 , die mangels eines Join-Partners in T_1 nicht in der Ergebnistabelle des Inner Join aufscheinen, ein zusätzlicher Datensatz erzeugt, der aus der betreffenden Zeile von T_2 und NULL-Werten für die Spalten von T_1 besteht. Die Ergebnistabelle enthält also in jedem Fall *alle* Datensätze von T_1 und T_2 (also die *vollen* Tabellen).

Das Schlüsselwort OUTER ist optional, da sich die Bedeutung schon aus den Schlüsselworten LEFT, RIGHT bzw. FULL ergibt. Wir werden es daher im folgenden weglassen.

Beispiel 3.12.: Bei dem Self-Join “Liste der Mitarbeiter mit Namen des Vorgesetzten”

```
SELECT r1.nachname, r2.nachname
FROM personal r1 JOIN personal r2
ON r1.vorges_id = r2.pid;
```

ergibt sich die in Abb. 27 dargestellte Ergebnistabelle. (Man beachte, dass hier unbedingt Tabellen-Aliase verwendet werden müssen, da bei einem Self-Join ja *eine* Tabelle *zweimal* in unabhängiger Weise verwendet wird.)

R1.NACHNAME	R2.NACHNAME
Meyer	Schmidt
Huber	Klement
Schmidt	Berger
Frisch	Schmidt
Klement	Berger

Abb. 27: Ergebnistabelle eines Inner Self-Join

Da der Firmenchef keinen Vorgesetzten hat, taucht er nicht in der ersten Spalte der Ergebnistabelle auf. Möchte man dies jedoch haben,

muss man einen Outer-Join verwenden:

```
SELECT r1.nachname, r2.nachname
FROM personal r1 LEFT JOIN personal r2
ON r1.vorges_id = r2.pid;
```

Damit erhält man die in Abb. 28a) gezeigte Ergebnistabelle.

R1.NACHNAME	R2.NACHNAME
Meyer	Schmidt
Huber	Klement
Schmidt	Berger
Frisch	Schmidt
Klement	Berger
Berger	

a)

R1.NACHNAME	R2.NACHNAME
Huber	Klement
Meyer	Schmidt
Frisch	Schmidt
	Meyer
	Huber
Schmidt	Berger
Klement	Berger
	Frisch

b)

Abb. 28: Ergebnistabellen von Outer-Joins

An dieser Stelle sei darauf hingewiesen, dass es grundsätzlich nicht gleichgültig ist, ob ein “linker” oder “rechter” Outer Join durchgeführt wird. Die Abfrage

```
SELECT r1.nachname, r2.nachname
FROM personal r1 RIGHT JOIN personal r2
ON r1.vorges_id = r2.pid;
```

liefert ein ganz anderes Resultat als der LEFT JOIN, nämlich das in Abb. 28b) Gezeigte.

□

Das folgende Beispiel illustriert die Verwendung von Outer Joins in Verbindung mit Aggregationsoperationen.

Beispiel 3.13.: Es soll eine Liste mit Personalnummern von Mitarbeitern und die Anzahl der Projekte, in denen sie tätig sind, ermittelt werden. Die Anweisung

```
SELECT pid, count(projid) FROM zuordnung
GROUP BY pid;
```

enthält nur Einträge für Mitarbeiter, die überhaupt in irgendeinem Projekt tätig sind (Abb. 29a)).

Will man hingegen die Information für alle Mitarbeiter in der Liste haben, muss man einen Outer-Join unter zusätzlicher Verwendung der Tabelle PERSONAL, die ja alle Personalnummern enthält, vornehmen (Abb. 30):

PID	COUNT(PROJID)
107	1
128	1
205	1
411	2

a)

PERSONAL.PID	COUNT(PROJID)
57	0
107	1
128	1
205	1
350	0
411	2

b)

Abb. 29: Inner- und Outer Join mit mehreren Tabellen

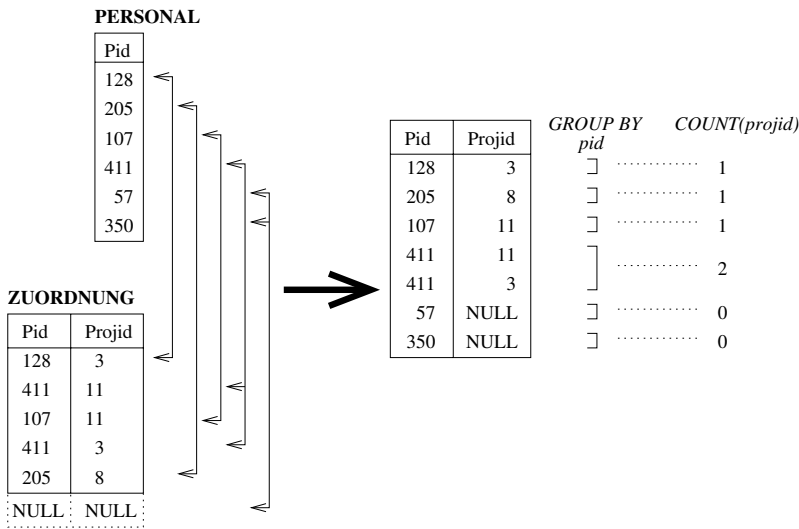


Abb. 30: Konstruktion eines Outer-Join mit Aggregationsoperationen

```
SELECT pid, COUNT(projid)
FROM personal NATURAL LEFT JOIN zuordnung
GROUP by pid;
```

Das Ergebnis dieses Outer Join findet man in Abb. 29b).



3.4.6. Subquers (Unterabfragen)

Innerhalb mancher SQL-Anweisungen können SELECT-Anweisungen zum Beschaffen von Werten verwendet werden. In diesem Fall bezeichnet man die SELECT-Anweisung als **Subquery**.

Wir verwenden zunächst Unterabfragen, um Werte für Bedingungen in WHERE- und HAVING-Klauseln von SELECT-Statements zu erhalten.

Beispiel 3.14.: Wie heißen die Mitarbeiter, die im Außendienst tätig sind?

```
SELECT nachname FROM personal
WHERE aid = (
    SELECT aid FROM abteilung
    WHERE bezeichnung = 'Außendienst'
);
```

□

Wird eine Subquery direkt mit einem Vergleichsoperator kombiniert, muss sichergestellt sein, dass sie genau *ein* Ergebnis liefert. Da dies im Zweifelsfall nicht genau vorhersehbar ist, empfiehlt sich besser die Verwendung von ANY, ALL oder IN.

Beispiel 3.15.: Welche Mitarbeiter arbeiten in einer Abteilung, die sich in München befindet?

```
SELECT nachname FROM personal
WHERE aid = ANY (
    SELECT aid FROM abteilung WHERE ort='München'
);
```

□

Beispiel 3.16.: Die obigen Beispiele sind noch nicht motivierend für die Notwendigkeit von Unterabfragen, denn sie lassen sich auch als Join wie folgt darstellen:

```
SELECT nachname
FROM personal JOIN abteilung USING (aid)
WHERE bezeichnung = 'Außendienst';
```

bzw.

```
SELECT nachname
FROM personal JOIN abteilung USING (aid)
WHERE abteilung.ort = 'München';
```

□

Das folgende Beispiel lässt sich jedoch nicht mehr nur durch einen Join lösen.

Beispiel 3.17.: Bestimmung der Nummer der Abteilung mit den meisten Mitarbeitern:

```
SELECT aid FROM personal
GROUP BY aid
HAVING COUNT(pid) >= ALL (
    SELECT COUNT(pid) FROM personal
```

```
GROUP BY aid
);
```

In der Unterabfrage werden zunächst die Mitarbeiterzahlen in den einzelnen Abteilungen bestimmt; für die Hauptabfrage interessiert nur diejenige Abteilung, deren Mitarbeiterzahl größer oder gleich allen Mitarbeiterzahlen in den Abteilungen ist. □

Die eben beschriebene Art von Subquery wird bei Ausführung der umschließenden Anweisung genau einmal ausgeführt.

Korrelierte Subqueries werden hingegen für jeden Datensatz, den das die Subquery umschließende Statement bearbeitet, ausgeführt.

Eine korrelierte Subquery liegt dann vor, wenn in der Subquery eine Tabelle der Hauptquery verwendet wird, die nicht in der Tabellenliste der Subquery vorkommt. Liegt etwa eine SELECT-Abfrage der folgenden Form vor:

```
SELECT ... FROM T1
WHERE T1.a IN (
    SELECT ... FROM T2
    WHERE T1.b = T2.c
);
```

so ist die Subquery korreliert, da innerhalb der Subquery Bezug auf die Tabelle T_1 genommen wird, die jedoch in der FROM-Liste der Subquery nicht vorkommt (die Namen T_1 und T_2 seien hierbei verschieden). Folglich ist T_1 die Tabelle aus der Hauptquery; dort kommt sie auch in der FROM-Liste vor.

Beispiel 3.18.: Für jede Abteilung soll der Abteilungsname zusammen mit dem Mitarbeiter mit der höchsten Personalnummer ausgegeben werden:

```
SELECT bezeichnung, nachname
FROM personal p JOIN abteilung USING (aid)
WHERE p.pid = (
    SELECT MAX(pid) FROM personal
    WHERE p.aid = aid
);
```

□

Zu beachten ist, dass innerhalb einer Subquery keine ORDER BY-Klausel vorkommen darf. Diese wäre ohnehin sinnlos, da die Resultate einer Subquery ja nicht direkt zur Ausgabe verwendet werden.

Auf eine Subquery kann der Operator **EXISTS** angewendet werden. Damit kann man überprüfen, ob eine Subquery mindestens einen Datensatz liefert: **EXISTS**(*subquery*) ist genau dann wahr, wenn *subquery* mindestens einen Datensatz liefert. Die Unentbehrlichkeit dieses Operators für bestimmte Abfragen wird im nächsten Abschnitt deutlich.

3.4.7. Positiv- und Negativabfragen mit den Operatoren **IN** und **EXISTS**

Bei den bis jetzt formulierten Abfragen hatten wir uns für Daten interessiert, für die Datensätze mit bestimmten Eigenschaften existieren.

Beispiel 3.19.: Mitarbeiter, die dem Projekt Nr. 11 zugeordnet sind:

```
SELECT nachname FROM personal NATURAL JOIN zuordnung
WHERE projid = 11;
```



Wir bezeichnen diese Abfragen als **Positivabfragen**.

Abfragen, die einen Join enthalten, bei denen in der **SELECT**-Liste nur Attribute *einer* Tabelle verwendet werden, kann man auch durch Anwendung des **IN**- oder des **EXISTS**-Operators auf eine Subquery formulieren:

Beispiel 3.20.: Mitarbeiter, die dem Projekt Nr. 11 zugeordnet sind:

```
SELECT nachname FROM personal
WHERE pid IN (
    SELECT pid FROM zuordnung
    WHERE projid = 11
);
```

oder

```
SELECT nachname FROM personal
WHERE EXISTS (
    SELECT pid FROM zuordnung
    WHERE personal.pid = pid
    AND projid = 11
);
```



Wir stellen fest, dass die Variante mit **IN** keinen Join, sondern nur eine nicht-korrelierte Subquery enthält. Allerdings sind solche Abfragen möglicherweise weniger effizient, da die Subquery eventuell eine Liste mit vielen Einträgen aufbaut, die in der Hauptquery gar nicht benötigt werden. (Stellen Sie sich dazu nur einmal vor, dass

die PERSONAL- und die ZUORDNUNG-Tabelle jeweils aus mehreren tausend Einträgen bestehen.)

Die Variante mit EXISTS ist offensichtlich wesentlich umständlicher als die ursprüngliche Abfrage, denn sie enthält nun sowohl eine korrelierte Subquery als auch einen Join. Wir werden im folgenden feststellen, dass der EXISTS-Operator in negierter Form wesentlich nützlicher ist.

Sehen wir uns einmal an, was im obigen Beispiel passiert, wenn wir Informationen haben wollen, für die Datensätze mit geforderten Eigenschaften *nicht* existieren.

Beispiel 3.21.: Mitarbeiter, die *nicht* dem Projekt Nr. 11 zugeordnet sind: Hier könnte man zunächst auf die Idee kommen, einfach die Bedingung `projid = 11` zu negieren:

```
SELECT nachname FROM personal NATURAL JOIN zuordnung
WHERE projid <> 11;
```

Die Ausgabe besteht dann aus den Namen Meyer, Huber und Frisch. Wie man durch Inspektion der Beispieltabellen feststellt, ist Herr Frisch jedoch dem Projekt Nr. 11 zugeordnet. Außerdem fehlen in der Ergebnistabelle die Namen Klement und Berger, da diese keinem Projekt, also insbesondere auch nicht Nr. 11, zugeordnet sind. Offensichtlich leistet obige Abfrage also nicht das, was wir wollten. Was ist nun der Grund dafür, dass Herr Frisch, obwohl er dem Projekt Nr. 11 zugeordnet ist, trotzdem in die Ergebnistabelle aufgenommen wird? Machen wir uns nochmals klar, dass ein Datensatz der gejointen Tabellen genau dann für die Ergebnistabelle selektiert wird, wenn die Bedingung `projid <> 11` erfüllt ist. Die hier vorliegende Situation ist in Abb. 31 dargestellt (zur Konstruktion des Equi-Join zwischen PERSONAL und ZUORDNUNG siehe Abb. 26).

Pid	PERSONAL.Nachname	...	ZUORDNUNG.Projid
128	Meyer		3
205	Huber		8
107	Schmidt	...	11
411	Frisch		11
411	Frisch		3

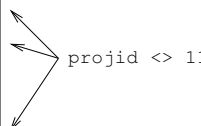


Abb. 31: Join mit Positivabfrage

Herr Frisch ist auch dem Projekt Nr. 3 zugeordnet; wir haben damit einen Datensatz, bei dem `projid <> 11` ist, und damit wird Herr

Frisch in die Ergebnistabelle aufgenommen. Das Fehlen der beiden anderen Namen in der Ergebnistabelle ist auch schnell aufgeklärt: Ein Join verbindet nur existierende Datensätze miteinander. Für **Klement** und **Berger** gibt es in ZUORDNUNG keine Datensätze, sie sind also überhaupt nicht im Join vorhanden.

Wir können dieses Problem also nicht mit einer Positivabfrage lösen, denn damit kann immer nur nach vorhandenen Datensätzen gefragt werden. Wir wollen aber eine Aussage über *nicht* vorhandene Datensätze.

Zur Lösung dieses Problems bieten sich zwei Arten an: Entweder man verwendet den IN- oder den EXISTS-Operator, jeweils in negierter Form:

```
SELECT nachname FROM personal
WHERE pid NOT IN (
    SELECT pid FROM zuordnung
    WHERE projid = 11
);
```

bzw.

```
SELECT nachname FROM personal
WHERE NOT EXISTS (
    SELECT pid FROM zuordnung
    WHERE personal.pid = pid
    AND projid = 11
);
```

□

Die Variante mit NOT EXISTS ist im allgemeinen der Variante mit NOT IN vorzuziehen, obwohl sie umständlicher zu formulieren ist. Grund dafür ist wieder der Aufbau einer Liste mit vielen nicht benötigten Einträgen bei der NOT IN-Variante.

Abfragen, bei denen Daten ermittelt werden sollen, für die *keine* Datensätze mit bestimmten Eigenschaften existieren, nennt man **Negativabfragen**.

Dazu abschließend noch ein weiteres Beispiel.

Beispiel 3.22.: Ermitteln aller Mitarbeiter, die überhaupt keinem Projekt zugeordnet sind:

```
SELECT nachname FROM personal
WHERE NOT EXISTS (
    SELECT projid FROM zuordnung
```

```
WHERE personal.pid = pid  
);
```



3.5. Virtuelle Tabellen (Views)

Eine **View** (zu deutsch: Sicht) ist eine virtuelle, dynamische Tabelle, über die auf Daten in anderen Tabellen (“Basistabellen”) oder weiteren Views zugegriffen werden kann. Eine View verhält sich in vielerlei Hinsicht wie eine “echte” Tabelle (und kann dann auch statt dieser verwendet werden). Views enthalten selbst keine Daten, sondern eine SELECT-Anweisung, die Informationen aus den Basistabellen oder weiteren Views abrufen (Virtualität). Die mit einer View assoziierte virtuelle Tabelle ist dann die Ergebnistabelle dieser SELECT-Anweisung. Da die eine View definierende SELECT-Anweisung erst durchgeführt wird, wenn auf die View (z. B. mit einer Datenbankabfrage) zugegriffen wird, wirken sich Änderungen an den Basistabellen stets auch auf die View aus (Dynamik).

Die Vorteile der Verwendung von Views sind:

- Vereinfachung von komplexen Datenstrukturen: Eine View verhält sich wie *eine* Tabelle, auch wenn die View mehrere Basistabellen zur Konstruktion der Ergebnistabelle benötigt.
- Darstellung von Daten unter anderen Gesichtspunkten: Mit Hilfe von Views können dynamisch Resultate aus Spalten von Basistabellen errechnet werden, ohne die Definition der Basistabellen zu ändern.
- Datenschutz innerhalb von Tabellen: Durch Verwendung einer View kann man den Zugriff auf bestimmte Zeilen und Spalten von Basistabellen beschränken. Auf diesen Aspekt wird in Abschnitt 5.4 genauer eingegangen.
- Vereinfachung der Datenbankprogrammierung: Komplizierte SQL-Statements können durch die Verwendung von Views (“Hilfsviews”) in einfachere Teile gegliedert werden.

Eine View wird mit dem CREATE VIEW-Statement erzeugt, das Syntaxdiagramm dazu ist in Abb. 32 dargestellt.

Hierbei bedeutet:

view Name der View

s_alias Namen für die durch die SELECT-Anweisung selektierten Spalten. Dies kann weggelassen werden, wenn in der SELECT-Liste ausschließlich Tabellenspalten als Ausdrücke verwendet werden. Ansonsten muss für alle selektierten Spalten ein

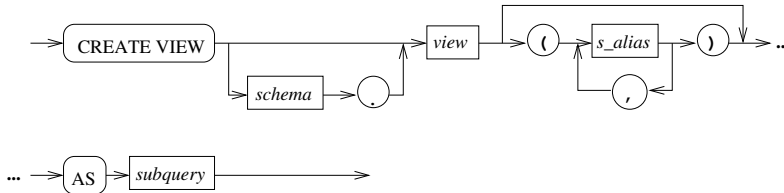


Abb. 32: Syntax des CREATE VIEW-Statements

Name vergeben werden (z. B. wenn in der SELECT-Liste eine Funktion verwendet wird).

Beispiel 3.23.: Erzeugen einer View mit dem Namen `mmit`, die Personalnummer, den Namen und den Vornamen aller in München ansässigen Mitarbeiter liefert:

```
CREATE VIEW mmit AS
  SELECT pid, nachname, vorname FROM personal
  WHERE ort = 'München';
```

Verwenden dieser View zum Erstellen einer Liste der Mitarbeiter, die in München wohnen und eine Personalnummer größer als 300 haben:

```
SELECT nachname FROM mmit
WHERE pid > 300;
```

□

Beispiel 3.24.: Erzeugen einer View `abt_anz`, die die Namen aller Abteilungen mit der Anzahl der dort beschäftigten Mitarbeiter liefert:

```
CREATE VIEW abt_anz (bez, anz) AS
  SELECT bezeichnung, COUNT(pid)
  FROM personal JOIN abteilung USING (aid)
  GROUP BY aid, bezeichnung;
```

Erstellen einer Liste der Abteilungen, in denen mindestens zwei Mitarbeiter arbeiten:

```
SELECT bez FROM abt_anz
WHERE anz >= 2;
```

□

Im Gegensatz zu SELECT-Abfragen, die nicht in der Datenbank gespeichert werden, erzeugt man durch CREATE VIEW ein Datenbankobjekt, das so lange erhalten bleibt, bis es explizit mit der DROP VIEW-Anweisung gelöscht wird.

Beispiel 3.25.: Löschen der zuvor erzeugten Views:

```
DROP VIEW mmit;
```


DEFAULT VALUES Damit wird ein Datensatz in die Tabelle eingefügt, der aus den bei der Definition der Tabelle (siehe Abschnitt 4.4.1) angegebenen DEFAULT-Werten besteht.

subquery Jede Zeile der Ergebnistabelle der Subquery wird in die Tabelle eingefügt.

Beispiel 3.26.: Ein neuer Mitarbeiter, Herr Michael Müller, wohnhaft in der Hauptstr. 5 in München, wird eingestellt. Er bekommt die Personalnummer 427, die restlichen Daten sind noch nicht bekannt:

```
INSERT INTO personal (pid,nachname,vorname,strasse,ort)
VALUES(427, 'Müller', 'Michael', 'Hauptstr. 5',
      'München');
```

Nach dieser Einfügeoperation liefert die Abfrage

```
SELECT * FROM personal WHERE pid=427;
```

das Ergebnis

```
(427, 'Müller', 'Michael', 'Hauptstr. 5',
'München', NULL, NULL, NULL, NULL)
```

Man hätte den Datensatz aus obigem Beispiel auch durch explizites Einsetzen von NULL-Werten in die VALUES-Liste formulieren können, wodurch man sich die Spaltenliste erspart hätte:

```
INSERT INTO personal
VALUES(427, 'Müller', 'Michael', 'Hauptstr. 5',
      'München', NULL, NULL, NULL, NULL);
```

Nachteil an dieser Variante ist, dass bei der Tabellendefinition (siehe Abschnitt 4.4.1) festgelegte Standardwerte für nicht spezifizierte Tabellenspalten nicht berücksichtigt werden.



Das Einfügen von Werten in eine Tabelle ist nicht nur direkt durch Angabe einer VALUES-Klausel möglich, sondern die Werte können auch aus anderen schon vorhandenen Werten in der Datenbank ermittelt werden.

Beispiel 3.27.: Angenommen, in unserer Musterdatenbank gibt es eine leere Tabelle **ORTE**, die aus einer Zeichenketten-Spalte besteht. Um die Wohnorte der Mitarbeiter in diese Tabelle aufzunehmen, benötigt man die Anweisung

```
INSERT INTO orte
SELECT DISTINCT ort FROM personal;
```

Man beachte die Angabe von DISTINCT in der Subquery, damit jeder Ort nur einmal aufgenommen wird.

□

Bei Datenbanksystemen, die die Modifizierung von Views unterstützen, kann eine INSERT INTO-Operation auch auf Views angewendet werden. Diese Operation ändert dann in Wirklichkeit die Basistabelle, auf die sich die View stützt. Damit eine View modifizierbar ist, muss sie allerdings eine bestimmte Struktur aufweisen – im Prinzip dergestalt, dass in eine View einzufügende Datensätze eindeutig auf Datensätze in der Basistabelle abgebildet werden können. Damit dürfen modifizierbare Views keine Konstrukte wie Joins, Mengenoperatoren, GROUP BY-Klauseln, Aggregationsfunktionen sowie den DISTINCT-Operator enthalten.

In PostgreSQL ist INSERT INTO bei Views zunächst nicht möglich. Eine PostgreSQL-spezifische Methode, dies doch und mit einer größeren Flexibilität als im SQL-Standard zu ermöglichen, wird in Abschnitt 3.6.4 vorgestellt.

3.6.2. Ändern bereits vorhandener Daten

Mit der Anweisung UPDATE (Abb. 34) werden ein oder mehrere bereits vorhandene Datensätze in einer Tabelle geändert.

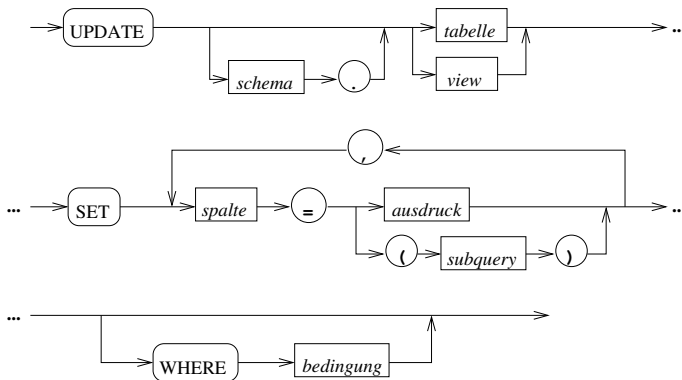


Abb. 34: Syntax des UPDATE-Statements

Hier bedeutet:

spalte Name der Spalte, die aktualisiert werden soll.

ausdruck legt den neuen Wert für die betreffende Spalte fest.

- subquery* Die Werte der Unterabfrage werden als Werte für die Spalten verwendet. Die Unterabfrage darf höchstens eine Ergebniszeile liefern; liefert sie keine, werden die betreffenden Spalten auf NULL gesetzt.
- WHERE** selektiert Datensätze, die aktualisiert werden sollen. Ist diese Klausel weggelassen, werden alle Datensätze aktualisiert.

Beispiel 3.28.: Die Abteilung ‘Produktion’ zieht nach Augsburg um:

```
UPDATE abteilung
SET ort = 'Augsburg'
WHERE bezeichnung = 'Produktion';
```

Danach liefert die Abfrage

```
SELECT * FROM abteilung
WHERE bezeichnung = 'Produktion';
```

das Ergebnis

(8, 'Produktion', 'Augsburg')



Beispiel 3.29.: Alle in München ansässigen Abteilungen ziehen nach Grasbrunn um:

```
UPDATE abteilung
SET ort = 'Grasbrunn'
WHERE ort = 'München';
```



Beispiel 3.30.: Alle Mitarbeiter, die im Projekt mit der Nr. 11 beschäftigt sind, bekommen den (im letzten Abschnitt eingefügten) Mitarbeiter mit dem Namen ‘Müller’ als Vorgesetzten:

```
UPDATE personal
SET vorges_id =
  (SELECT pid FROM personal WHERE nachname='Müller')
WHERE pid IN
  (SELECT pid FROM zuordnung WHERE projid = 11);
```

Achtung: Dies funktioniert nur dann, wenn es genau einen Mitarbeiter ‘Müller’ gibt!



Normalerweise wird das Ergebnis einer Subquery nur einmal berechnet. Verwendet die Subquery jedoch auch Spalten der zu aktualisierenden Tabelle, so wird die Subquery für jede zu aktualisierende Zeile ausgeführt (**korreliertes Update**).

Beispiel 3.31.: Wir formulieren die Abfrage aus dem vorhergehenden Beispiel als korreliertes Update:

```
UPDATE personal
SET vorgesch_id =
  (SELECT pid FROM personal WHERE nachname='Müller')
WHERE EXISTS (
  SELECT pid FROM zuordnung
  WHERE personal.pid = pid AND projid = 11
);
```

□

Für das Ändern von Daten in Views gelten die gleichen Einschränkungen wie bei INSERT INTO.

3.6.3. Löschen von Daten

Die Anweisung DELETE FROM (Abb. 35) löscht Datensätze aus einer Tabelle.

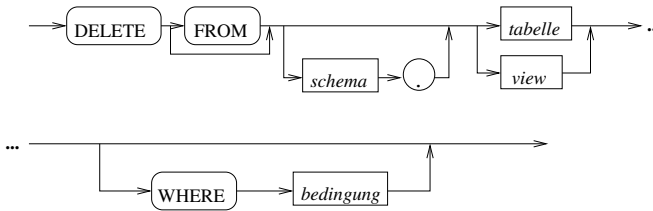


Abb. 35: Syntax des DELETE FROM-Statements

Hierbei bedeutet:

WHERE Bedingung für die Datensätze, die gelöscht werden sollen. Es werden nur die Datensätze gelöscht, bei denen die angegebene Bedingung den Wert TRUE liefert. Achtung: Wird die WHERE-Klausel weggelassen, so werden *alle* Datensätze aus der Tabelle gelöscht.

Beispiel 3.32.: Löschen aller Mitarbeiter aus der Tabelle PERSONAL, die in der Abteilung Nr. 5 arbeiten:

```
DELETE FROM personal
WHERE aid = 5;
```

□

Für das Löschen von Daten in Views gelten die gleichen Einschränkungen wie bei INSERT INTO.

3.6.4. Modifizierbare Views

Auf **modifizierbare Views** können INSERT-, UPDATE- oder DELETE-Operationen angewendet werden, um eine Modifikation der Daten in den Basistabellen der View vorzunehmen.

In PostgreSQL sind Views zunächst stets readonly-Objekte, d. h. es können zwar Auswahlabfragen (SELECT) mit einer View ausgeführt werden, aber keine Änderungsabfragen (INSERT, UPDATE, DELETE). PostgreSQL bietet jedoch die Möglichkeit, bei Ausführung von SQL-Auswahl- oder Änderungsabfragen über sog. **Regeln** weitere SQL-Abfragen vorzugeben, die zusätzlich zur oder statt der ursprünglichen Abfrage ausgeführt werden sollen. Die Syntax für die Definition einer neuen Regel ist in Abb. 36 dargestellt.

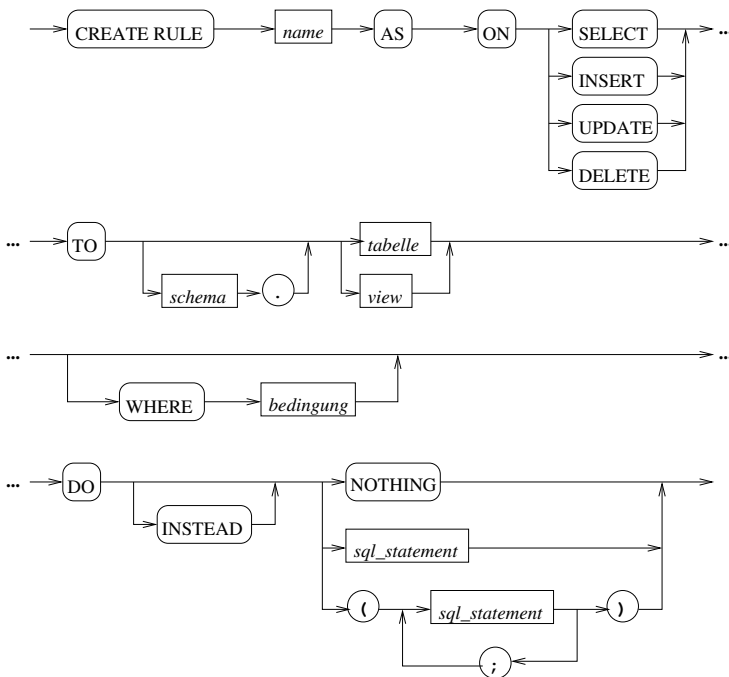


Abb. 36: Syntax des CREATE RULE-Statements

Hierbei bedeutet:

name Name der Regel, die definiert werden soll. Dieser muss pro Tabelle bzw. View eindeutig sein.

ON	Hinter diesem Schlüsselwort ist die Art der Abfrage (SELECT, INSERT, UPDATE oder DELETE) anzugeben, für die die Regel gelten soll.
TO	Es folgt der Name der Tabelle bzw. View, auf die sich die Regel bezieht.
WHERE	Die Regel wird nur dann angewandt, wenn die folgende Bedingung erfüllt ist.
INSTEAD	Die Regel ersetzt die durch die Ausgangsabfrage gegebene Aktion. Ist dieses Schlüsselwort nicht angegeben, wird die Regel zusätzlich ausgeführt.
NOTHING	Es wird keine Aktion ausgeführt.
<i>sql_statement</i>	Es kann ein oder eine Folge von SQL-Statements angegeben werden.

Wir behandeln das sehr mächtige Rule-System von PostgreSQL nur in dem Umfang, in dem es für die Realisierung von modifizierbaren Views erforderlich ist.

Innerhalb einer Regel hat man Zugriff auf zwei spezielle Pseudotabellen: Die Tabelle OLD (nur bei Regeln für UPDATE und DELETE) enthält den alten zu aktualisierenden bzw. zu löschenden Datensatz, die Tabelle NEW (nur bei Regeln für INSERT und UPDATE) enthält den neu einzufügenden oder aktualisierten Datensatz.

Beispiel 3.33.: Betrachten wir nochmals die auf Seite 59 definierte View `mmit`, die Personalnummer, Name und Vorname aller in München wohnenden Mitarbeiter liefert. Unter der Voraussetzung, dass Personalnummer, Name und Vorname jeden Mitarbeiter eindeutig identifizieren (was in der Praxis schon bei der Personalnummer alleine der Fall sein dürfte - siehe auch das Beispiel auf Seite 90), entspricht jedem Datensatz in der View `mmit` genau ein Datensatz in der Basistabelle `PERSONAL`. Diese View ist also modifizierbar. Damit sie auch in PostgreSQL modifizierbar wird, müssen wir folgende drei Regeln definieren:

```
CREATE RULE ins AS ON INSERT TO mmit DO INSTEAD
    INSERT INTO personal (pid,nachname,vorname)
        VALUES (NEW.pid,NEW.nachname,NEW.vorname);

CREATE RULE upd AS ON UPDATE TO mmit DO INSTEAD
UPDATE personal
    SET pid = NEW.pid,
        nachname = NEW.nachname,
        vorname = NEW.vorname
```

```
WHERE pid = OLD.pid
  AND nachname = OLD.nachname
  AND vorname = OLD.vorname;

CREATE RULE del AS ON DELETE TO mmit DO INSTEAD
DELETE FROM personal
WHERE pid = OLD.pid
  AND nachname = OLD.nachname
  AND vorname = OLD.vorname;
```



Man beachte, dass bei UPDATE- und DELETE-Regeln für Views *nur* Datensätze aktualisiert bzw. gelöscht werden können, die tatsächlich in der View enthalten sind, auch wenn sich die UPDATE- bzw. DELETE-Abfrage auf die gesamte Basistabelle bezieht. Dieses Verhalten ist semantisch durchaus wünschenswert: Modifikationsoperationen an einer View sollen sich ja nur auf die Datensätze in der View beziehen. Bei INSERT-Regeln haben wir diese Semantikgarantie nicht, da ja ein neuer Datensatz in die View eingefügt wird. Hier wäre etwa durch Angabe zusätzlicher Werte bei der INSERT-Operation auf der Basistabelle sicherzustellen, dass ein in die View eingefügter Datensatz nachher tatsächlich in der View enthalten ist (d. h. der neue Datensatz muss die Bedingungen der die View definierenden SELECT-Abfrage erfüllen).

Beispiel 3.34.: In unserem letzten Beispiel haben wir die Situation, dass etwa

```
INSERT INTO mmit VALUES (500, 'Vogel', 'Claudia');
```

einen Datensatz einfügt, aber dieser danach nicht in der View enthalten ist, da der Wohnort auf den NULL-Wert gesetzt wird. Wir sollten daher die INSERT-Regel besser wie folgt formulieren:

```
CREATE RULE ins AS ON INSERT TO mmit DO INSTEAD
  INSERT INTO personal (pid,nachname,vorname,ort)
    VALUES (NEW.pid,NEW.nachname,NEW.vorname,
      'München');
```



Definierte Regeln kann man mit DROP RULE wieder löschen.

Beispiel 3.35.: Die für die View mmit definierten Regeln können wir mit

```
DROP RULE ins ON mmit;
DROP RULE upd ON mmit;
```

DROP RULE del ON mmit;
löschen.



3.7. Ein- und Ausgabe von Datenbankinhalten

Datenbanken dienen zur effizienten Speicherung von Informationen. **Information** an sich ist abstrakt. Um Information *darzustellen*, benötigt man eine geeignete **Repräsentation**. Für viele Arten von Information ist die Repräsentation so selbstverständlich, dass man nicht mehr zwischen den beiden Begriffen differenziert.

Ein gutes Beispiel hierfür ist die Zahlendarstellung. Wir stellen z. B. die Zahl eintausendzweihundertvierunddreißig durch die Zeichenfolge
1234

dar und sprechen auch von der Zahl 1234. Dennoch kann man auch bei der Zahlendarstellung gut den Unterschied zwischen Information und Repräsentation erkennen. Betrachten wir die Zeichenfolgen

34.5

und

34.50

die, obwohl verschieden, beide den gleichen Zahlenwert repräsentieren. Ein- und dieselbe Information kann also verschiedene Repräsentationen haben.

Bei der Speicherung von Informationen in einer Rechanlage muss auch eine geeignete Repräsentation für die interne Darstellung gewählt werden. Bekanntlich können Rechanlagen nur Binärzahlen (also 0-1-Ziffernfolgen) darstellen. Da man in einem Datenbanksystem außer Zahlen auch andere Informationen (Zeichenketten, Datumsangaben, ...) speichern will, muss für alle diese Datentypen eine geeignete interne Repräsentation festgelegt werden. Eine nähere Behandlung dieser Thematik liegt jedoch außerhalb des Rahmens dieses Buches.

Von praktischer Bedeutung ist für uns mehr die externe Repräsentation von Datentypen, d. h. in welcher Form Werte eines Datentyps dem Benutzer präsentiert werden (und in der anderen Richtung, in welcher Form Werte angegeben werden müssen, die der Benutzer in die Datenbank eintragen will).

Bei der Ausgabe von Datenbankinhalten per SELECT-Statement müssen diese zur Repräsentation stets in Zeichenketten umgewandelt

werden, also z. B. die Zahl 1234 in die Zeichenkette aus dem Zeichen '1', gefolgt von '2', gefolgt von '3', gefolgt von '4'.

Da es – wie oben ausgeführt – für ein- und dieselbe Information durchaus verschiedene Repräsentationen geben kann, wählt das Datenbanksystem im konkreten Fall zunächst eine Standarddarstellung. Diese ist DBMS-spezifisch und nicht standardisiert. Andererseits gibt es in vielen DBMS auch die Möglichkeit, über entsprechende Umwandlungsfunktionen eine spezielle Darstellung zu erreichen. Die folgenden Ausführungen dazu beziehen sich auf PostgreSQL, sind aber in vielen DBMS in ähnlicher Form zu finden.

Beispiel 3.36.: Die Abfrage

```
SELECT pid FROM personal;
```

liefert erwartungsgemäß

128

205

107

411

57

350

wobei hier zu bemerken ist, dass die Ausgabe rechtsbündig (alle Hunderter-, Zehner- und Einerstellen übereinander) erfolgt. Will man hingegen eine Darstellung mit einer festen Stellenzahl und führenden Nullen haben, muss man sich der im folgenden besprochenen Konvertierungsfunktion bedienen.



Die Konvertierungsfunktion

```
TO_CHAR( $n$ ,  $f$ )
```

wandelt einen numerischen Wert n in eine Zeichenkette um, wobei die Formatzeichenkette (Formatstring) f angibt, wie die gelieferte Zeichenkette aussehen soll, die `TO_CHAR` zurückliefert. Für jeden Teil der Ausgabezeichenkette enthält f ein Formatzeichen, das die Belegung des entsprechenden Teils der Ausgabezeichenkette festlegt.

Die wichtigsten Formatzeichen sind:

- 9 steht für eine Stelle der Zahl. Steht auf der betreffenden Position eine führende Null, wird diese durch ein Leerzeichen dargestellt.
- 0 steht für eine Stelle der Zahl. Eine führende Null wird auch als Null dargestellt.

- D setzt an die betreffende Stelle den Dezimalpunkt. Falls ‘D’ nicht im Formatstring vorkommt, wird eine ganze Zahl in der Ausgabezeichenkette geliefert.

Beispiel 3.37.: Um die Darstellung der Personalnummern mit einer festen Stellenzahl von vier Stellen mit führenden Nullen zu erreichen, schreibt man

```
SELECT TO_CHAR(pid,'0000') FROM personal;
```

und erhält als Ausgabe

```
0128
0205
0107
0411
0057
0350
```



Man beachte, dass die Zahl der ‘0’- bzw. ‘9’-Stellen vor dem Dezimalpunkt auch die größte Zahl festlegt, die verarbeitet werden kann. Hat die umzuwandelnde Zahl vor dem Dezimalpunkt mehr Stellen als dafür im Formatstring reserviert sind, wird eine Zeichenkette bestehend aus ‘#’-Zeichen geliefert, um anzuzeigen, dass der Darstellungsbereich überschritten wurde. Hat hingegen eine Zahl hinter dem Dezimalpunkt mehr Stellen als im Formatstring angegeben sind, erfolgt eine kaufmännische Rundung auf die angegebene Stellenzahl.

Auch die Darstellung von Datums- und Zeitangaben kann man mit einer Konvertierungsfunktion in das gewünschte Format bringen. Diese Funktion ist ebenfalls

```
TO_CHAR(d, f)
```

wobei *d* eine Datums-, Zeit- oder Zeitraumangabe und *f* der Formatstring ist. *f* kann u. a. folgende Formatzeichen enthalten:

YYYY	4-stellige Jahreszahl
YYY bzw. YY bzw. Y	Die letzten 3 bzw. 2 bzw. 1 Stelle(n) der Jahreszahl
Q	Quartal (liefert ‘1’, ‘2’, ‘3’ oder ‘4’)
MM	Monat numerisch, 2-stellig (‘01’, ..., ‘12’)
Mon	Die ersten drei Zeichen des ausgeschriebenen Monatsnamens (sprachumgebungs-spezifisch)
DD	Tag des Monats numerisch, 2-stellig (‘01’, ..., ‘31’)
HH	Stunde, 2-stellig, 12er-Format (‘01’, ..., ‘12’)

HH24	Stunde, 2-stellig, 24er-Format ('00', ..., '23')
MI	Minute, 2-stellig ('00', ..., '59')
SS	Sekunde, 2-stellig ('00', ..., '59')
- / , . ; :	Diese Satzzeichen werden 1:1 in die Ausgabe übernommen
"text"	text wird 1:1 in die Ausgabe übernommen

Beispiel 3.38.: Mit

```
SELECT TO_CHAR(einstellung, 'DD. Mon YYYY')
FROM personal;
```

erhalten wir die Einstellungsdaten der Mitarbeiter in folgenden Format:

```
19. Jan 1994
27. May 1991
02. Nov 1990
14. Sep 1995
04. Oct 1990
28. May 1993
```



Bei der *Eingabe* von Datenbankinhalten (z. B. in einem INSERT-Statement) spielen Konvertierungsfunktionen für die numerischen und Zeichenketten-Datentypen keine besonders wichtige Rolle, da über die implizite Typkonversion (siehe Abschnitt 3.3.7) bereits alles automatisch erledigt wird. Hingegen hat eine Konvertierungsfunktion für Zeichenketten in Datums- oder Zeitangaben durchaus eine wichtige Berechtigung.

Obwohl PostgreSQL viel flexibler bei der Erkennung von Datums- und Zeitangaben ist, als es der SQL-Standard erfordert, kommt man in manchen Fällen nicht um eine explizite Konvertierung herum. Natürlich ist eine explizite Konvertierung auch dann nötig, wenn man gewährleisten will, dass Datums- und Zeitangaben in einem bestimmten Format vorgenommen werden. Die "inverse" Umwandlungsfunktion für Zeichenketten in Datumsangaben lautet

```
TO_DATE(s, f)
```

und wandelt eine Zeichenkette *s* im Format *f* (Formatzeichen wie oben) in einen Wert vom Typ `TIMESTAMP` (der dann implizit je nach Bedarf in einen Wert vom Typ `DATE` oder `TIME` ungewandelt werden kann) um.

Beispiel 3.39.: Um für den im Beispiel auf Seite 61 neu eingestellten Mitarbeiter Michael Müller das Einstellungsdatum auf den 10.02.2004 zu setzen, könnten wir folgende SQL-Abfrage verwenden:

```
UPDATE personal
SET einstellung='20040210' WHERE pid=427;
```

Verwendet man hingegen

```
UPDATE personal
SET einstellung='10.02.2004' WHERE pid=427;
```

so stellt man bei einem zur Kontrolle ausgeführten SELECT möglicherweise fest, dass hier – je nach den Konfigurationseinstellungen von PostgreSQL – der 02.10.2004 als Einstellungsdatum gesetzt wurde. Um zu erreichen, dass auch dieses Datumsformat richtig interpretiert wird, schreiben wir

```
UPDATE personal
SET einstellung=TO_DATE('10.02.2004', 'DD.MM.YYYY')
WHERE pid=427;
```

□

Sehr interessante Möglichkeiten hat man in PostgreSQL bei der Eingabe von Intervallen. Diese werden als Stringliteral als Folge von

zahl einheit

angegeben, wobei *einheit* die Werte **second**, **minute**, **hour**, **day**, **week**, **month** oder **year** (und die Pluralbildungen davon) haben kann. Ein Zeitintervall von 3 Tagen, 12 Stunden und 15 Minuten kann dann durch das Stringliteral

```
'3 days 12 hours 15 minutes'
```

dargestellt werden. Alternativ kann man den Stunden-, Minuten- und Sekundenanteil auch im Format

hh:mm:ss

angeben, so dass obiges Intervall auch durch das Stringliteral

```
'3 days 12:15:00'
```

repräsentiert wird.

Die Ausgabe von Intervallwerten erfolgt analog dem Eingabeformat; der Stunden-, Minuten- und Sekundenanteil wird stets in dem oben angegebenen Alternativformat dargestellt.

4. Datenbank-Entwurf

Im letzten Abschnitt haben wir mit einer bereits vorhandenen Datenbank (mit schon vorhandenen Tabellen) gearbeitet. In der Praxis muss jedoch vor der Benutzung einer Datenbank erst ihr Entwurf erfolgen. Wir wissen, dass alle Informationen in relationalen Datenbanken in Form von Tabellen (Relationen) gespeichert werden. Folglich versteht man unter dem Entwurf (engl.: Design) einer Datenbank die Definition geeigneter Tabellen, die die gewünschten Informationen speichern sollen.

Die anfallenden Informationen liegen jedoch in den meisten Fällen nicht direkt in Relationenform vor, oder zumindest nicht in einer solchen Form, die sich “gut” für die Darstellung in einer Datenbank eignet.

Was ist nun eine “gute” Form für ein Relationenschema (= Relationsname + Attribute) in einer Datenbank? Hier gibt es im wesentlichen zwei Beurteilungskriterien:

- Logische Ebene: Wie leicht kann der Benutzer das Relationenschema und die Bedeutung seiner Attribute interpretieren? Je leichter dem Benutzer die Interpretation fällt, umso leichter fällt ihm die Formulierung von Abfragen, und umso weniger Fehler wird er machen.
- Physikalische Ebene: Wie effizient wird die auf dem Schema aufgebaute Relation gespeichert, geändert und abgerufen?

Wir werden nun Probleme behandeln, die bei “schlechten” Relationenschemata in Datenbanken auftreten können.

4.1. Anomalien in Datenbanken

Wir betrachten zur Illustration das Relationenschema

BESTELLUNGEN (KUNDE, ORT, VERTRETER, MENGE)

das die bei einer Firma vorliegenden Bestellungen erfassen soll. Die Attribute bedeuten:

KUNDE	Name des Kunden (identifiziert eindeutig den Kunden)
ORT	Unternehmenssitz des Kunden
VERTRETER	Für den Kunden zuständiger Vertreter – abhängig vom Unternehmenssitz des Kunden
MENGE	Menge des bestellten Gutes (wir nehmen an, dass die Firma nur einen Artikel vertreibt)

Für jeden Kunden sei höchstens ein Eintrag in BESTELLUNGEN vorhanden (dann ist {KUNDE} Schlüssel von BESTELLUNGEN). Die Relation habe den in Abb. 37 dargestellten Inhalt.

KUNDE	ORT	VERTRETER	MENGE
Auer	Passau	1	9
Blank	Regensburg	2	20
Christ	München	3	3
Dorn	Passau	1	5

Abb. 37: Eine Relation mit Anomalien

Mit diesem Relationenschema können folgende Anomalien auftreten:

Änderungsanomalie: Ändert sich der zuständige Vertreter für das Gebiet Passau, müssen mehrere Tupel geändert werden, da die Zuordnung Vertreter – Ort mehrfach in der Tabelle abgelegt ist.

Lösung: Definiere zwei Tabellen

BESTELL1 (KUNDE, ORT, MENGE)

ZUST (ORT, VERTRETER)

so dass die Information, welcher Vertreter für welches Gebiet zuständig ist, nur noch einmal abgelegt wird.

Löschanomalie: Wird ein Kunde gelöscht, weil z. Zt. keine Bestellungen von ihm vorliegen, geht die Information über seinen Standort verloren.

Lösung: Wir zerlegen obige Tabelle BESTELL1 weiter in

BESTELLM (KUNDE, MENGE)

und

STANDORT (KUNDE, ORT)

Somit gehen bei Löschungen in BESTELLM keine Standortinformationen mehr verloren.

Einfüge-(Eintrage-)Anomalie: Ein neuer Datensatz kann nur dann in BESTELLUNGEN eingetragen werden, wenn alle vier Attributwerte bekannt sind (nach dem theoretischen Relationenmodell müssen bei einem Element der Relation alle Werte bekannt sein – NULL-Werte gibt es im theoretischen Relationenmodell nicht). In der Praxis wird

es ausreichen, wenn zumindest die Attributwerte aller Schlüssel bekannt sind (Schlüsselwerte dürfen nicht NULL sein) – die restlichen unbekannten Attributwerte können mit dem NULL-Wert modelliert werden. Jedoch ist die Relation $\text{VERTRETER} \leftrightarrow \text{ORT}$ auch für sich allein interessant.

Die beschriebenen Probleme lassen sich in vielen Fällen durch eine Normalisierung der Relationen verbessern, wenn auch nicht immer völlig lösen.

4.2. Normalformen von Relationen

Wir benötigen zum Verständnis der nun folgenden Definitionen die Konzepte von funktionaler Abhängigkeit sowie Schlüsseln, die bei der Vorstellung des Relationenmodells eingeführt wurden.

4.2.1. Minimale Abhängigkeitssysteme

Wir hatten bis jetzt funktionale Abhängigkeiten durch Analyse einer konkreten Relation bestimmt. In der Praxis ist es aber so, dass elementare funktionale Abhängigkeiten durch die beabsichtigte Bedeutung von Attributen vorgegeben sind. Es gibt nun Regeln, wie man aus funktionalen Abhängigkeiten weitere funktionale Abhängigkeiten gewinnt. Seien X , Y und Z Teilmengen der Attributmenge eines Relationenschemas R .

Regel I: Ist $Y \subseteq X$, so gilt $X \rightarrow Y$. (Reflexivität)

Regel II: Gilt $X \rightarrow Y$, so gilt auch $X \cup Z \rightarrow Y \cup Z$. (Augmentierung)

Regel III: Gilt $X \rightarrow Y$ und $Y \rightarrow Z$, so gilt auch $X \rightarrow Z$. (Transitivität)

Diese Regeln heißen **Armstrong'sche Inferenzregeln**. Es lässt sich zeigen, dass diese Regeln ausreichen, um aus einem gegebenen System von funktionalen Abhängigkeiten alle weiteren funktionalen Abhängigkeiten abzuleiten, die bezüglich des gegebenen Systems gültig sind.

Für ein System F von funktionalen Abhängigkeiten bezeichnen wir mit F^+ das System aller funktionalen Abhängigkeiten, die sich durch Anwendung der Inferenzregeln aus F ableiten lassen. F^+ heißt **Abschluss** von F .

Wenn man zwei verschiedene Systeme F und G von funktionalen Abhängigkeiten gegeben hat, kann man sich die Frage stellen, ob aus diesen beiden Systemen die gleichen Abhängigkeiten ableitbar sind,

d. h. ob $F^+ = G^+$ gilt. Ist dies der Fall, bezeichnen wir F und G als **äquivalent**.

Für ein gegebenes Relationenschema ist man an einem System F von funktionalen Abhängigkeiten interessiert, das keine “überflüssige” Information enthält, das also **minimal** in folgendem Sinne ist:

- (1) Für jede Abhängigkeit $X \rightarrow Y$ in F gilt, dass Y aus genau einem Attribut besteht.
- (2) F enthält keine redundanten Abhängigkeiten, d. h. es gibt keine Abhängigkeit, die aus den anderen Abhängigkeiten in F (durch Anwendung der Inferenzregeln) ableitbar ist.
- (3) Eine Abhängigkeit $X \rightarrow A$ in F kann nicht durch $Y \rightarrow A$ mit $Y \subseteq X$, $Y \neq X$, ersetzt werden, ohne dass sich die Menge der ableitbaren Abhängigkeiten ändert.

Die Forderung (1) bewirkt die Entstehung vieler “kleiner” Abhängigkeiten. Wir fassen diese zur kürzeren Notation folgendermaßen zusammen: (Hierbei sei F minimal gemäß obiger Definition.) Sind

$$X \rightarrow A_1, \dots, X \rightarrow A_{k_X}$$

alle Abhängigkeiten in F mit X auf der linken Seite, so ersetzen wir diese durch eine Abhängigkeit

$$X \rightarrow A_1 \cup \dots \cup A_{k_X}$$

und erhalten damit offensichtlich ein äquivalentes System von funktionalen Abhängigkeiten. Diese Ersetzung führt man für alle X durch, die auf der linken Seite (mindestens) einer Abhängigkeit in F vorkommen. Man erhält ein System F_k , das dem Minimalsystem F entspricht, aber kompakter zu schreiben ist. Wir werden in diesem Text, wenn wir ein minimales Abhängigkeitssystem konkret angeben, immer die kompakte Form verwenden, meinen dies aber nur als Abkürzung für die minimale Form.

Ein minimales System von Abhängigkeiten eines Relationenschemas kann in der Praxis immer durch die vorgesehene Bedeutung (Semantik) der Attribute hergeleitet werden.

4.2.2. Erste Normalform

Beispiel 4.1.: Betrachten wir die in Abb. 38 dargestellte Relation FREIZEIT.

Das Attribut HOBBIES hat Mengen von Tupeln als Werte, ist also selber wieder eine Relation. Relationen dieser Form werden von Daten-

NR	NAME	HOBBIES(HNAME,PRIOR)
17	Beyer	{(Radfahren,3), (Musikhören,2), (Schwimmen,1)}
22	Schneider	{(Theater,1), (Reisen,1)}
9	Mitterer	{(Wandern,1), (Schwimmen,2)}
63	Schmitt	{(Radfahren,1)}

Abb. 38: Eine Relation mit nichtatomaren Attributen

banksystemen im allgemeinen nicht unterstützt; daher sollen Relationen, die “einfache” Attributwerte haben, besonders ausgezeichnet werden.

□

Ein Relationenschema ist in **erster Normalform**, wenn alle Attribute atomare Werte haben, d. h. Attributwerte keine Tupel oder Mengen sind.

Oben vorgestelltes Relationenschema ist also sicher nicht in erster Normalform. Da jedoch in der Praxis durchaus Relationen dieser Form auftreten, ist es angebracht, sich anzusehen, wie man Relationenschemata, die nicht in erster Normalform vorliegen, in die erste Normalform überführt.

Hierzu dient folgendes Verfahren:

- Identifiziere einen Schlüssel für das Ausgangs-Relationenschema R .
- Bilde ein Relationenschema F , das alle Attribute von R enthält, die atomare Werte haben.
- Für jedes Attribut A , das selbst wieder ein Relationenschema ist, identifiziert man zunächst einen **partiellen Schlüssel** von A . Ein partieller Schlüssel von A ist eine Menge von Attributen, die für *jeden einzelnen* Datensatz der Ausgangsrelation R bezüglich A die Schlüsseleigenschaft besitzt. Man bildet dann ein Relationenschema R_A , das aus dem Schlüssel von R und den Attributen des Relationenschemas von A besteht. Ein Schlüssel von R_A besteht aus dem Schlüssel von R und dem identifizierten partiellen Schlüssel von A .
- Enthalten die oben gebildeten Relationenschemata R_A selbst Attribute mit nichtatomaren Werten, wendet man das Verfahren nochmals auf die R_A an.

Beispiel 4.2.: Aus der Ausgangsrelation FREIZEIT entstehen die in Abb. 39 dargestellten Relationen.

□

NR	NAME	NR	HNAME	PRIOR
17	Beyer	17	Radfahren	3
22	Schneider	17	Musikhören	2
9	Mitterer	17	Schwimmen	1
63	Schmitt	22	Theater	1
		22	Reisen	1
		9	Wandern	1
		9	Schwimmen	2
		63	Radfahren	1

Abb. 39: Relation nach Überführung in erste Normalform

4.2.3. Zweite Normalform

Ein Relationenschema R ist in **zweiter Normalform**, wenn es bereits in erster Normalform ist und kein nichtprimales Attribut von einer echten Teilmenge eines Schlüssels von R funktional abhängig ist (oder: jedes nichtprimale Attribut voll funktional abhängig von jedem Schlüssel von R ist).

Hieraus folgt: Sind alle Schlüssel von R einelementig, so ist R bereits in zweiter Normalform.

Beispiel 4.3.: Um die Überführung einer Relation in die zweite Normalform zu illustrieren, betrachten wir als Beispiel folgendes Relationenschema

KINO (PLCODE, SAAL, PLATZ, REIHE, PREIS, GROESSE) mit dem ein Besitzer eines großen Kino-Centers mit mehreren Spielsälen die Verwaltung der Sitzplätze übernehmen will. Die Attribute bedeuten:

- PLCODE Eindeutige Platznummer im ganzen Kino-Center
- SAAL Nummer des Spielsaals
- PLATZ Nummer des Platzes im Spielsaal
- REIHE Reihe des Platzes im Spielsaal
- PREIS Eintrittspreis für den betreffenden Platz
- GROESSE Größe des Spielsaals

Sowohl das Attribut PLCODE allein als auch die Attribute SAAL und PLATZ identifizieren jedes Element der Relation eindeutig und sind somit Schlüssel für KINO. Wir haben damit zunächst folgende funktionale Abhängigkeiten (wie leicht zu erkennen ist, bilden diese ein minimales Abhängigkeitssystem):

a1: $\{\text{PLCODE}\} \mapsto \{\text{PLCODE}, \text{SAAL}, \text{PLATZ}, \text{REIHE}, \text{PREIS}, \text{GROESSE}\}$

a2: $\{\text{SAAL}, \text{PLATZ}\} \mapsto \{\text{PLCODE}, \text{SAAL}, \text{PLATZ}, \text{REIHE}, \text{PREIS}, \text{GROESSE}\}$

Natürlich ist die Saalgröße durch die Saalnummer bestimmt:

a3: $\{\text{SAAL}\} \mapsto \{\text{GROESSE}\}$

Zusätzlich nehmen wir an, dass der Preis eines Platzes nur durch seine Reihe bestimmt ist:

a4: $\{\text{REIHE}\} \mapsto \{\text{PREIS}\}$

Wir nehmen an, dass außer diesen funktionalen Abhängigkeiten keine weiteren Abhängigkeiten bestehen, die sich nicht aus den obigen Abhängigkeiten herleiten lassen. Die Abhängigkeiten sind in Abb. 40 graphisch dargestellt.

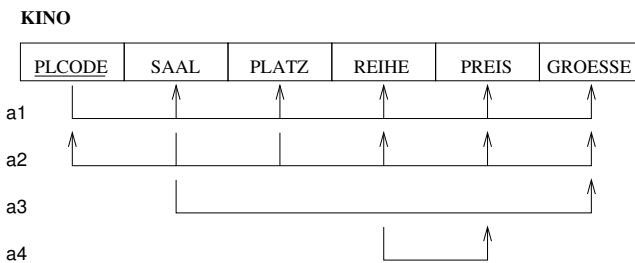


Abb. 40: Abhängigkeiten im Relationenschema KINO

Das Attribut GROESSE ist von $\{\text{SAAL}\}$, einer echten Teilmenge des Schlüssels $\{\text{SAAL}, \text{PLATZ}\}$, funktional abhängig und verletzt somit die Bedingung für die zweite Normalform.

Wir normalisieren, indem wir das “störende” Attribut GROESSE aus KINO entfernen (das verbleibende Relationenschema nennen wir KINO1) und zusammen mit dem Attribut SAAL, von dem GROESSE abhängig ist, in ein eigenes Relationenschema KINO2 aufnehmen. (In diesem ist logischerweise SAAL ein Schlüssel.) Es ergibt sich also das in Abb. 41 dargestellte Bild.

Während im ursprünglichen Relationenschema KINO noch eine Änderungsanomalie bei der Änderung der Zahl von Sitzplätzen in einem Saal vorhanden war, ist diese Anomalie durch die Normalisierung verschwunden.

□

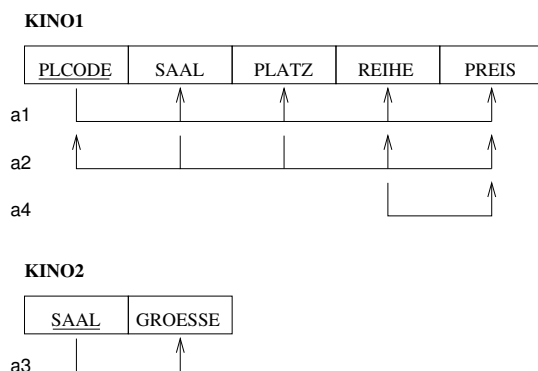


Abb. 41: KINO nach Transformation in zweite Normalform

4.2.4. Dritte Normalform

Ein Relationenschema R ist in **dritter Normalform**, wenn es in erster Normalform ist und für jede funktionale Abhängigkeit $X \rightarrow Y$ mit einelementigem Y in R gilt: X ist Superschlüssel von R oder Y ist prim.

Man beachte, dass ein Relationenschema in dritter Normalform automatisch auch in zweiter Normalform ist. Jedoch ist es nicht notwendig, für den Test, ob ein Relationenschema in dritter Normalform ist, die zweite Normalform zu berechnen.

Beispiel 4.4.: Wir betrachten wieder unsere Relationenschemata KINO1 und KINO2. Offensichtlich ist KINO2 in dritter Normalform: Als einzige Abhängigkeit haben wir $\{SAAL\} \rightarrow \{GROESSE\}$, und $\{SAAL\}$ ist Schlüssel, daher Superschlüssel von KINO2.

KINO1 ist nicht in dritter Normalform: Wir haben die Abhängigkeit $\{REIHE\} \rightarrow \{PREIS\}$, aber weder ist $\{REIHE\}$ Superschlüssel noch $\{PREIS\}$ primes Attribut. Wir spalten daher genau wie bei der zweiten Normalform KINO1 auf in zwei Relationen KINO1A und KINO1B; das Resultat ist in Abb. 42 zu sehen.

Änderungsanomalien bei Änderung des Eintrittspreises sind durch den Übergang in die dritte Normalform beseitigt.

□

Beachte: Relationen mit nur einem Nichtschlüsselattribut sind automatisch in dritter Normalform.

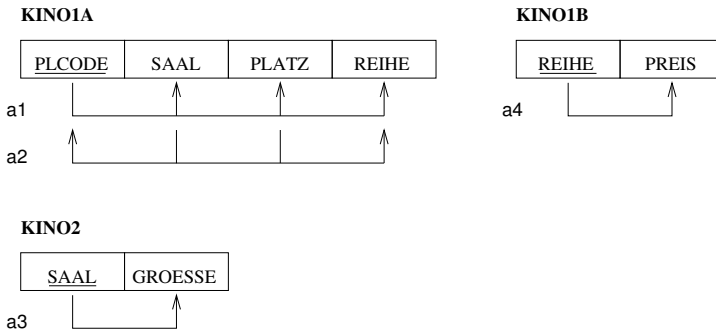


Abb. 42: KINO1 und KINO2 nach Transformation in dritte Normalform

4.2.5. Weitere Normalformen

Es gibt noch eine Reihe von weiteren Normalformen für Relationenschemata, etwa die Boyce-Codd-Normalform (BCNF), die vierte Normalform und die Domain-Key-Normalform (DKNF). Diese sind jedoch für die Praxis von untergeordneter Relevanz und werden deshalb nicht weiter betrachtet.

4.2.6. Vor- und Nachteile der Normalisierung

Zweifellos entstehen durch die Normalisierung Nachteile; unter anderem:

- Unübersichtlichere und schwieriger zu handhabende Datenbanken, da durch die Normalisierung mehr Tabellen entstehen.
- Längere Antwortzeiten durch die Notwendigkeit, Views einzuführen sowie Joins bei Datenbankabfragen zu verwenden.

Demgegenüber stehen jedoch viel gewichtigere Vorteile:

- Verminderung von Redundanz durch Auslagerung von redundanter Information in separate Tabellen
- Weniger Anomalien
- Mehr Konsistenz
- Speicherplatzersparnis.

Beispiel 4.5.: Das Argument der Speicherplatzersparnis wollen wir kurz an unserem KINO-Beispiel beleuchten.

Die Werte von PLCODE, SAAL, PLATZ, REIHE und GROESSE seien ganzzahlige Werte, die jeweils 4 Bytes Speicherplatz belegen. Die Werte von PREIS seien Festkommazahlen, die jeweils 8 Bytes belegen.

Damit haben wir

pro Eintrag in KINO	28 Bytes
pro Eintrag in KINO1A	16 Bytes
pro Eintrag in KINO1B	12 Bytes
pro Eintrag in KINO2	8 Bytes

Angenommen, das Kino hat 4 Säle mit jeweils 100 Plätzen in jeweils 10 Reihen.

In den unnormalisierten Relationen:

400 Einträge in KINO belegen 11200 Bytes.

In den normalisierten Relationen:

400 Einträge (Plätze) in KINO1A belegen	6400 Bytes
10 Einträge (Reihen) in KINO1B belegen	120 Bytes
4 Einträge (Säle) in KINO2 belegen	32 Bytes
Gesamt:	<u>6552</u> Bytes

Dies entspricht einer Speicherplatzersparnis von über 40% in diesem Beispiel. Generell gilt: Die Ersparnis wird umso höher, je mehr Daten gespeichert werden.

□

4.3. Dekomposition

Bei der Herstellung der zweiten oder dritten Normalform hatten wir “störende” funktionale Abhängigkeiten durch Aufspaltung einer Relation in zwei Relationen entfernt. Dieses Verfahren kann auf beliebige funktionale Abhängigkeiten angewendet werden und funktioniert formal wie folgt:

Sei $R(A)$ Relationenschema mit Attributmenge A . Seien $X, Y \subseteq A$ und gelte $X \rightarrow Y$. Dann heißen die Relationenschemata

$$R_1(A \setminus Y) \quad \text{und} \quad R_2(X \cup Y)$$

Dekomposition von R bezüglich $X \rightarrow Y$.

Offensichtlich gilt

$$R(A) \leftarrow R_1 \overset{N}{\bowtie}_{R_1.X=R_2.X} R_2$$

wobei $\overset{N}{\bowtie}$ einen Natural-Join bezeichnet. Die Dekompositionsoperation führt also zu Relationenschemata, die im erwarteten Sinne äquivalent zum Ausgangs-Relationenschema sind.

Beispiel 4.6.: Dass die Dekompositionsoption nicht nur für die Herstellung von Normalformen Sinn macht, sehen wir recht einfach durch nochmalige Betrachtung des Beispiels aus Abschnitt 4.1. Hier hatten wir das Relationenschema BESTELLUNGEN, das einschließlich der funktionalen Abhängigkeiten nochmals in Abb. 43 dargestellt ist.

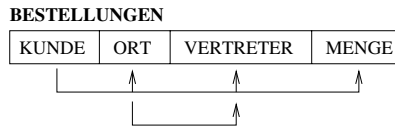


Abb. 43: Relation mit Anomalien und Abhängigkeiten

Diese Relation hatten wir zerlegt in die beiden in Abb. 44 dargestellten Relationen.



Abb. 44: Relationen mit Anomalien in dritter Normalform

Wie man leicht erkennt, ist sowohl BESTELL1 als auch ZUST in dritter Normalform. Wir haben nun zwar die Änderungsanomalie für die Zuordnung Vertreter – Ort beseitigt, aber die in Abschnitt 4.1 beschriebene Löschanomalie bleibt bestehen. Um diese zu beseitigen, ist eine Dekomposition der Relation BESTELL1 bezüglich der funktionalen Abhängigkeit $\{KUNDE\} \rightarrow \{ORT\}$ erforderlich. Das Resultat sieht man in Abb. 45.

□

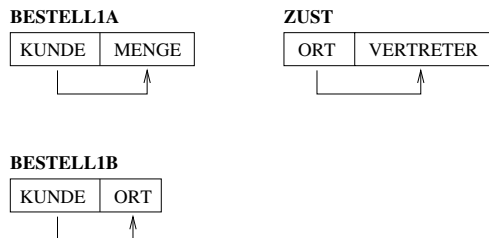


Abb. 45: Relationen ohne Änderungs- und Löschanomalie

4.4. Datenbankdefinition mit SQL

Nach dem theoretischen Datenbank-Entwurf einschließlich einer eventuellen Normalisierung und Dekomposition erhält man eine Menge von Relationenschemata, die als Tabellen in der Datenbank anzulegen sind.

4.4.1. Anlegen von Tabellen

Hierzu dient das Kommando CREATE TABLE. Beim Anlegen einer Tabelle müssen folgende Informationen bekannt sein:

- Name der Tabelle
- Namen und Datentypen der Tabellenspalten
- Eventuelle Standardwerte für Tabellenspalten (können auch nachträglich angegeben werden, siehe Abschnitt 4.4.2)
- Eventuelle Integritätsbedingungen (Constraints) für die Tabellenspalten (können auch nachträglich angegeben werden, siehe Abschnitt 4.4.2). Hierauf wird in Abschnitt 4.4.3 näher eingegangen.

Abb. 46 zeigt das Syntaxdiagramm des CREATE TABLE-Statements.

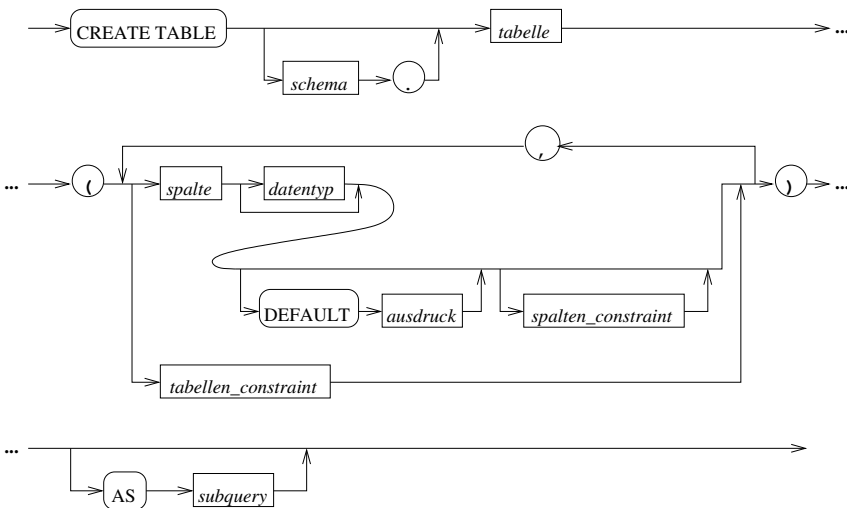


Abb. 46: Syntax des CREATE TABLE-Statements

Hierbei bedeutet:

spalte

Name der definierten Tabellenspalte.

datatype

Typ der Daten in der betreffenden Spalte. Dies muss ein gültiger SQL-Datentyp sein und ist

	immer anzugeben, sofern nicht mit einer AS-Klausel gleichzeitig Daten in die Tabelle eingefügt werden.
DEFAULT <i>ausdruck</i>	Bezeichnet den Standardwert, der beim Einfügen neuer Datenwerte in die betreffende Spalte eingetragen wird, wenn beim Einfügen kein Wert dafür angegeben ist.
AS <i>subquery</i>	Die Werte der durch die Unterabfrage gelieferten Ergebnistabelle werden sofort in die neu definierte Tabelle eingefügt. Bei dieser Form des CREATE TABLE-Statements dürfen bei der Spaltendefinition keine Datentypen angegeben werden, da sich diese aus der Unterabfrage ergeben.
<i>tabellen_constraint</i>	siehe Abschnitt 4.4.3.
<i>spalten_constraint</i>	siehe Abschnitt 4.4.3.

Beispiel 4.7.: Die Tabelle PERSONAL der Musterdatenbank FIRMA könnte wie folgt erzeugt worden sein:

```
CREATE TABLE personal (  
    pid INTEGER,  
    nachname CHARACTER VARYING(20),  
    vorname CHARACTER VARYING(15),  
    strasse CHARACTER VARYING(30),  
    ort CHARACTER VARYING(20),  
    einstellung DATE,  
    gehalt NUMERIC(7,2),  
    vorgesch_id INTEGER,  
    aid INTEGER  
);
```



Das folgende Beispiel zeigt eine Tabellendefinition mit gleichzeitigem Einfügen von Daten.

Beispiel 4.8.: Erzeugen einer Tabelle WOHNORTE, die die Wohnorte der Mitarbeiter enthält:

```
CREATE TABLE wohnorte (ort)  
AS SELECT DISTINCT ort FROM personal;
```

Die Abfrage

```
SELECT * FROM wohnorte;
```

liefert dann

Augsburg
Bad Tölz
Freising
München

□

Mit dem SQL-Befehl DROP TABLE kann man eine Tabelle (einschließlich aller enthaltenen Datensätze) löschen.

Beispiel 4.9.: Löschen der gerade erzeugten Tabelle WOHNORTE:
DROP TABLE wohnorte;

□

4.4.2. Ändern von Tabellendefinitionen

Die Definition einer bereits vorhandenen Tabelle kann mit ALTER TABLE geändert werden (Abb. 47).

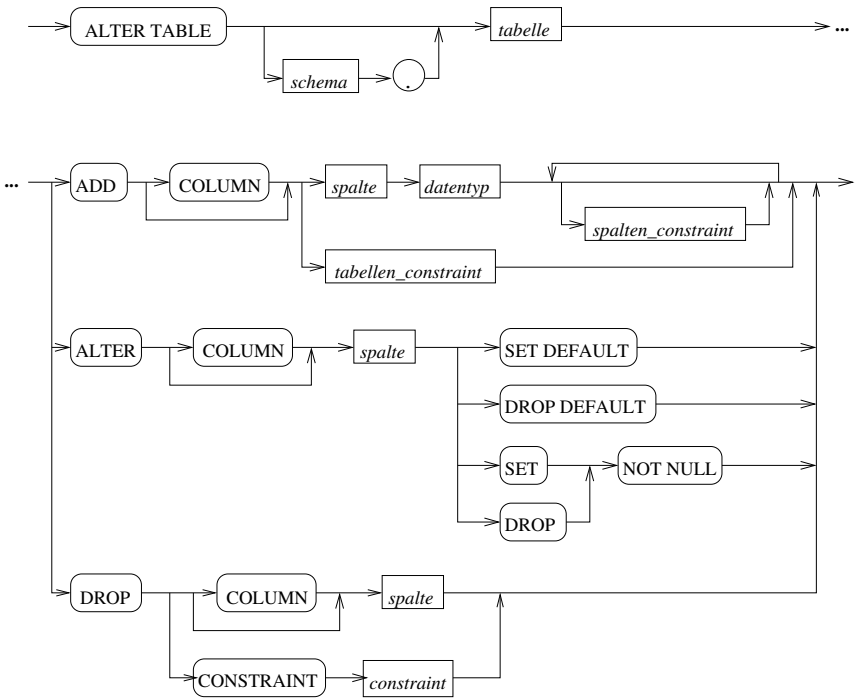


Abb. 47: Syntax des ALTER TABLE-Statements

Hierbei bedeuten:

ADD	Hinzufügen einer neuen Spalte oder eines Tabellen-constraints. Für alle Datensätze, die sich bereits in der Tabelle befinden, erhält die neue Spalte einen NULL-Wert.
ALTER	Setzen bzw. Löschen von DEFAULT-Werten einer Spalte sowie Setzen und Löschen von NOT NULL-Constraints (siehe folgenden Abschnitt). Die Formen SET NOT NULL und DROP NOT NULL sind nicht im SQL-Standard enthalten.
DROP COLUMN	Die angegebene Spalte wird aus der Tabelle entfernt. Dabei gehen selbstverständlich auch alle Inhalte der betreffenden Spalte verloren.
DROP CONSTRAINT	Entfernt den Constraint mit dem Namen <i>constraint</i> aus der Tabellendefinition.

Beispiel 4.10.: Die Tabelle PERSONAL soll um eine Spalte ‘Zulage’ ergänzt werden, in der besondere Zahlungen, die ein Mitarbeiter zusätzlich zum normalen Gehalt erhält, erfasst werden. Es wird angenommen, dass die Zulagen kleiner als 1000 EUR sind und auf Cent genau angegeben werden:

```
ALTER TABLE personal ADD zulage NUMERIC(5,2);
```



Beispiel 4.11.: Um die soeben angelegte Spalte ‘Zulage’ wieder zu löschen, gibt man die Anweisung

```
ALTER TABLE personal DROP COLUMN zulage;
```



4.4.3. Constraints

Um die Verwaltung von Tabellen zu erleichtern sowie die Integrität einer Datenbank zu sichern, kann man Integritätsbedingungen (integrity constraints) für Tabellen festlegen.

Es gibt folgende Integritätsbedingungen:

- NOT NULL-Constraint auf Spalten. Dieser stellt sicher, dass die betreffende Spalte niemals einen NULL-Wert enthält.
- UNIQUE-Constraint auf einer Menge von Spalten. Damit wird sichergestellt, dass in der betreffenden Spaltenmenge kein Eintrag – ausgenommen eventuelle NULL-Werte – doppelt auftritt.
- PRIMARY KEY-Constraint auf einer Menge von Spalten. Dieser Constraint legt den Primärschlüssel der Tabelle fest und stellt sicher,

dass es keine zwei Datensätze in der Tabelle mit gleichen PRIMARY KEY-Werten gibt. Ferner wird damit erreicht, dass keine der Spalten des PRIMARY KEY einen NULL-Wert enthält.

- FOREIGN KEY-Constraint auf einer Menge von Spalten (Abhängigkeits-Constraint). Dieser Typ von Constraint assoziiert eine Spaltenmenge einer Tabelle (Kind-Tabelle) mit einer als PRIMARY KEY oder UNIQUE definierten Spaltenmenge einer anderen Tabelle (Eltern-Tabelle). Dabei kann die Kind-Tabelle nur Einträge enthalten, für die bezüglich der entsprechenden Spaltenmenge schon ein Eintrag in der Eltern-Tabelle besteht. Ausnahme: NULL-Werte sind von dieser Regelung nicht betroffen. Die referenzierten Spalten der Eltern-Tabelle müssen entweder mit einem PRIMARY KEY- oder einem UNIQUE-Constraint versehen sein.
- CHECK-Constraints geben eine Bedingung an, die ein Datensatz erfüllen muss, damit er in der Tabelle enthalten sein kann. Auch wenn die Bedingung zu NULL evaluiert, ist dieser Constraint erfüllt. Ein CHECK-Constraint kann sich in der aktuellen PostgreSQL-Version nur auf den aktuellen Datensatz beziehen und ist daher nur recht eingeschränkt verwendbar.

Bei einer Tabelle, die bereits Daten enthält, kann ein Constraint nur dann definiert werden, wenn die Datensätze in der Tabelle den Constraint erfüllen. In eine Tabelle können nur Datensätze eingefügt werden, die die auf der Tabelle definierten Constraints erfüllen. Bei der Änderung von Datensätzen müssen die resultierenden Datensätze die vorhandenen Constraints erfüllen. Auch das Löschen von Datensätzen aus Tabellen ist nur dann möglich, wenn dadurch keine vorhandenen Constraints verletzt werden.

Die Syntax der Constraintdefinition im CREATE TABLE- bzw. ALTER TABLE-Statement geht aus Abb. 48 (Tabellenconstraints) bzw. Abb. 49 (Spaltenconstraints) hervor.

Hierbei bedeutet:

- | | |
|--------------------|--|
| CONSTRAINT | Optionale Vergabe eines selbstgewählten Namens an den Constraint. Man sollte einem Constraint selbst einen Namen geben, wenn man sich nachher auf ihn beziehen will (z. B. ihn wieder löschen will). |
| FOREIGN KEY | |
| REFERENCES | Mit dieser Klausel wird angegeben, von welchen Spalten der Eltern-Tabelle die Spalten abhängen. |

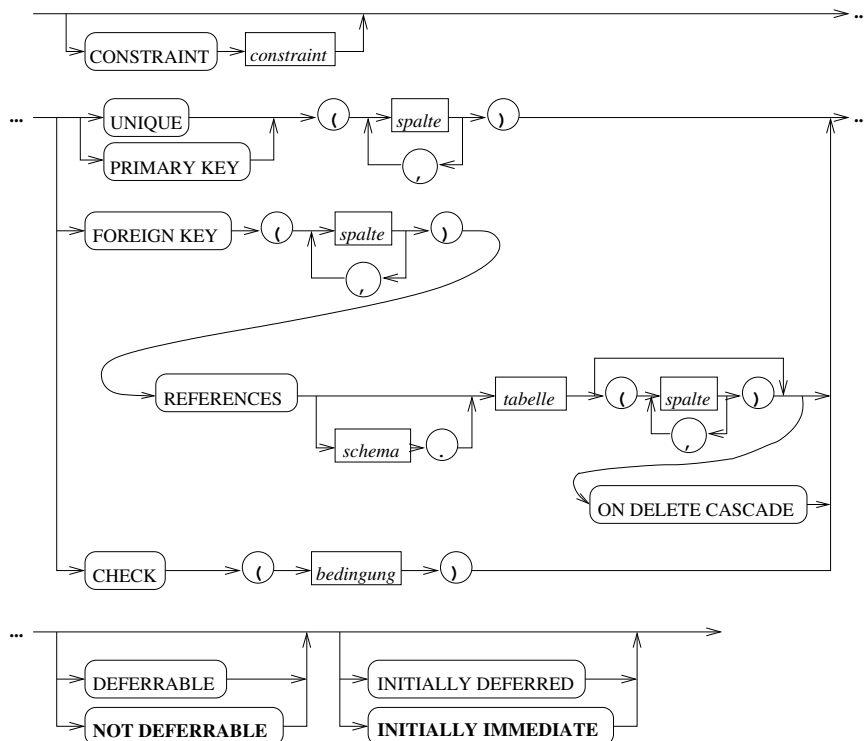


Abb. 48: Syntax einer Tabellenconstraintdefinition

ON DELETE

CASCADE

Spezifiziert, dass ein Datensatz automatisch gelöscht wird, wenn der referenzierte Datensatz in der Eltern-Tabelle gelöscht wird. Ist dies nicht angegeben, führt der Versuch, einen Datensatz aus der Eltern-Tabelle zu löschen, zu einer Fehlermeldung, falls noch eine Referenz auf den Datensatz existiert.

DEFERRABLE

NOT DEFERRABLE Hier wird festgelegt, ob die normalerweise sofort nach Ausführung eines Statements stattfindende Prüfung auf Gültigkeit eines Constraints auf einen späteren Zeitpunkt – nämlich dem Ende einer Transaktion (siehe Abschnitt 5.1) – verlegt werden kann. Standardmäßig ist keine Verlegung möglich (**NOT DEFERRABLE**); wenn diese möglich sein soll, muss **DEFERRABLE** angegeben werden.

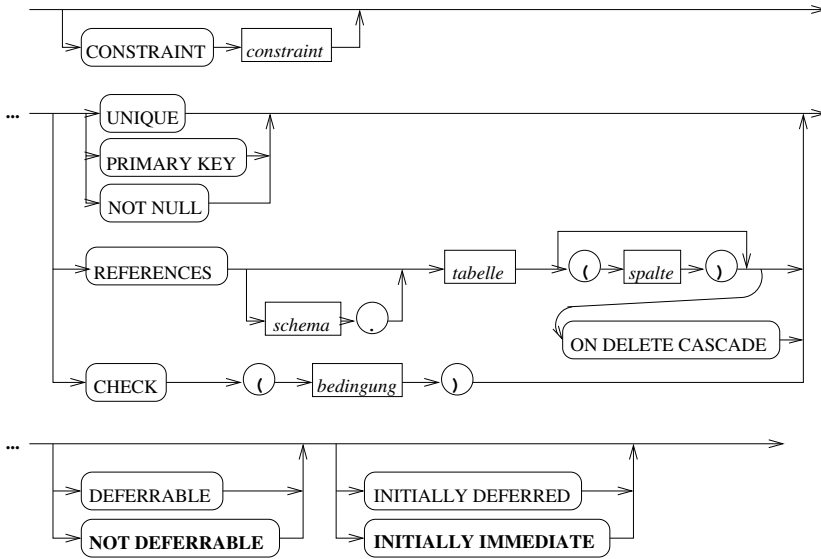


Abb. 49: Syntax einer Spaltenconstraintdefinition

INITIALLY DEFERRED

INITIALLY IMMEDIATE

Hier legt man den grundsätzlichen Zeitpunkt der Prüfung der Constraintgültigkeit fest. **INITIALLY IMMEDIATE** bedeutet, dass die Gültigkeit nach jedem Statement geprüft wird (dies kann jedoch, soweit zusätzlich **DEFERRABLE** angegeben ist, geändert werden). Bei **INITIALLY DEFERRED** wird die Gültigkeit erst am Transaktionsende geprüft (dies kann jedoch in einer Transaktion geändert werden). Ist diese Klausel nicht angegeben, wird **INITIALLY IMMEDIATE** angenommen. In PostgreSQL besteht im Gegensatz zum SQL-Standard die Einschränkung, dass der **DEFERRED**-Modus nur für **FOREIGN KEY**-Constraints gewählt werden kann.

Beispiel 4.12.: Am Beispiel unserer Firmendatenbank wollen wir die Verwendung von Constraints im praktischen Beispiel demonstrieren:

– Primärschlüssel:

PERSONAL: {Pid}

ABTEILUNG: {Aid}

PROJEKT: {Projid}

ZUORDNUNG: {Pid, Projid}

In SQL lautet die Definition dieser Primärschlüssel:

```
ALTER TABLE personal ADD PRIMARY KEY (pid);
ALTER TABLE abteilung ADD PRIMARY KEY (aid);
ALTER TABLE projekt ADD PRIMARY KEY (projid);
ALTER TABLE zuordnung ADD PRIMARY KEY (pid,projid);
```

- Für folgende Spalten ist es sinnvoll, NULL-Werte auszuschließen:
PERSONAL.Nachname, ABTEILUNG.Bezeichnung,
PROJEKT.Name.

Definition dieser Constraints in SQL:

```
ALTER TABLE personal ALTER nachname SET NOT NULL;
ALTER TABLE abteilung ALTER bezeichnung SET NOT NULL;
ALTER TABLE projekt ALTER name SET NOT NULL;
```

Man beachte hier, dass NOT NULL-Constraints nur in Form eines Spalten- und nicht eines Tabellenconstraints definiert werden können.

- Folgende Spalten/Spaltenmengen sollen keine doppelten Werte enthalten:

{ABTEILUNG.Bezeichnung, ABTEILUNG.Ort}

sowie

PROJEKT.Name.

In SQL:

```
ALTER TABLE abteilung ADD UNIQUE (bezeichnung,ort);
ALTER TABLE projekt ADD UNIQUE (name);
```

- Foreign-Key-Constraints:

Hier bedeutet die Schreibweise $T_1.X \leftarrow T_2.Y$, dass Werte der Attributmenge X aus T_1 als Werte der Attributmenge Y in T_2 vorkommen müssen. Hier also:

{PERSONAL.Vorges_Id} \leftarrow {PERSONAL.Pid}

denn der Vorgesetzte muss existieren,

{PERSONAL.Aid} \leftarrow {ABTEILUNG.Aid}

(die Abteilung, der ein Mitarbeiter zugeordnet ist, muss existieren),

{ZUORDNUNG.Pid} \leftarrow {PERSONAL.Pid}

und

{ZUORDNUNG.Projid} \leftarrow {PROJEKT.Projid}

da der Mitarbeiter und das Projekt in einer Projektzuordnung existieren müssen.

Wir definieren diese Constraints in SQL wie folgt:

```
ALTER TABLE personal ADD FOREIGN KEY (vorges_id)
REFERENCES personal (pid);
ALTER TABLE personal ADD FOREIGN KEY (aid)
```

```

REFERENCES abteilung (aid);
ALTER TABLE zuordnung ADD FOREIGN KEY (pid)
REFERENCES personal (pid);
ALTER TABLE zuordnung ADD FOREIGN KEY (projid)
REFERENCES projekt (projid);
– Werte in der Spalte
  PERSONAL.Gehalt
sollten größer als 0 und kleiner als 20000 sein:
ALTER TABLE personal
  ADD CHECK (gehalt > 0 and gehalt < 20000);

```

□

Beispiel 4.13.: Folgende Operationen verletzen die definierten Constraints und werden daher vom Datenbanksystem mit einer Fehlermeldung quittiert:

- Entfernung des Mitarbeiters mit der Nr. 128 aus der Tabelle PERSONAL:

```
DELETE FROM personal WHERE pid=128;
```

verletzt den FOREIGN KEY-Constraint von der Tabelle ZUORDNUNG.

- Entfernung des Mitarbeiters mit der Nr. 57 aus der Tabelle PERSONAL:

```
DELETE FROM personal WHERE pid=57;
```

verletzt den FOREIGN KEY-Constraint von der Tabelle PERSONAL selbst, da dieser Mitarbeiter Vorgesetzter anderer Mitarbeiter ist.

- Löschen des Projektes mit der Nr. 3 aus der Tabelle PROJEKT:

```
DELETE FROM projekt WHERE projid=3;
```

verletzt den FOREIGN KEY-Constraint von der Tabelle ZUORDNUNG.

- Einfügen eines neuen Mitarbeiters mit Namen Alfons Anders und Personalnummer 128:

```
INSERT INTO personal (pid,nachname,vorname)
VALUES (128, 'Anders', 'Alfons');
```

verletzt den PRIMARY KEY-Constraint der Tabelle PERSONAL.

Um wie im vorletzten Fall ein Projekt zu löschen und den Datenbestand konsistent zu halten, muss man zuerst in ZUORDNUNG die Zuordnungen von Mitarbeitern zu diesem Projekt löschen und kann dann den Eintrag in PROJEKT löschen:

```
DELETE FROM zuordnung WHERE projid = 3;
DELETE FROM projekt WHERE projid=3;
```

Mit der ON DELETE CASCADE-Klausel bei der Definition des entsprechenden Constraints ließe sich dies auch automatisieren.



4.4.4. Indizes

Indizes sind Strukturen, die auf Tabellen eingerichtet werden können. Mit Hilfe eines Index laufen Zugriffe auf Daten in der Tabelle im allgemeinen schneller ab als ohne Index. Die Syntax für Datenbankabfragen ändert sich bei Verwendung eines Index nicht; ein Index wirkt sich lediglich auf die Ausführungsgeschwindigkeit eines Tabellenzugriffs aus.

Ein Index auf eine Datenbanktabelle kann mit einem Index am Schluss eines Buches verglichen werden: Anstatt das ganze Buch von vorn bis hinten nach der gewünschten Information zu durchsuchen, konsultiert man den Index, der direkt die Stelle angibt, an der die Information steht.

Sobald man einen Index erzeugt, wird er automatisch vom Datenbanksystem benutzt und verwaltet, d. h. alle Änderungen an den Daten der Tabelle werden auch im Index berücksichtigt.

Jeder Index einer Tabelle wird auf einer Menge von Spalten definiert. Es werden dann in Abfragen die Tabellenzugriffe beschleunigt, die in der WHERE-Klausel Bezug auf eine indizierte Spalte nehmen. Abfragen mit ORDER BY-Klausel werden ebenfalls beschleunigt, wenn sie nach indizierten Spalten sortiert werden.

Es ist möglich und durchaus sinnvoll, mehrere Indizes auf einer Tabelle zu definieren. Da beim Ändern der Daten der Tabelle jedoch auch die Indizes geändert werden müssen, werden bei Vorhandensein vieler Indizes die Datenmodifikationsbefehle langsamer.

Indizes werden intern in einer effizienten Datenstruktur gespeichert, beispielsweise in Form sogenannter balancierter B-Bäume. Dabei handelt es sich um Strukturen, die auf in sortierter Reihenfolge vorliegende Informationen einen besonders effizienten Zugriff (in nahezu konstanter Zeit) ermöglichen. Ein Knoten eines B-Baums, der kein Blatt ist, enthält k Präfixe eines Suchbegriffs, sowie $k + 1$ Nachfolgerknoten. Ist der tatsächliche Suchbegriff größer oder gleich dem i -ten, aber kleiner als der $(i + 1)$ -te Präfix, so befindet sich der Suchbegriff in dem Ast des $(i + 1)$ -ten Nachfolgers. Die Blätter des Baumes bilden die Suchbegriffe selbst. Werden Änderungen am Index vorgenommen, wird der

B-Baum automatisch so umstrukturiert, dass wieder ein effizienter Zugriff möglich ist.

Ein Index auf der Spalte ‘Nachname’ der Tabelle PERSONAL könnte wie in Abb. 50 gezeigt strukturiert sein.

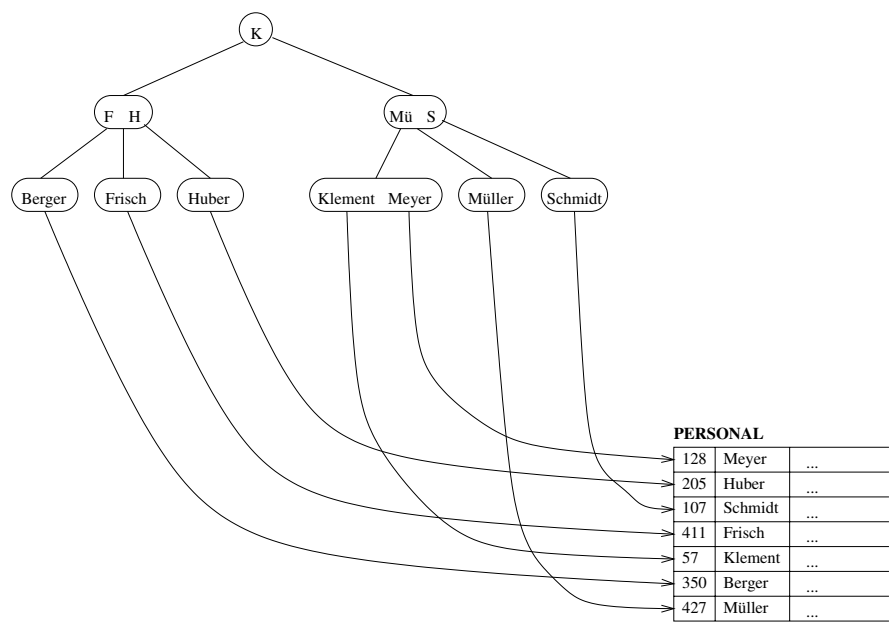


Abb. 50: Struktur eines Index

Aus dem Beispiel wird deutlich, dass das Aufsuchen des Datensatzes ‘Müller’ ohne Index 7 Zugriffe benötigt (da es der letzte Datensatz ist, muss die gesamte Tabelle durchsucht werden), bei Verwendung des Index werden hingegen nur 3 Zugriffe benötigt (Besuch der Knoten ‘K’, ‘Mü S’, ‘Müller’).

Ein Index wird umso effizienter, je mehr Einträge die Tabelle enthält.

Ein Index wird mit dem Kommando CREATE INDEX erzeugt (Abb. 51).

Hierbei bedeutet:

index Name des zu erzeugenden Index.

ON Tabelle, auf der der Index erzeugt wird.

spalte Liste der Spalten, auf denen der Index definiert werden soll.

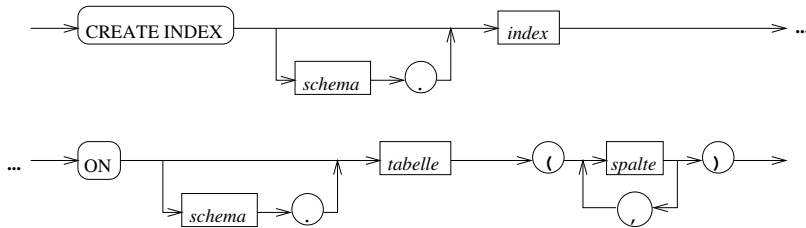


Abb. 51: Syntax des CREATE INDEX-Statements

Beispiel 4.14.: Erzeugen eines Index 'i_nachname' auf der Spalte 'Nachname' der Tabelle PERSONAL:

```
CREATE INDEX i_nachname ON personal (nachname);
```

□

Ein Index kann mit DROP INDEX wieder gelöscht werden. Löschen eines Index hat keinen Einfluss auf die in der Tabelle gespeicherten Daten.

Beispiel 4.15.: Löschen des Index 'i_nachname':

```
DROP INDEX i_nachname;
```

□

Beachte: Bei der Definition von PRIMARY KEY- und UNIQUE-Constraints wird automatisch ein Index auf den betreffenden Spalten angelegt, um die Einhaltung der Constraints zu überwachen.

Indizes sind im SQL-Standard nicht vorgesehen, sind aber in den meisten Datenbanksystemen implementiert.

5. Datenbankbetrieb

In diesem Abschnitt soll auf praktische Probleme im laufenden Datenbankbetrieb und ihre Handhabung mit SQL eingegangen werden.

5.1. Das Transaktionskonzept in Datenbanksystemen

Änderungen, die mit DML-Statements am Datenbestand vorgenommen werden, sind zunächst nur vorläufig – sie können mit DCL-Statements entweder endgültig ausgeführt oder zurückgenommen werden. Dies ist die Grundlage des Transaktionskonzepts.

Eine **Transaktion** ist eine logisch zusammengehörende Folge von Datenbankoperationen (die über SQL-Statements realisiert werden). Eine Transaktion ist unteilbar (atomar): Entweder *alle* durch die Transaktion bewirkten Änderungen werden in die Datenbank übernommen (COMMIT-Operation) oder *alle* Änderungen werden rückgängig gemacht (ROLLBACK-Operation), als ob die Transaktion nie stattgefunden hätte.

Damit wird sichergestellt, dass es zu keinen Dateninkonsistenzen kommt, weil eine Transaktion z. B. wegen eines Hard- oder Softwarefehlers nicht zu Ende ausgeführt werden kann. Die durch eine abgebrochene Transaktion bewirkten Änderungen werden nicht in die Datenbank übernommen (automatischer ROLLBACK).

Beispiel 5.1.: Eine Bank verwaltet Giro- und Sparkontenstände in zwei Tabellen, außerdem gibt es eine weitere Tabelle, in der Kontenbewegungen protokolliert werden. Ein Kunde möchte EUR 1000 von seinem Girokonto Nr. 24637 auf sein Sparkonto Nr. 90955 überweisen. Die Folge von Operationen, die diese Kontenbewegung (einschließlich Protokollierung) vornimmt, ist eine Transaktion in obigem Sinne (Abb. 52).



Um die Konsistenz des Datenbestandes der Bank zu wahren, müssen entweder *alle* durch die Transaktion bewirkten Änderungen übernommen oder aber *alle* Änderungen verworfen werden. Die Übernahme findet statt, wenn das COMMIT-Statement ausgeführt wird. Wird wegen eines Systemabsturzes COMMIT nicht ausgeführt, werden die Änderungen nicht übernommen.

Eine Transaktion beginnt in PostgreSQL mit dem START TRANSACTION-Statement. Bei Statements, die nicht in einem durch START

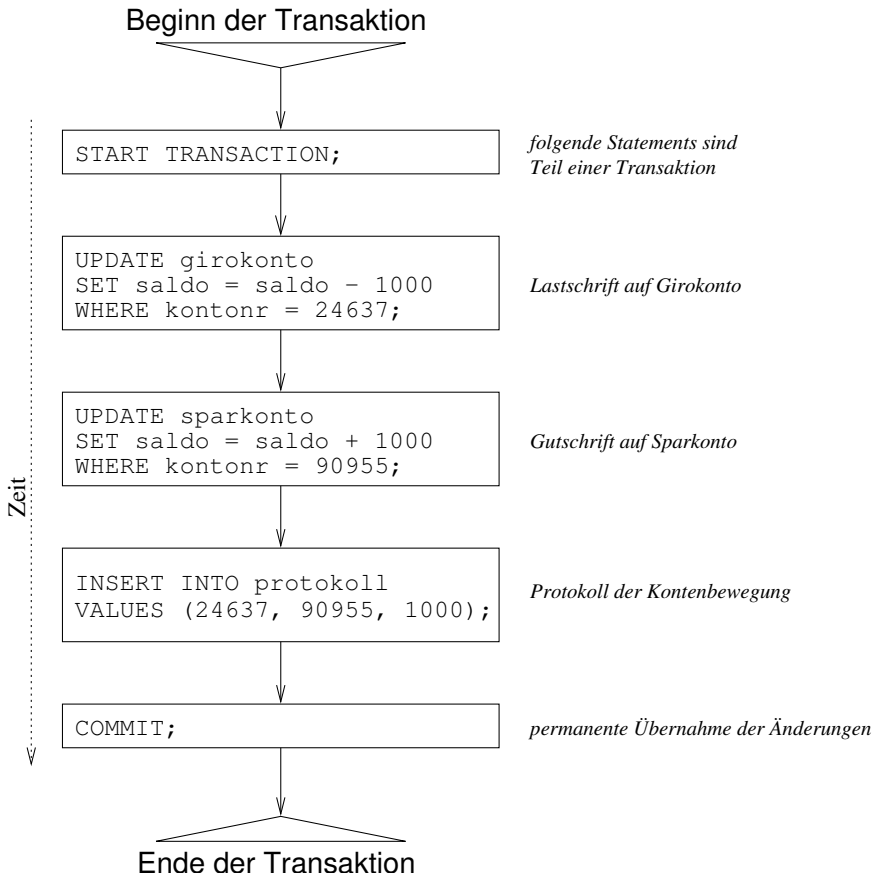


Abb. 52: Beispiel für eine Transaktion

TRANSACTION gegebenen Transaktionskontext stehen, erfolgt ein implizites COMMIT nach Ausführung des betreffenden Statements (sog. Autocommit).

Eine Transaktion wird u. a. beendet

- explizit durch das COMMIT-Statement, das die Übernahme der Änderungen in die Datenbank bewirkt.
- explizit durch das ROLLBACK-Statement, das alle in der Transaktion vorgenommenen Änderungen wieder verwirft.
- implizit durch Auftreten nicht behebbarer Fehler (z. B. Abbruch der Verbindung zum Datenbank-Server). Hier wird bei PostgreSQL automatisch ein ROLLBACK ausgeführt; im SQL-Standard ist dies eine Kann-Bestimmung.

5.2. Mehrbenutzerbetrieb

Im praktischen Einsatz von Datenbanksystemen bedeutet Mehrbenutzerbetrieb nicht nur, dass mehrere Benutzer jeweils gleichzeitig auf ihre eigenen Tabellen zugreifen, sondern vor allem, dass mehrere Benutzer parallel auf die *gleichen* Tabellen zugreifen können. Bei parallelen Zugriffen auf die gleichen Daten können folgende Probleme auftreten:

- “dirty reads”:
Eine Transaktion T1 liest Daten einer Transaktion T2, die noch nicht mit COMMIT permanent gemacht wurden. T1 liest also Daten, die noch gar nicht wirklich existieren.
- Nicht wiederholbare Leseoperationen (“nonrepeatable reads”):
In einer Transaktion liefern zwei identische Abfragen verschiedene Resultate, wenn zwischen der Ausführung der beiden Abfragen eine andere (inzwischen beendete) Transaktion Änderungen an den Daten vorgenommen hat.
- Inkonsistente Leseoperationen (“phantom reads”):
Nachdem eine Transaktion T1 eine Abfrage mit Bedingung C ausgeführt hat, ändert eine andere Transaktion T2 Daten, die die Bedingung C erfüllen, so dass bei nochmaliger Ausführung der gleichen Abfrage in T1 andere Daten geliefert werden.
- Verlorene Änderungen:
Eine Änderung in einer Transaktion T1 überschreibt eine Änderung in einer Transaktion T2, bevor T2 ihre Änderungen mit COMMIT permanent machen kann.

Eine einfache Lösung dieser Probleme ist mit **Sperren (Locks)** möglich: Tabellen bzw. Datensätze, auf denen gerade gearbeitet wird, werden für den Zugriff durch andere Transaktionen gesperrt.

Man unterscheidet grundsätzlich zwei Arten von Sperren:

- Lese-Sperre (share lock, ‘S’) zum Lesen von Daten: Verbietet das Ändern der gesperrten Daten durch andere Transaktionen. Der Lesezugriff durch andere Transaktionen wird nicht eingeschränkt.
- Exklusiv-Sperre (exclusive lock, ‘X’) zum Ändern von Daten: Keine andere Transaktion kann auf die gesperrten Daten zugreifen (weder lesend noch schreibend).

Auf einer Tabelle bzw. einem Datensatz können mehrere Lese-Sperren, jedoch höchstens eine Exklusiv-Sperre gesetzt sein. Es können nicht gleichzeitig Lese-Sperren und Exklusiv-Sperren auf der gleichen Tabelle bzw. dem gleichen Datensatz aktiv sein.

Man kann Sperren auf ganze Tabellen oder nur auf gewisse Datensätze anwenden.

Kann eine Transaktion eine Sperre nicht setzen, weil eine andere bereits gesetzte Sperre dies verhindert, wartet die Transaktion, bis das Setzen der Sperre wieder möglich ist. Hierdurch kann es zu sogenannten **Deadlock**-Situationen kommen, in denen zwei Transaktionen gegenseitig auf die Freigabe von Datensätzen warten.

In Abb. 53 ist eine typische Deadlock-Situation dargestellt.

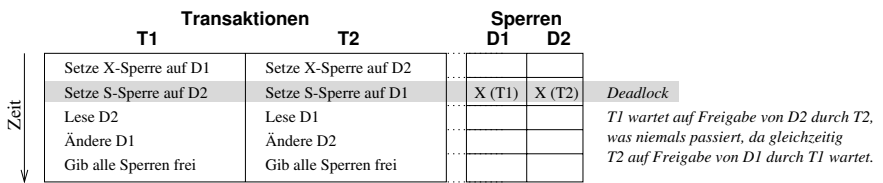


Abb. 53: Eine typische Deadlock-Situation

In manchen Datenbanksystemen (u. a. auch in PostgreSQL) werden Deadlocks automatisch erkannt und nach Prioritätsregeln aufgelöst. Ansonsten muss man durch geschickte Programmierung vermeiden, dass Deadlock-Situationen überhaupt auftreten können.

Das explizite Setzen von Sperren – das im SQL-Standard auch nicht vorgesehen ist – ist jedoch oft nicht die beste Lösung für die Realisierung konkurrierender Zugriffe auf Daten. Man fährt meist besser, indem man die Sicherstellung der Datenkonsistenz dem Datenbanksystem selbst überlässt. Im SQL-Standard kann man je nach der gewünschten Art der Datenkonsistenz einen von vier der in Abb. 54 aufgeführten Transaktionsmodi wählen.

Modus	dirty reads	nonrep. reads	phantom reads
READ UNCOMMITTED	möglich	möglich	möglich
READ COMMITTED	nicht möglich	möglich	möglich
REPEATABLE READ	nicht möglich	nicht möglich	möglich
SERIALIZABLE	nicht möglich	nicht möglich	nicht möglich

Abb. 54: Transaktionsmodi im SQL-Standard

Das Problem von verlorenen Änderungen einer Transaktion ist im SQL-Standard stets ausgeschlossen.

In PostgreSQL gibt es nur die Transaktionsmodi READ COMMITTED und SERIALIZABLE. Der Modus einer Transaktion wird beim Starten der Transaktion mit START TRANSACTION vorgenommen. Die Syntax dieses Statements ergibt sich aus Abb. 55.

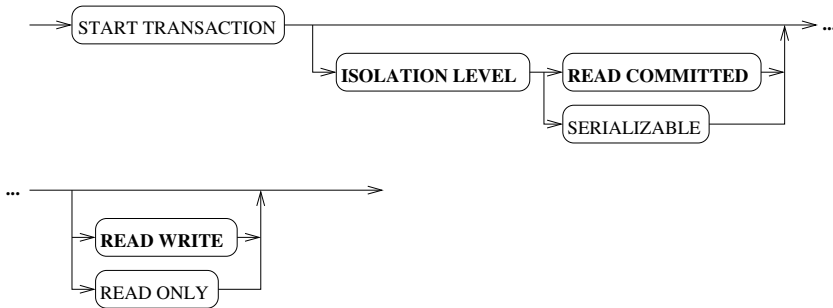


Abb. 55: Syntax des START TRANSACTION-Statements

Der Standardmodus einer Transaktion ist in PostgreSQL READ COMMITTED (im SQL-Standard ist er SERIALIZABLE). Zusätzlich kann noch spezifiziert werden, ob es sich um eine Schreib-/Lese-Transaktion (wird standardmäßig angenommen) oder um eine Nur-Lese-Transaktion handelt. Im letzteren Fall sind in der Transaktion keine Änderungsabfragen (INSERT, UPDATE, DELETE) sowie keine Änderungen am Datenbankschema durch DDL-Statements erlaubt.

Eine Schreib-/Lese-Transaktion kann bereits vorhandene Daten ändern (mit UPDATE und DELETE) oder auch mit der in Abb. 56 gezeigten FOR UPDATE-Klausel eines SELECT-Statements zur Änderung vormerken. In diesen Fällen werden die betroffenen Datensätze automatisch mit einer Lese-Sperre versehen, so dass Änderungen nur von genau einer Transaktion ausgeführt werden können.

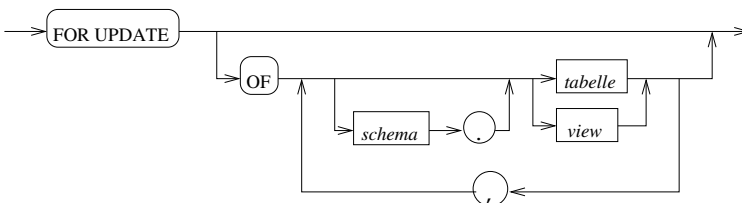


Abb. 56: Syntax der FOR UPDATE-Klausel

Bei der FOR UPDATE-Klausel gibt man die Namen von Tabellen an, deren selektierte Datensätze mit einer Lese-Sperre versehen werden sollen, also bei evtl. konkurrierenden Transaktionen als geändert angesehen werden sollen. Sind keine Tabellen angegeben, werden die betroffenen Datensätze aller Tabellen in der Abfrage gesperrt. Im SQL-Standard muss jeweils eine Tabellenspalte angegeben werden, diese ist jedoch irrelevant, da Sperren zeilen- und nicht feldweise gesetzt werden.

Alle Sperren werden bei Transaktionsende automatisch aufgehoben. Das Transaktionsende ist gleichzeitig die einzige Möglichkeit, Sperren aufzuheben.

“Dirty reads” können in PostgreSQL nicht vorkommen, da während der Ausführung einer Abfrage stets die bei *Beginn* der Abfrage vorliegenden Daten verwendet werden (READ COMMITTED-Modus). Man beachte aber, dass z. B. zwei verschiedene SELECT-Statements in der gleichen Transaktion durchaus verschiedene Resultate liefern können, nämlich dann, wenn eine andere Transaktion inzwischen Änderungen mit COMMIT abgeschlossen hat.

Eine READ COMMITTED-Transaktion T1, die Daten ändern will, die mittlerweile (seit Beginn von T1) von einer anderen Transaktion T2 geändert oder (mit SELECT FOR UPDATE) zur Änderung vorgemerkt wurden, wartet ggf. zunächst, bis T2 abgeschlossen wird. Wenn T2 mit ROLLBACK beendet wird, ergeben sich keine Probleme, da ja dann nur T1 die Daten letztlich ändert. Wird hingegen T2 mit COMMIT beendet, so nimmt T1 die Änderungen an den bereits von T2 geänderten Daten vor. Dies kann bei komplexeren Szenarien zu Konsistenzproblemen führen; wenn diese nicht ausgeschlossen werden können, sollte die betreffende Transaktion im SERIALIZABLE-Modus ausgeführt werden.

Im SERIALIZABLE-Modus sieht eine Abfrage immer den Zustand der Daten bei Beginn der Transaktion, d. h. dass zwei gleiche Abfragen in der gleichen Transaktion immer die gleichen Resultate liefern (dies gilt natürlich nicht, wenn die Transaktion inzwischen selbst die Daten geändert hat).

Will eine SERIALIZABLE-Transaktion T1 Daten ändern, die mittlerweile (seit Beginn von T1) von einer anderen Transaktion T2 geändert oder (mit SELECT FOR UPDATE) zur Änderung vorgemerkt wurden, wartet ggf. T1 zunächst, bis T2 abgeschlossen wird. Wenn T2 mit ROLLBACK beendet wird, ergeben sich keine Probleme, da ja dann

nur T1 die Daten letztlich ändert. Wird hingegen T2 mit COMMIT beendet, so wird T1 mit einem impliziten ROLLBACK (und Fehlermeldung an die Instanz, die die Transaktion angestoßen hat) abgebrochen, denn eine Transaktion im Modus `SERIALIZABLE` kann keine Daten ändern, die seit dem Beginn dieser Transaktion von einer anderen Transaktion modifiziert wurden – T1 würde ja sonst möglicherweise die von T2 gemachten Änderungen wieder überschreiben.

In dem Fall, dass eine `SERIALIZABLE`-Transaktion T1 aus dem oben geschilderten Grund abgebrochen wird, muss von der aufrufenden Instanz dafür gesorgt werden, dass T1 nochmals ausgeführt wird – T1 sieht dann die von T2 vorgenommenen Änderungen bereits am Beginn der nochmaligen Ausführung.

Beispiel 5.2.: Das Gehalt des Mitarbeiters Nr. 411 soll erhöht werden, wobei der Umfang der Erhöhung aufgrund der Kenntnis des bisherigen Gehalts festgesetzt werden soll. In Abb. 57 ist gezeigt, wie eine Realisierung dieser Transaktion (T1) mit einer konkurrierenden Transaktion (T2) in einer Mehrbenutzerumgebung aussehen kann. Beide Transaktionen laufen im Modus `READ COMMITTED`. Man beachte, dass T2 abgebrochen werden würde, wenn T2 im Modus `SERIALIZABLE` gestartet worden wäre.

□

Für den Fall, dass die automatisch gesetzten Sperren in einer konkreten Situation nicht ausreichen, können in PostgreSQL (und auch in anderen DBMS) manuelle Sperren gesetzt werden. Zum Setzen einer Lese-Sperre verwende man

```
LOCK TABLE tabelle IN SHARE MODE;
```

für eine Exklusiv-Sperre

```
LOCK TABLE tabelle IN EXCLUSIVE MODE;
```

Man beachte, dass es kein entsprechendes `UNLOCK TABLE`-Statement gibt; alle Sperren bleiben bis zum Transaktionsende bestehen. Aus diesem Grund ist `LOCK TABLE` auch nur im Transaktionskontext sinnvoll, da ein Autocommit nach einem `LOCK TABLE`-Statement die Sperre sofort wieder aufheben würde.

5.3. Transaktionen und Constraints

Wie schon im Abschnitt 4.4.3 ausgeführt, kann die Gültigkeitsüberprüfung von Constraints jeweils am Ende eines Statements oder erst am Ende (COMMIT) der gesamten Transaktion stattfinden. Das

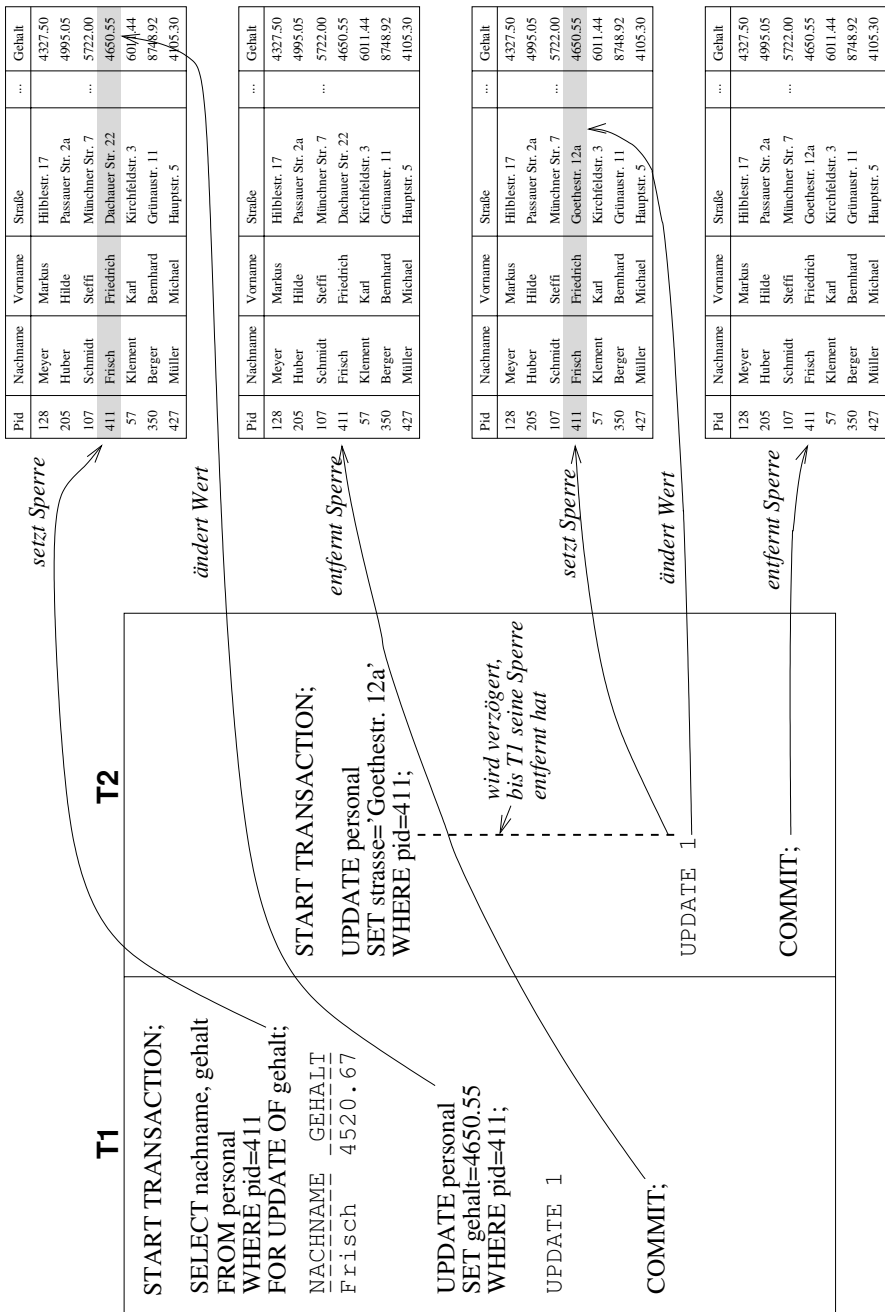


Abb. 57: Ablauf von zwei konkurrierenden Transaktionen

Standardverhalten wird constraintspezifisch bereits bei der Definition des Constraints mit `INITIALLY DEFERRED` bzw. `INITIALLY IMMEDIATE` festgelegt.

Will man in einer konkreten Transaktion das Standardverhalten aller oder bestimmter Constraints ändern, so kann man dies mit dem `SET CONSTRAINTS`-Statement erreichen, dessen Syntax in Abb. 58 dargestellt ist.

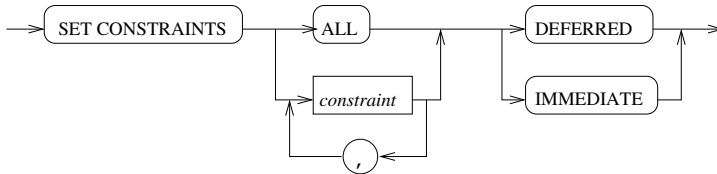


Abb. 58: Syntax des SET CONSTRAINTS-Statements

Hierbei ist zu beachten, dass ein Constraint nur dann im DEFERRED-Modus (am Ende einer Transaktion) geprüft werden kann, wenn dies bereits bei der Constraintdefinition mit der DEFERRABLE-Klausel (siehe Abschnitt 4.4.3) erlaubt wurde.

`SET CONSTRAINTS` bezieht sich nur auf die aktuelle Transaktion, nach dem Ende der Transaktion wird die Gültigkeitsprüfung von Constraints wieder wie bei der Constraintdefinition festgelegt vorgenommen.

5.4. Datenschutz und Zugriffsrechte

Datenbanksysteme ermöglichen die Verwirklichung von Datenschutzkonzepten über eine eigene Benutzerverwaltung. Jeder Benutzer, der mit der Datenbank arbeiten will, muss als Datenbank-Benutzer registriert sein.

Auch ein registrierter Datenbank-Benutzer hat zunächst nur Zugriff auf Objekte, die ihm “gehören” (das sind im Normalfall nur die, die er selbst angelegt hat). Er kann also beispielsweise nicht einfach Tabellen eines anderen Benutzers lesen oder gar ändern.

Sämtliche Rechte, die ein Datenbank-Benutzer innerhalb der Datenbank hat, werden mit Hilfe von **Privilegien** kontrolliert. Man unterscheidet zwischen **Systemprivilegien**, die die generelle Ausführung

von bestimmten Aktionen ermöglichen, und **Objektprivilegien**, die die Ausführung von bestimmten Aktionen auf einem *bestimmten* Objekt erlauben. Systemprivilegien können i. a. nur vom Datenbankverwalter vergeben werden, Objektprivilegien für jeweils eigene Objekte auch von jedem Datenbank-Benutzer. Privilegien werden entweder an bestimmte Benutzer oder an *alle* Benutzer (über den speziellen Benutzernamen PUBLIC) vergeben.

Man verwendet folgende Terminologie:

Grantor eines Privilegs: Benutzer, der das Privileg vergeben hat.

Grantee eines Privilegs: Benutzer, an den das Privileg vergeben wurde.

Systemprivilegien sind nicht standardisiert, so dass diese in jedem DBMS eigenständig implementiert sind. In PostgreSQL gibt es nur sehr wenige Arten von Systemprivilegien:

- Anmelderecht an einer Datenbank. Dieses Recht wird nicht über ein SQL-Statement vergeben, sondern über Konfigurationseinstellungen der PostgreSQL-Installation. Diese legen fest, welcher Benutzer sich von wo aus im Netz an welcher Datenbank anmelden darf und ob der Benutzer dazu ein Passwort benötigt.
- Anlegen neuer Datenbanken durch einen Benutzer. Hat ein Benutzer dieses Recht, so wird er automatisch Eigentümer für neue Datenbanken, die er selbst anlegt, und hat in diesen Datenbanken zunächst lokale Superuser-Rechte.
- Eintragen weiterer Benutzer durch einen Benutzer. Dieses Recht impliziert globale Superuser-Rechte, d. h. der Benutzer darf alle SQL-Aktionen in allen Datenbanken der Installation ausführen.

Im Normalfall ist derjenige Benutzer, der ein Objekt erzeugt, automatisch Eigentümer dieses Objekts. Der Datenbank-Superuser darf aber bei bestimmten Objekten (z. B. Datenbanken, Schemata) den Eigentümer explizit setzen.

Objektprivilegien, die an ein bestimmtes Objekt gebunden sind, dürfen von jedem Benutzer für seine *eigenen* Objekte (z. B. Tabellen und Views) vergeben werden. Außerdem darf ein Benutzer seine Privilegien für Objekte an andere Benutzer “weitervererben”, für die ihm dieses Recht vom Eigentümer des betreffenden Objekts eingeräumt wurde.

Es gibt folgende Arten von Objektprivilegien:

DELETE Grantee darf Tabelleneinträge mit DELETE FROM löschen.

INSERT	Grantee darf Tabelleneinträge mit INSERT INTO hinzufügen. Dieses Privileg kann im SQL-Standard auch selektiv für bestimmte Tabellenspalten vergeben werden, in PostgreSQL bezieht es sich grundsätzlich auf eine ganze Tabellenzeile.
REFERENCES	Grantee darf einen Constraint definieren, der die Tabelle als Eltern-Tabelle benutzt. Dieses Privileg kann im SQL-Standard auch selektiv für bestimmte Tabellenspalten vergeben werden, in PostgreSQL bezieht es sich grundsätzlich auf die gesamte Tabelle.
RULE	Grantee darf eine Regel für die Tabelle bzw. View erzeugen (siehe Abschnitt 3.6.4). Da es Regeln nur in PostgreSQL gibt, ist dieses Privileg nicht im SQL-Standard enthalten.
SELECT	Grantee darf die Tabelle mit SELECT abfragen. Im SQL-Standard kann dieses Privileg auch nur für bestimmte Tabellenspalten vergeben werden.
TRIGGER	Grantee darf einen Trigger (siehe Abschnitt 6.4) auf die Tabelle erzeugen.
UPDATE	Grantee darf Tabelleneinträge mit UPDATE ändern. Dieses Privileg kann im SQL-Standard auch selektiv für bestimmte Tabellenspalten vergeben werden, in PostgreSQL bezieht es sich grundsätzlich auf eine ganze Tabellenzeile.

Die Vergabe von Objektprivilegien erfolgt mit GRANT (Abb. 59), das Entziehen mit REVOKE (Abb. 60).

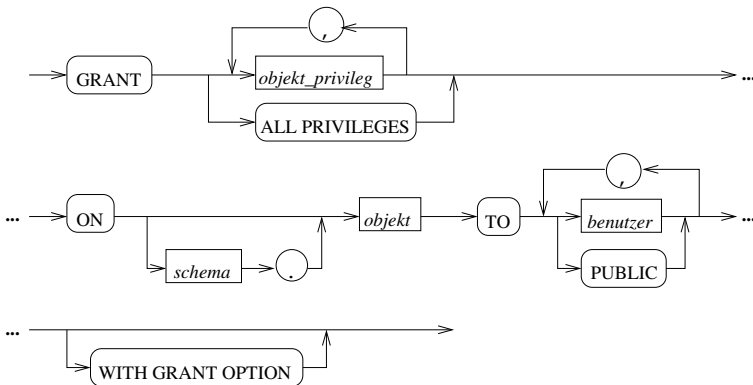


Abb. 59: Syntax des GRANT-Statements

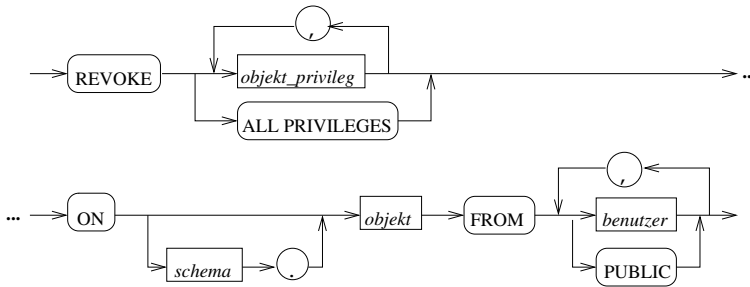


Abb. 60: Syntax des REVOKE-Statements

Hierbei bedeutet:

ALL PRIVILEGES	Vergabe aller zulässigen Privilegien.
<i>objekt</i>	Objekt, für das die Privilegien vergeben werden.
<i>benutzer</i>	Benutzer, an die die Privilegien vergeben werden (Grantees).
PUBLIC	Privilegienvergabe an alle Benutzer.
WITH GRANT OPTION	Privilegien, die der Grantee mit dieser Option erhalten hat, kann er selber an andere Benutzer mit GRANT weitergeben. Ohne diese Option können Privilegien nicht vererbt werden.

Ähnliches gilt für die Entziehung von Privilegien mit REVOKE.

Bei Views ist zu beachten, dass lediglich der Eigentümer einer View Privilegien auf den Basistabellen haben muss. Der Grantee von Privilegien für eine View benötigt keine Privilegien auf den Basistabellen der View. Damit lassen sich selektive Schutzmechanismen implementieren, die alleine durch Vergabe von Privilegien nicht erreicht werden können, wie folgendes Beispiel zeigt.

Beispiel 5.3.: In der Datenbank mit den Tabellen PERSONAL, ABTEILUNG, PROJEKT und ZUORDNUNG gibt es die Benutzer FIRMA, LOHNBUERO, PROJEKTMGR und PERSONALVERW. Alle Tabellen gehören dem Benutzer FIRMA. Der Datenschutz soll nun durch folgende Vorgaben realisiert werden:

- Der Benutzer PROJEKTMGR soll SELECT-Zugriff auf die Spalten Id, Nachname, Vorname der Tabelle PERSONAL haben. Er soll alle Privilegien auf den Tabellen PROJEKT und ZUORDNUNG besitzen.

- Der Benutzer LOHNBUERO soll auf die Spalten Id, Nachname, Vorname und Gehalt SELECT-Zugriff haben, außerdem UPDATE-Zugriff auf die Spalte Gehalt.
- Der Benutzer PERSONALVERW soll SELECT-, INSERT-, UPDATE- und DELETE-Zugriff auf die gesamte Tabelle PERSONAL haben. Darüber hinaus soll PERSONALVERW einen SELECT-Zugriff auf die Tabelle ABTEILUNG erhalten.

Der Benutzer FIRMA erzeugt nun zur Implementierung der selektiven Spaltenzugriffe mit SELECT zwei Views (Abb. 61):

```
CREATE VIEW pbdaten
```

```
AS SELECT pid, nachname, vorname FROM personal;
```

und

```
CREATE VIEW pgehdaten
```

```
AS SELECT pid, nachname, vorname, gehalt FROM personal;
```

Tabelle PERSONAL

Pid	Nachname	Vorname	Straße	Ort	Einstellung	Gehalt	Vorges_Id	Aid
128	Meyer	Markus	Hilblestr. 17	München	19.01.1994	4327.50	107	8
205	Huber	Hilde	Passauer Str. 2a	Augsburg	27.05.1991	4995.05	57	5
107	Schmidt	Steffi	Münchner Str. 7	Freising	02.11.1990	5722.00	350	8
411	Frisch	Friedrich	Dachauer Str. 22	München	14.09.1995	4520.67	107	8
57	Klement	Karl	Kirchfeldstr. 3	Bad Tölz	04.10.1990	6011.44	350	5
350	Berger	Bernhard	Grünaustr. 11	München	28.05.1993	8748.92	(NULL)	1
427	Müller	Michael	Hauptstr. 5	München		4105.30	(NULL)	(NULL)

View PBDATEN

Pid	Nachname	Vorname	Straße	Ort	Einstellung	Gehalt	Vorges_Id	Aid
-----	----------	---------	--------	-----	-------------	--------	-----------	-----

View PGEHDATEN

Pid	Nachname	Vorname	Straße	Ort	Einstellung	Gehalt	Vorges_Id	Aid
-----	----------	---------	--------	-----	-------------	--------	-----------	-----

Abb. 61: Erzeugen von Views zum “Verstecken” von Spalten

Damit der UPDATE-Zugriff in der View PGEHDATEN nur auf die Spalte GEHALT erfolgen kann, definieren wir eine spezielle Regel:

```
CREATE RULE upd AS ON UPDATE TO pgehdaten DO INSTEAD
UPDATE personal SET gehalt = NEW.gehalt
WHERE pid = OLD.pid
```

```
AND nachname = OLD.nachname
```

```
AND vorname = OLD.vorname;
```

Man beachte, dass durch diese Regel Änderungsversuche an anderen Spalten der View stillschweigend ignoriert werden.

Die Privilegien werden folgendermaßen vergeben:

```
GRANT SELECT ON pbdaten TO projektmgr;  
GRANT ALL PRIVILEGES ON projekt TO projektmgr;  
GRANT ALL PRIVILEGES ON zuordnung TO projektmgr;  
GRANT SELECT ON pgehdaten TO lohnbuero;  
GRANT UPDATE ON pgehdaten TO lohnbuero;  
GRANT SELECT, INSERT, UPDATE, DELETE ON personal  
    TO personalverw;  
GRANT SELECT ON abteilung TO personalverw;
```

Nun wird eine neue Mitarbeiterin, Paula Prechtel, wohnhaft in Passau, Wiener Str. 27, eingestellt. Sie erhält die Personalnummer 503, den Vorgesetzten Nr. 57 und soll in der Abteilung Nr. 5 arbeiten. Ihr Gehalt beträgt EUR 4490.30.

Die PERSONALVERWALTUNG wird die neue Mitarbeiterin zunächst mit folgendem Befehl in die Datenbank aufnehmen:

```
INSERT INTO personal  
    (pid,nachname,vorname,strasse,ort,einstellung,  
     vorges_id,aid)  
VALUES (503,'Prechtel','Paula','Wiener Str. 27',  
        'München', CURRENT_DATE, 57, 5);
```

einfügen. Sodann setzt das LOHNBUERO das Gehalt in die View PGEHDATEN ein:

```
UPDATE pgehdaten  
SET gehalt = 4490.30 WHERE pid = 503;
```

Der Benutzer LOHNBUERO kann keine weiteren Änderungen an der Personal-Tabelle vornehmen.



Man kann mit Views nicht nur den Zugriff auf Spalten, sondern auch auf Zeilen einer Tabelle beschränken.

Beispiel 5.4.: Der Leiter von Projekt Nr. 11 sei als Benutzer P11MGR in der Datenbank eingetragen. Er soll Lesezugriff auf Personalnummer, Nachname und Vorname nur derjenigen Mitarbeiter erhalten, die im Projekt Nr. 11 arbeiten. Dies erreichen wir durch Anlegen einer View, etwa mit

```
CREATE VIEW daten AS  
    SELECT personal.pid, nachname, vorname  
    FROM personal NATURAL JOIN zuordnung  
    WHERE projid = 11;
```

und Vergabe von Leserechten für P11MGR auf diese View:

`GRANT SELECT ON daten TO p11mgr;`

Die dann vorliegende Situation ist in Abb. 62 dargestellt.

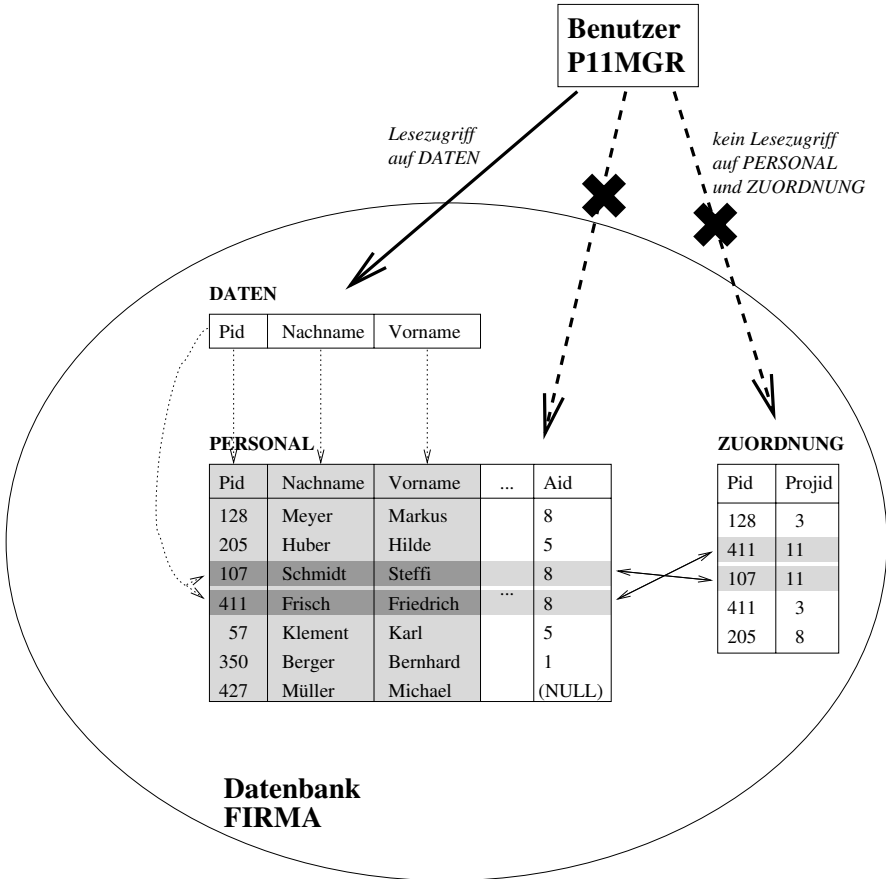


Abb. 62: Datenschutz durch selektiven Tabellenzugriff über Views

Über die View DATEN ist dem Benutzer P11MGR nur Zugriff auf die dunkel hinterlegten Elemente der Tabelle PERSONAL möglich, obwohl die View selbst auf alle Datensätze der Tabellen PERSONAL und ZUORDNUG zugreifen kann (und muss).

□

5.5. Zugriff auf Datenbank-Metadaten

In einem SQL-standardkonformen DBMS (also auch in PostgreSQL) gibt es ein spezielles Schema `IMPLEMENTATION_SCHEMA`, das (lesenden) Zugriff auf die Informationen über die in der Datenbank enthaltenen Objekte und ihre Eigenschaften erlaubt (sog. *Metadaten* der Datenbank). Die Objekte des `INFORMATION_SCHEMA` sind Views, die auf tatsächlichen (datenbankspezifischen) Systemtabellen aufbauen und erlauben so einen DBMS-unabhängigen Zugriff auf Metadaten.

Unter anderem gibt es dort folgende Views:

- `TABLES`: Tabellen, die dem Benutzer zugänglich sind.
- `VIEWS`: Views, die dem Benutzer zugänglich sind.
- `COLUMNS`: Tabellenspalten, die dem Benutzer zugänglich sind.
- `TABLE_PRIVILEGES`: Privilegien auf zugänglichen Tabellen.
- `COLUMN_PRIVILEGES`: Privilegien auf zugänglichen Tabellenspalten.

Informationen über die Spalten dieser Views und ihre Bedeutung finden sich in Teil 11 des SQL:2003-Standards sowie in der Dokumentation des betreffenden DBMS.

6. Modulare Erweiterungen von SQL

Bekanntlich ist SQL eine nichtprozedurale Sprache. SQL-Statements werden nacheinander abgearbeitet. Bei dem Teil von SQL, den wir bis jetzt behandelt haben (das sogenannte **interaktive SQL**), gibt es keine direkte Kommunikation zwischen SQL-Statements (außer über Daten in Tabellen).

Außer der interaktiven Variante gibt es auch das **modulare SQL**, das prozedurale Konzepte, wie sie von konventionellen Programmiersprachen bekannt sind, einführt. In PostgreSQL besteht für die modulare Variante auch die Möglichkeit, Module, die in anderen Programmiersprachen formuliert sind, einzubinden.

Die modularen Erweiterungen von PostgreSQL sind recht nahe verwandt mit der in Teil 4 des SQL:2003-Standards definierten Erweiterung SQL/PSM, jedoch syntaktisch nicht kompatibel. Der Rest dieses Abschnitts ist daher PostgreSQL-spezifisch.

6.1. Benutzerdefinierte Funktionen

Im Gegensatz zu den in SQL bereits standardmäßig vorhandenen Funktionen (z. B. `ROUND`, `SUBSTRING` oder `LPAD`), die man einfach benutzen kann, müssen benutzerdefinierte Funktionen explizit definiert werden. Sie sind ab dann als Objekte in der Datenbank vorhanden und können wie Standardfunktionen in Ausdrücken verwendet werden.

(Benutzerdefinierte) Funktionen⁷ können in PostgreSQL in verschiedenen Sprachen spezifiziert werden, u. a. in gewöhnlichem SQL sowie der prozeduralen Erweiterung PL/pgSQL. Im Rahmen dieses Buches wollen wir nur auf modulare Konstrukte im Rahmen von Funktionen eingehen, die in SQL selbst oder der prozeduralen Variante PL/pgSQL formuliert sind.

Funktionen werden mit dem `CREATE FUNCTION`-Statement angelegt; das Syntaxdiagramm dazu findet sich in Abb. 63.

Hier bedeuten:

<code>OR REPLACE</code>	Neudefinition einer bereits vorhandenen Funktion. Dies ist insbesondere in der Entwicklungsphase praktisch, da man damit die bereits vor-
-------------------------	---

⁷ Wenn wir im folgenden von “Funktion” sprechen, meinen wir immer die benutzerdefinierte Variante.

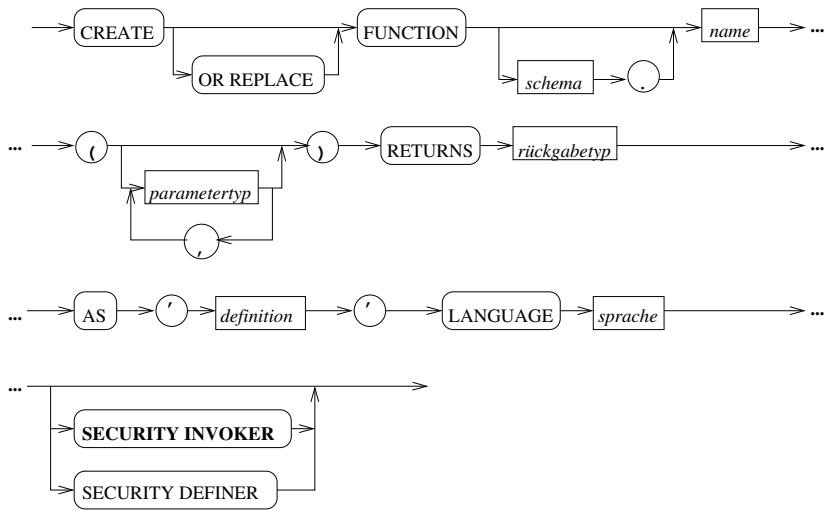


Abb. 63: Syntax des CREATE FUNCTION-Statements

	handene Funktion gleichen Namens nicht erst mit DROP FUNCTION löschen muss.
<i>parametertyp</i>	Datentyp des betreffenden Parameters; das kann entweder ein gewöhnlicher SQL-Datentyp (siehe Abschnitt 3.3.4) sein oder der Typ einer bereits existierenden Tabellenspalte, der mit <i>tabellenspalte%TYPE</i> spezifiziert wird.
<i>rückgabety</i>	Datentyp des Rückgabewertes. Dieser kann wie ein <i>parametertyp</i> spezifiziert sein; zusätzlich gibt es den speziellen Rückgabety <code>void</code> , wenn die Funktion keinen Rückgabewert liefert.
<i>definition</i>	Hier wird als Stringliteral der Code angegeben, der die Definition der Funktion (sog. <i>body</i>) bildet und bei Aufruf der Funktion ausgeführt wird. Wenn im Body der Funktion Anführungszeichen vorkommen, müssen diese – da ja die Definition als Literal angegeben wird – doppelt geschrieben werden.
sprache	Kann u. a. <code>SQL</code> (siehe folgenden Abschnitt 6.2) oder <code>PLPGSQL</code> (siehe Abschnitt 6.3) sein.
SECURITY INVOKER	Die Funktion wird im Sicherheitskontext (d. h. mit den Privilegien) des Benutzers ausgeführt, der die Funktion aufruft (Standardeinstellung).

SECURITY DEFINER Die Funktion wird im Sicherheitskontext des Benutzers, der die Funktion definiert hat (“Eigentümer”), ausgeführt. Damit kann in der Funktion beispielsweise der Zugriff auf Objekte stattfinden, auf die der Benutzer, der die Funktion aufruft, keine Rechte hat, wohl aber der Eigentümer der Funktion.

6.2. Funktionen mit SQL-Statements

Bei Funktionen, die in der Sprache SQL definiert sind, besteht der Body aus nichts weiter als einer Folge von SQL-Statements. Funktionsparameter werden über die Pseudonamen \$1 für den ersten Parameter, \$2 für den zweiten Parameter usw. angesprochen.

Die Ergebnisliste des letzten SQL-Statements bildet den Rückgabewert – da wir uns im Rahmen dieses Buches auf Rückgabewerte der vorgestellten skalaren SQL-Datentypen beschränken, muss das letzte SQL-Statement eine SELECT-Abfrage sein und eine Ergebnistabelle liefern, die nur aus einer Spalte besteht. Eine Ausnahme bilden SQL-Funktionen mit Rückgabotyp `void`: Hier darf das letzte SQL-Statement in der Funktion keine SELECT sein, da ja *kein* Wert zurückgeliefert werden soll.

Beispiel 6.1.: Wir formulieren eine Funktion `mitproabt`, die die Anzahl der Mitarbeiter in der durch den Parameter gegebenen Abteilungsnummer liefert:

```
CREATE FUNCTION mitproabt(INTEGER)
RETURNS BIGINT
AS '
    SELECT COUNT(*) FROM personal WHERE aid = $1;
' LANGUAGE SQL;
```

Der Rückgabewert ist hier `BIGINT` und nicht etwa `INTEGER`, da die `COUNT`-Funktion einen Wert vom Typ `BIGINT` liefert. Nach Definition dieser Funktion liefert etwa

```
SELECT mitproabt(8);
```

das Ergebnis 3, während

```
SELECT mitproabt(7);
```

den Wert 0 liefert.

□

Besteht die Ergebnistabelle des abschließenden SELECT nicht genau aus einer Zeile, so wird, wenn mehr als eine Zeile geliefert wird, die

erste Zeile (der erste Wert) als Rückgabewert verwendet. Da die Reihenfolge der Zeilen in der Ergebnistabelle einer SQL-Abfrage nicht festgelegt ist, wenn sie nicht explizit sortiert wird, sollte im Zweifelsfall immer eine ORDER BY-Klausel verwendet werden. Liefert das abschließende SELECT überhaupt keine Zeile (keinen Wert), so wird als Rückgabewert NULL geliefert.

Beispiel 6.2.: Die folgende Funktion `mitabtblist` liefert für eine gegebene Abteilungsnummer die Nummer des Mitarbeiters mit der kleinsten Personalnummer, der dieser Abteilung zugeordnet ist.

```
CREATE FUNCTION mitabtblist(integer)
RETURNS integer
AS '
    SELECT pid FROM personal WHERE aid = $1 ORDER BY 1;
' LANGUAGE SQL;
```

Dann liefert

```
SELECT mitabtblist(8);
```

das Resultat

```
107
```

aber

```
SELECT mitabtblist(7) IS NULL;
```

liefert TRUE, da ja kein Mitarbeiter in Abteilung Nr. 7 arbeitet, die Funktion daher NULL zurückgibt und der Test auf NULL erfüllt ist. □

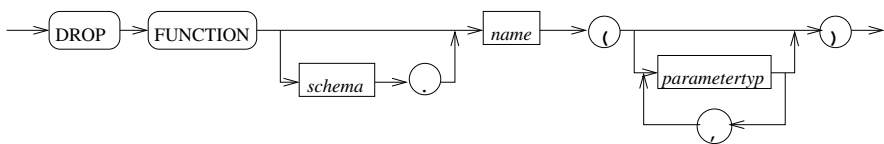


Abb. 64: Syntax des DROP FUNCTION-Statements

Funktionen kann man mit `DROP FUNCTION` (Abb. 64) wieder löschen; hier sind zusätzlich zum Namen der Funktion auch die Typen der Parameter (wie bei der Definition) anzugeben, da es in PostgreSQL (wie z. B. auch in C++) erlaubt ist, verschiedene Funktionen mit unterschiedlicher Parameterisierung unter dem gleichen Namen zu definieren (sog. **Overloading**).

6.3. PL/pgSQL

Bei PL/pgSQL handelt es sich um eine prozedurale Sprache für ein PostgreSQL-Datenbanksystem. Die Sprache kann als Erweiterung von SQL angesehen werden, da die bekannten SQL-Datentypen und die meisten SQL-Konstrukte auch in PL/pgSQL-Programmen verwendet werden können.

Wir wollen im folgenden einen Einblick in die Möglichkeiten von PL/pgSQL geben, ohne auf jedes einzelne Detail einzugehen. Eine detaillierte Beschreibung der Sprache findet sich in der PostgreSQL-Dokumentation [Pg2004].

Man beachte, dass die Sprachkonstrukte von PL/pgSQL in einer PostgreSQL-Datenbank nur zur Verfügung stehen, wenn die Sprache vom Datenbankadministrator freigeschaltet wurde.⁸

Wir setzen für die folgenden Ausführungen voraus, dass der Leser Basiskenntnisse in einer konventionellen Programmiersprache (PASCAL, C) besitzt.

PL/pgSQL ist eine blockstrukturierte Sprache: Ein PL/pgSQL-Programm (das nicht alleine, sondern nur als Body einer Funktion auftreten kann) besteht jeweils aus einem **Block**. Blöcke können wiederum beliebig viele verschachtelte Unterblöcke enthalten.

Ein PL/pgSQL-Block besteht aus folgenden Teilen in der angegebenen Reihenfolge:

- Deklarationsteil (optional): Dient zur Vereinbarung (Deklaration) von Variablen und Konstanten und wird mit dem Schlüsselwort `DECLARE` eingeleitet.
- Anweisungsteil: Enthält eine Folge von PL/pgSQL-Statements und SQL-Statements. Er wird mit dem Schlüsselwort `BEGIN` eingeleitet und mit dem Schlüsselwort `END`, gefolgt von einem Semikolon, beendet.

Jede Deklaration und jedes Statement in einem Block wird durch ein Semikolon abgeschlossen.

⁸ Wenn Sie mit einer eigenen Datenbankinstallation arbeiten, können Sie PL/pgSQL mit dem `createlang`-Utility in der Datenbank installieren - siehe dazu die PostgreSQL-Dokumentation [Pg2004].

6.3.1. Ein Beispiel

Wir wollen zunächst an einem einfachen Beispiel die – im wesentlichen herkömmlichen prozeduralen Programmiersprachen entsprechenden – Sprachkonstrukte von PL/pgSQL erläutern.

Beispiel 6.3.: Für die Mitarbeiter unserer Beispielfirma soll es eine Gehaltserhöhung geben, und zwar gestaffelt nach der Anzahl der Projekte, in denen ein Mitarbeiter tätig ist: Wer in keinem Projekt tätig ist, bekommt 0.5% Gehaltserhöhung, wer in einem Projekt tätig ist, 1%, wer in zwei oder mehr Projekten tätig ist, 2%.

Wir stellen hier zunächst eine Funktion `erhoehung1` vor, die diese Gehaltserhöhung nur für den Mitarbeiter mit der als Parameter übergebenen Personalnummer vornimmt (Abb. 65); der allgemeine Fall kann mit dem später erläuterten Cursor-Konzept gelöst werden.

```
1  CREATE OR REPLACE FUNCTION erhoehung1 (INTEGER)
2  RETURNS void AS '
3  DECLARE
4      pnum INTEGER;
5      factor NUMERIC(4,3);
6      for_id ALIAS FOR $1;
7  BEGIN
8      SELECT COUNT(projid) INTO pnum
9      FROM personal NATURAL JOIN zuordnung
10     WHERE personal.pid = for_id;
11     IF pnum = 0 THEN
12         factor := 1.005;
13     ELSIF pnum = 1 THEN
14         factor := 1.01;
15     ELSE
16         factor := 1.02;
17     END IF;
18     UPDATE personal
19     SET gehalt = gehalt * factor
20     WHERE pid = for_id;
21     RETURN;
22 END;
23 ' LANGUAGE plpgsql;
```

Abb. 65: PL/pgSQL-Programm zur Gehaltserhöhung für einen Mitarbeiter

Die Zeilen 3–6 bilden den Deklarationsteil zur Vereinbarung von Variablen und Konstanten. Als Datentypen bietet PL/pgSQL die SQL-Datentypen und folgende Konstrukte:

tabelle.spalte%TYPE entspricht dem Datentyp der angegebenen Tabellenspalte.

tabelle%ROWTYPE entspricht einem RECORD-Datentyp, wobei die Komponenten wie die Tabellenspalten heißen und den gleichen Datentyp wie die Tabellenspalten haben.

In Zeile 6 wird der Name `for_id` als Alias für den Funktionsparameter, der ja eigentlich mit `$1` angesprochen wird, vereinbart.

Die Zeilen 7–22 bilden den Anweisungsteil des Programms. In den Zeilen 8–10 wird Information in vereinbarte Variablen selektiert. `SELECT...INTO` speichert das Ergebnis in die angegebenen Variablen, anstatt es (wie bei einem gewöhnlichen SQL-`SELECT`) auf dem Bildschirm auszugeben.

Ähnlich wie bei der Rückgabe von Werten einer SQL-Funktion wird nur der erste Datensatz berücksichtigt, wenn die `SELECT`-Abfrage mehrere Datensätze liefern würde. Für den Fall, dass die `SELECT`-Abfrage kein Ergebnis liefert, werden die angegebenen Variablen auf `NULL` gesetzt. Die spezielle Systemvariable `FOUND` gibt nach Ausführung einer `SELECT...INTO`-Abfrage Aufschluss darüber, ob die Abfrage mindestens einen Datensatz geliefert hat – in diesem Fall hat `FOUND` den logischen Wert `true`, sonst `false`.

In den Zeilen 11–17 befindet sich eine `IF...THEN...ELSE`-Kaskade, die jeweils den Erhöhungsfaktor auf unterschiedliche Werte setzt.

Das `RETURN`-Statement in Zeile 21 ist wichtig, da PL/pgSQL-Funktionen immer mit `RETURN` beendet werden müssen. Wenn die Funktion ein Ergebnis liefern soll, muss als Argument von `RETURN` der gewünschte Wert angegeben werden; in unserem Fall einer `void`-Funktion geben wir nur `RETURN` an.

Zuweisungen an Variablen geschehen in PL/pgSQL mit

variable := *ausdruck*;

Bei Ausdrücken können die gleichen Operationen und Funktionen wie in interaktivem SQL verwendet werden (soweit dies im Einzelfall sinnvoll ist).



6.3.2. Kontrollstrukturen

Hierunter versteht man Sprachkonstrukte, die die normale Ausführungsreihenfolge von Anweisungen (sequentiell, also eine Anweisung nach der anderen) beeinflussen.

Bedingte Ausführung (Verzweigung):

1. Form: *IF bedingung THEN anweisungen*
 END IF;
2. Form: *IF bedingung THEN anweisungen*
 ELSE anweisungen
 END IF;
3. Form: *IF bedingung THEN anweisungen*
 ELSIF bedingung THEN anweisungen
 ELSE anweisungen
 END IF;

Einfache Schleife:

```
LOOP anweisungen END LOOP;
```

Das Verlassen der Schleife ist mit EXIT möglich:

```
EXIT;
```

führt immer zum Verlassen der Schleife,

```
EXIT WHEN bedingung;
```

nur dann, wenn die *bedingung* erfüllt (TRUE) ist.

WHILE-Schleife:

```
WHILE bedingung LOOP anweisungen END LOOP;
```

FOR-Zählschleife:

```
FOR zaehler IN untergrenze..obergrenze  
LOOP anweisungen  
END LOOP;
```

untergrenze und *obergrenze* müssen ganzzahlig sein.

Folgende Ersatzdarstellung entspricht obiger FOR-Schleife:

```
zaehler := untergrenze;  
WHILE zaehler <= obergrenze LOOP  
    anweisungen  
    zaehler := zaehler + 1;  
END LOOP;
```

6.3.3. Das Cursor-Konzept

Wie am Beispiel gesehen, ist SELECT...INTO nur dann sinnvoll, wenn man sich für höchstens einen Datensatz des Abfrageresultats

interessiert. Um Abfragen bearbeiten zu können, die mehrere Datensätze liefern, bedient man sich eines sogenannten **Cursors**. Ein Cursor ermöglicht die Bearbeitung eines Abfrageresultats Datensatz für Datensatz.

Angenommen, wir wollen das Ergebnis der Abfrage

```
SELECT pid, nachname FROM personal;
```

in PL/pgSQL bearbeiten. Dazu definieren wir im Deklarationsteil mit

```
c1 CURSOR FOR SELECT pid, nachname FROM personal;
```

einen Cursor mit dem Namen `c1`, sowie mit

```
p personal.pid%TYPE;
```

```
nn personal.nachname%TYPE;
```

Variablen, die die Werte jeweils eines Datensatzes der Ergebnistabelle aufnehmen können.

Auf einem Cursor stehen folgende Operationen zur Verfügung:

```
OPEN cursor;
```

führt die bei der Vereinbarung des Cursors angegebene Abfrage durch, und

```
FETCH cursor INTO variable, ..., variable;
```

holt jeweils den nächsten Datensatz der Ergebnistabelle der Abfrage und speichert ihn in den angegebenen Variablen.

```
CLOSE cursor;
```

verwirft eventuell noch verbleibende Datensätze der Ergebnistabelle und gibt damit den vom Cursor belegten Speicherplatz frei. Der Cursor kann jedoch dann mit `OPEN...FETCH...CLOSE` nochmals benutzt werden.

Das Cursorprinzip ist am Beispiel in Abb. 66 gezeigt.

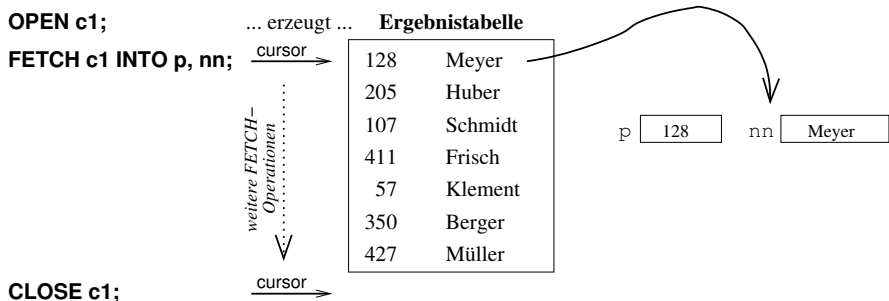


Abb. 66: Arbeitsweise eines PL/pgSQL-Cursors

Beispiel 6.4.: Wir schließen an das einführende Beispiel an und lösen das Gehaltserhöhungsproblem für alle Mitarbeiter. Offensichtlich kann der wesentliche Teil des in Abb. 65 gezeigten Programms übernommen werden, es muss lediglich noch ein Cursor definiert werden, der sämtliche Personalnummern liefert. Diese werden in einer Schleife abgearbeitet; innerhalb der Schleife wird dann festgestellt, in wie vielen Projekten der betreffende Mitarbeiter arbeitet, die prozentuale Höhe der Gehaltserhöhung berechnet und die tatsächliche Gehaltserhöhung vorgenommen. Das zugehörige Programm ist in Abb. 67 abgebildet.

```
1  CREATE OR REPLACE FUNCTION erhoehung2 () RETURNS void AS '  
2  DECLARE  
3      pnum INTEGER;  
4      factor NUMERIC(4,3);  
5      for_id INTEGER;  
6      cpnum CURSOR FOR SELECT pid FROM personal;  
7  BEGIN  
8      OPEN cpnum;  
9      LOOP  
10         FETCH cpnum INTO for_id;  
11         EXIT WHEN NOT FOUND;  
12         SELECT COUNT(projid) INTO pnum  
13         FROM personal NATURAL JOIN zuordnung  
14         WHERE personal.pid = for_id;  
15         IF pnum = 0 THEN  
16             factor := 1.005;  
17         ELSIF pnum = 1 THEN  
18             factor := 1.01;  
19         ELSE  
20             factor := 1.02;  
21         END IF;  
22         UPDATE personal  
23         SET gehalt = gehalt * factor  
24         WHERE pid = for_id;  
25     END LOOP;  
26     CLOSE cpnum;  
27     RETURN;  
28 END;  
29 ' LANGUAGE plpgsql;
```

Abb. 67: PL/pgSQL-Programm zur Gehaltserhöhung für alle Mitarbeiter

Man beachte, dass nach jeder FETCH-Operation die spezielle Variable FOUND Aufschluss darüber gibt, ob noch ein weiterer Datensatz vorhanden war. Dies nutzen wir, um in Zeile 11 ggf. die Schleife zu verlassen, weil alle Datensätze behandelt wurden.

□

Beispiel 6.5.: Finden des Namens und der Personalnummer des Mitarbeiters mit dem niedrigsten Gehalt. (Dieses Beispiel kann natürlich auch in interaktivem SQL programmiert werden. Es eignet sich jedoch wegen seiner leichten Überschaubarkeit auch für die Demonstration von PL/pgSQL-Konzepten.) Wir setzen dabei voraus, dass eine Tabelle

RESULT(pid INTEGER, name CHARACTER VARYING(20)) existiert, in die das Ergebnis geschrieben werden kann.⁹ Das Programm ist in Abb. 68 abgebildet.

In Zeile 6 wird eine Variable des speziellen Typs RECORD definiert, die eine ganze Tabellenzeile aufnehmen kann. RECORD-Variablen besitzen in PL/pgSQL bei der Deklaration noch keine festgelegte Struktur, diese ergibt sich erst bei einer Initialisierung der Variablen – hier in Zeile 15. Danach heißen die Komponenten von `my_rec` wie die Spalten der durch den Cursor `c1` implizierten Ergebnistabelle.

□

In den meisten Situationen lässt sich die Datensatzbearbeitung statt mit OPEN...FETCH...CLOSE auch mit einer sogenannten Query-FOR-Schleife durchführen. Diese Schleife hat folgendes Format:

```
FOR record_variable IN SELECT-Abfrage
LOOP anweisungen END LOOP;
```

Am Anfang der FOR-Schleife wird implizit ein Cursor mit der angegebenen Abfrage angelegt und geöffnet. Bei jedem Durchlauf wird ein Datensatz in die *record_variable* geholt. Wenn kein Datensatz mehr vorhanden ist, wird der Cursor geschlossen und die Schleife verlassen.

Beispiel 6.6.: Das letzte Beispielprogramm kann alternativ wie in Abb. 69 gezeigt formuliert werden.

□

⁹ Es gibt in PL/pgSQL auch die Möglichkeit, Wertetupel als Resultat einer Funktion abzuliefern; im Rahmen dieses Buches behandeln wir dies jedoch nicht. Darum benötigen wir für die Rückgabe des aus zwei Werten bestehenden Ergebnisses eine Tabelle.

```
1 CREATE OR REPLACE FUNCTION mingehl () RETURNS void AS '  
2 DECLARE  
3     c1 CURSOR FOR  
4         SELECT pid, nachname, gehalt  
5         FROM personal;  
6     my_rec RECORD;  
7     mingeh personal.gehalt%TYPE;  
8     minid personal.pid%TYPE;  
9     minname personal.nachname%TYPE;  
10 BEGIN  
11     DELETE FROM result;  
12     minid := 0;  
13     OPEN c1;  
14     LOOP  
15         FETCH c1 INTO my_rec;  
16         EXIT WHEN NOT FOUND;  
17         IF minid = 0 OR my_rec.gehalt < mingeh THEN  
18             minid := my_rec.pid;  
19             mingeh := my_rec.gehalt;  
20             minname := my_rec.nachname;  
21         END IF;  
22     END LOOP;  
23     CLOSE c1;  
24     IF minid <> 0 THEN  
25         INSERT INTO result VALUES (minid, minname);  
26     END IF;  
27     RETURN;  
28 END;  
29 ' LANGUAGE plpgsql;
```

Abb. 68: Benutzung eines Cursors
mit OPEN ... FETCH ... CLOSE

6.3.4. Fehlerbehandlung

Eine Funktion kann in vielen verschiedenen Kontexten aufgerufen werden (z. B. auch automatisch als *Trigger*, wie wir in Abschnitt 6.4 sehen werden), wo es nicht unbedingt die Möglichkeit gibt, eine Fehlersituation etwa durch Abfrage eines von der Funktion zurückgelieferten Fehlercodes festzustellen.

Daher gibt es in PL/pgSQL die Möglichkeit, eine sogenannte **Ausnahmebedingung** (engl. **Exception**) anzustoßen, die den Abbruch

```
1 CREATE OR REPLACE FUNCTION mingeh2 () RETURNS void AS '  
2 DECLARE  
3     my_rec RECORD;  
4     mingeh personal.gehalt%TYPE;  
5     minid personal.pid%TYPE;  
6     minname personal.nachname%TYPE;  
7 BEGIN  
8     DELETE FROM result;  
9     minid := 0;  
10    FOR my_rec IN  
11        SELECT pid, nachname, gehalt  
12        FROM personal  
13    LOOP  
14        IF minid = 0 OR my_rec.gehalt < mingeh THEN  
15            minid := my_rec.pid;  
16            mingeh := my_rec.gehalt;  
17            minname := my_rec.nachname;  
18        END IF;  
19    END LOOP;  
20    IF minid <> 0 THEN  
21        INSERT INTO result VALUES (minid, minname);  
22    END IF;  
23    RETURN;  
24 END;  
25 ' LANGUAGE plpgsql;
```

Abb. 69: Implizite Benutzung eines Cursors mit einer FOR-Schleife

der Funktion und – soweit die Funktion in einem Transaktionskontext ausgeführt wird – der gesamten Transaktion bewirkt.

Hierzu dient das Statement

RAISE EXCEPTION *stringliteral*, *variable*, ...;

Das angegebene *stringliteral* ist ein Text, der an die aufrufende Instanz zurückgeliefert wird, wobei in diesem Literal vorkommende Prozentzeichen (%) jeweils durch den entsprechenden Wert der angegebenen *variable* ersetzt werden.

Ist die aufrufende Instanz etwa ein interaktiver SQL-Kommandozeileninterpreter, so wird im Normalfall der Text auf dem Bildschirm ausgegeben.

Beispiel 6.7.: Die in Abb. 70 gezeigte Funktion `which_project` liefert für die als Parameter übergebene Personalnummer den Namen

des Projekts, dem der betreffende Mitarbeiter zugeordnet ist. Ist der Mitarbeiter keinem oder mehr als einem Projekt zugeordnet, wird die Funktion mit einer Ausnahmebedingung abgebrochen.

```
1  CREATE OR REPLACE FUNCTION which_project (INTEGER)
2  RETURNS TEXT AS '
3  DECLARE
4      look_for ALIAS FOR $1;
5      cnt INTEGER;
6      pn TEXT;
7  BEGIN
8      SELECT COUNT(*) INTO cnt FROM zuordnung
9      WHERE pid = look_for;
10     IF cnt > 1 THEN
11         RAISE EXCEPTION ''Zu viele Projekte (%)'', cnt;
12     END IF;
13     IF cnt = 0 THEN
14         RAISE EXCEPTION ''Kein Projekt'';
15     END IF;
16     SELECT name INTO pn
17     FROM projekt NATURAL JOIN zuordnung
18     WHERE pid = look_for;
19     RETURN pn;
20 END;
21 ' LANGUAGE plpgsql;
```

Abb. 70: Programm mit Ausnahmebedingungen

Erfolgt ein interaktiver Aufruf von `which_project` mit

```
SELECT which_project(128);
```

so wird der String 'Druckauftrag Fa. Karl' zurückgeliefert. Der Aufruf

```
SELECT which_project(411);
```

resultiert in der Fehlermeldung 'Zu viele Projekte (2)', während

```
SELECT which_project(350);
```

die Fehlermeldung 'Kein Projekt' generiert.



Man beachte, dass die durch `RAISE` generierten Fehlermeldungen keinesfalls Rückgabewerte der Funktion sind. Eine durch `RAISE` erzeugte Exception bewirkt den sofortigen Abbruch der Funktion, so dass diese auch keinen Rückgabewert mehr liefern kann.

6.3.5. Hierarchische Strukturen und rekursive Funktionen

In manchen Fällen enthalten Tabellen Daten, die eine hierarchische Struktur darstellen. In unserer Musterdatenbank ist dies bei der Tabelle PERSONAL der Fall. Hier ist durch die Attribute Id und Vorges_Id die in Abb. 71 dargestellte Hierarchie “ist Vorgesetzter von” gegeben.

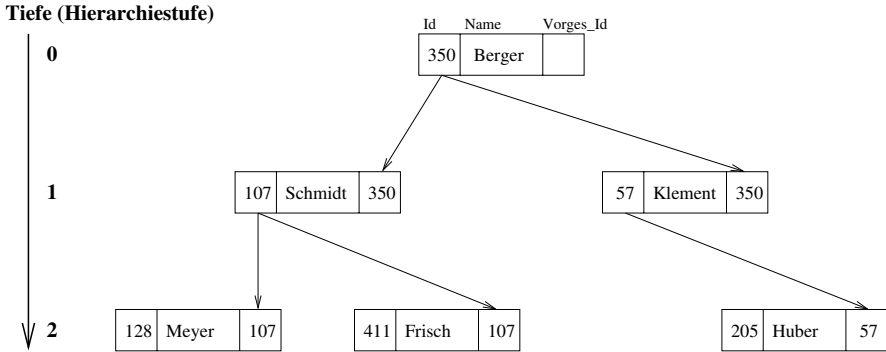


Abb. 71: Hierarchische Struktur in der PERSONAL-Tabelle

Eine hierarchische Struktur zeichnet sich durch folgende Eigenschaften aus:

- Zu einem Datensatz existieren ein oder mehrere Eltern-Datensätze (Vorgängerdatensätze). Das Vorliegen einer Eltern-Kind-Beziehung zwischen zwei Datensätzen kann aus den Attributen der Datensätze ermittelt werden.
- Es gibt einen oder auch mehrere Datensätze, die keine Eltern-Datensätze besitzen. Diese werden Wurzeln der Hierarchie genannt.
- Ein Datensatz r_1 , der Nachkomme eines Datensatzes r_2 ist, kann nicht gleichzeitig Vorfahre von r_2 sein.

Mathematisch entspricht eine hierarchische Struktur einem gerichteten kreisfreien Graphen.

In der Tabelle PERSONAL sind die Wurzeln der Hierarchie die Mitarbeiter, die keinen Vorgesetzten haben, und ein Mitarbeiter r_1 ist Vorgänger des Mitarbeiters r_2 , wenn sein Attribut Pid mit dem Attribut Vorges_Id von r_2 übereinstimmt.

In der Praxis tauchen oft verschiedene Fragestellungen im Zusammenhang mit diesen hierarchischen Strukturen auf. Bei unserer PERSONAL-Tabelle können dies etwa folgende Probleme sein:

- a) Hierarchiestufe eines Mitarbeiters, d. h. Anzahl der ihm übergeordneten Vorgesetzten.
- b) Anzahl der direkten und indirekten Untergebenen eines Mitarbeiters.

Die Antwort auf Problem a) lässt sich folgendermaßen finden:

- Alle Mitarbeiter, bei denen das Attribut `vorges_id` NULL ist, befinden sich auf Hierarchiestufe 0.
- Sei r_1 Vorgesetzter von r_2 . Dann ist die Hierarchiestufe von r_2 um 1 größer als die Hierarchiestufe von r_1 .

Um die Hierarchiestufen aller Mitarbeiter zu ermitteln, ist also ein rekursiver Durchlauf durch die PERSONAL-Tabelle gemäß der Mitarbeiter-Vorgesetzten-Beziehung erforderlich.

Beispiel 6.8.: Mit der Funktion `hstufe` soll die Hierarchiestufe des Mitarbeiters mit der als Parameter übergebenen Nummer bestimmt und zurückgeliefert werden. Der Code dieser Funktion ist in Abb. 72 dargestellt. Diese Funktion setzt genau das rekursive Prinzip, das oben für Problem a) gegeben wurde, um.

```
1 CREATE OR REPLACE FUNCTION hstufe (INTEGER)
2 RETURNS INTEGER AS '
3 DECLARE
4     p ALIAS FOR $1;
5     v personal.vorges_id%TYPE;
6 BEGIN
7     SELECT vorges_id INTO v
8     FROM personal
9     WHERE pid = p;
10    IF NOT FOUND THEN RETURN NULL;
11    END IF;
12    IF v IS NULL THEN RETURN 0;
13    ELSE RETURN hstufe(v) + 1;
14    END IF;
15 END;
16 ' LANGUAGE plpgsql;
```

Abb. 72: Bestimmung der Hierarchiestufe eines Mitarbeiters

Die Funktion gibt NULL zurück, falls eine nicht existierende Personalnummer übergeben wurde. Die in Abb. 73 zu sehende Graphik zeigt die rekursive Aufrufstruktur beim Aufruf von

`SELECT hstufe(205);`

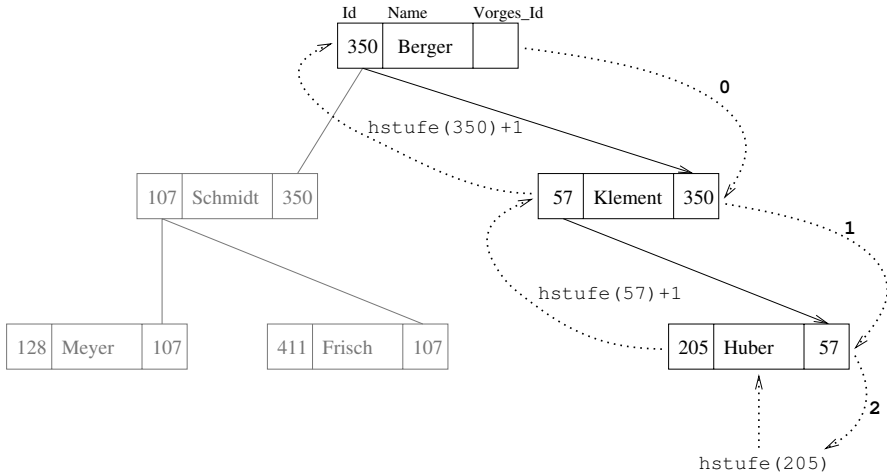


Abb. 73: Rekursive Aufrufstruktur bei der Bestimmung der Hierarchiestufe

Wir könnten nun die Hierarchiestufe für alle Mitarbeiter unter Verwendung der im letzten Beispiel definierten Funktion mit

`SELECT pid, hstufe(pid) FROM personal;`

ermitteln. Dies ist jedoch äußerst ineffizient, da hierbei für jeden Mitarbeiter immer wieder die Hierarchie bis zu einem “Chef” zurückverfolgt wird. Dies ist jedoch gar nicht notwendig, da – wie wir im letzten Beispiel gesehen haben – bei der Berechnung der Hierarchiestufe eines Mitarbeiters automatisch auch die Hierarchiestufen der übergeordneten Mitarbeiter ermitteln.

Es empfiehlt sich daher eine andere Vorgehensweise, die wir im folgenden vorstellen wollen: Ausgehend von den “Wurzelknoten” (Mitarbeiter mit NULL-Attributwert von `vorges_id`) besuchen wir jeden Knoten (Datensatz) einmal; die jeweilige Hierarchiestufe ergibt sich dann aus der um 1 erhöhten Stufe des Vorgängers. Die Knotenreihenfolge muss dann so gewählt werden, dass ein Knoten erst dann besucht wird, wenn der jeweilige Vorgänger bereits besucht wurde.

Eine Reihenfolge, die diese Bedingung erfüllt, ist die sog. **depth-first-Ordnung**, die für unsere konkrete PERSONAL-Tabelle in Abb. 74 dargestellt ist.

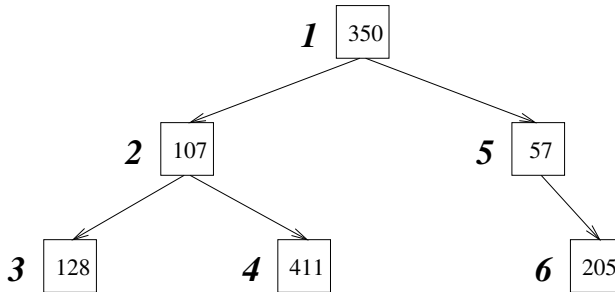


Abb. 74: Depth first-Ordnung in der PERSONAL-Tabelle

Für das folgende Beispiel nehmen wir an, dass in unserer Datenbank eine Tabelle

STUFE (pid INTEGER, level INTEGER)

definiert ist.

Beispiel 6.9.: Mit den in Abb. 75 dargestellten Funktionen `hstufe_all()` und `hstufe_all(INTEGER, INTEGER)`¹⁰ wird bei Aufruf von

```
SELECT href_all ();
```

in die Tabelle STUFE für jeden Mitarbeiter ein Datensatz mit Personalnr. und Hierarchiestufe geschrieben.

Die erste Funktion ohne Parameter die sozusagen zur “Initialisierung”, indem die Hierarchiestufen für die “Chefs” festgelegt werden. Dann wird jeweils für alle direkten Untergebenen die zweite Funktion mit Parametern aufgerufen.

`hstufe_all(parent, plevel)` besucht alle Knoten der Struktur ab Wurzelknoten *parent* auf Hierarchiestufe *plevel*. Man beachte, dass die Aufrufe dieser Funktion in den Zeilen 12 und 29 jeweils über ein PERFORM-Statement erfolgen müssen. PERFORM verhält sich in PL/pgSQL wie SELECT INTO mit dem Unterschied, dass ein eventuelles Ergebnis der SELECT-Abfrage verworfen wird. Da unsere Funk-

¹⁰ Dies ist ein Beispiel für das auf S. 115 beschriebene Overloading von Funktionsnamen.

```
1  CREATE OR REPLACE FUNCTION hstufe_all ()
2  RETURNS void AS '
3
4  DECLARE
5      p RECORD;
6
7  BEGIN
8      DELETE FROM stufe;
9
10     FOR p IN
11         SELECT pid FROM personal
12         WHERE vorges_id IS NULL
13     LOOP
14         INSERT INTO stufe VALUES (p.pid, 0);
15         PERFORM hstufe_all (p.pid, 0);
16     END LOOP;
17     RETURN;
18 ' LANGUAGE plpgsql;
19
20 CREATE OR REPLACE FUNCTION hstufe_all (INTEGER, INTEGER)
21 RETURNS void AS '
22
23 DECLARE
24     parent ALIAS FOR $1;
25     plevel ALIAS FOR $2;
26     p RECORD;
27
28 BEGIN
29     FOR p IN
30         SELECT pid FROM personal
31         WHERE vorges_id = parent
32     LOOP
33         INSERT INTO stufe VALUES (p.pid, plevel+1);
34         PERFORM hstufe_all (p.pid, plevel+1);
35     END LOOP;
36     RETURN;
37 END;
38 ' LANGUAGE plpgsql;
```

Abb. 75: Bestimmung der Hierarchiestufe für alle Mitarbeiter

tion ohnehin keinen Wert zurückliefert, muss sie in jedem Fall mit `PERFORM` aufgerufen werden.

□

Beispiel 6.10.: Führt man die im letzten Beispiel definierte Funktion `hstufe_all()` mit

```
SELECT hstufe_all ();
```

aus, so enthält die Tabelle `STUFE` dann für jeden Mitarbeiter die Personalnummer und die Hierarchiestufe, d. h. für unsere Beispieltabelle liefert

```
SELECT * FROM stufe;
```

dann das Resultat

pid	level
350	0
107	1
128	2
411	2
57	1
205	2

□

Betrachten wir nun das oben unter b) geschilderte Problem, die Anzahl der direkten und indirekten Untergebenen eines Mitarbeiters zu finden. Hier geht man – offensichtlich wieder rekursiv – folgendermaßen vor:

- Für einen Mitarbeiter r , der kein Vorgesetzter ist (d. h. dessen Personalnummer nicht als Attributwert von `vorges_id` auftaucht) ist der gesuchte Wert gleich 0.
- Angenommen, der Mitarbeiter r ist Vorgesetzter der Mitarbeiter r_1, \dots, r_n . Dann ist der gesuchte Wert gleich der Summe von

(Anzahl der direkten oder indirekten Untergebenen von r_i) + 1

über $i = 1, \dots, n$ aufsummiert.

Beispiel 6.11.: Die Funktion `untergebene` soll die Anzahl der direkten und indirekten Untergebenen des Mitarbeiters mit der als Parameter übergebenen Personalnummer liefern. Der Code dieser Funktion findet sich in Abb. 76.

□

Zur weiteren Vertiefung des Umgangs mit hierarchischen Strukturen betrachten wir ein Problem aus der Fertigungsindustrie. Eine Stückliste für Baugruppen sei als Relationenschema

```
1 CREATE OR REPLACE FUNCTION untergebene (INTEGER)
2 RETURNS INTEGER AS '
3 DECLARE
4     n ALIAS FOR $1;
5     s INTEGER;
6     r INTEGER;
7     p RECORD;
8 BEGIN
9     s := 0;
10    FOR p IN
11        SELECT pid FROM personal
12        WHERE vorges_id = n
13    LOOP
14        SELECT untergebene (p.pid) INTO r;
15        s := s + r + 1;
16    END LOOP;
17    RETURN s;
18 END;
19 ' LANGUAGE plpgsql;
```

Abb. 76: Bestimmung der Zahl der Untergebenen eines Mitarbeiters

TEILE(OT,UT,N)
gegeben, wobei ein Datensatz jeweils angibt, welches Bauteil OT aus wie vielen (N) Bauteilen UT besteht. In Abb. 77 ist als Beispiel die Stückliste für die Baugruppe “Steckdosenleiste” gegeben, die den – stark vereinfachten – Aufbau einer handelsüblichen 5-fach-Steckdosenleiste beschreibt.

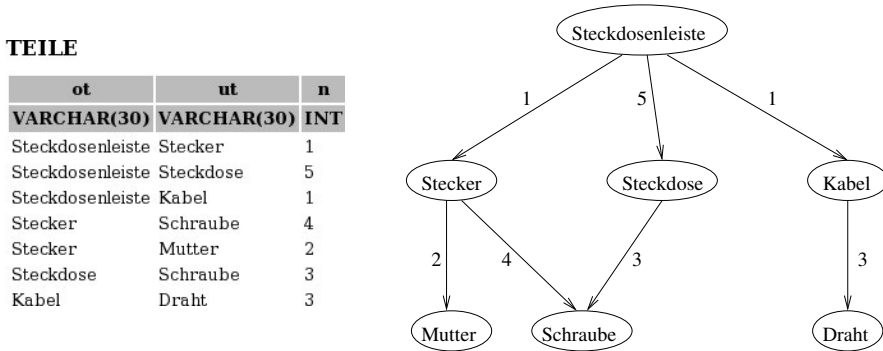


Abb. 77: Eine Stückliste als Relation und als hierarchische Struktur

Für die Bedarfsplanung des Fertigungsbetriebes muss nun die Anzahl und Art der Komponenten bestimmt werden, aus denen die zu fertigenden Baugruppen bestehen.

Beispiel 6.12.: Wir definieren eine Funktion `numcomp`, die für einen gegebenen Baugruppennamen `grp` und einen gegebenen Komponentennamen `comp` berechnet, aus wie vielen solcher Komponenten die Baugruppe besteht (Abb. 78).

```
1  CREATE OR REPLACE FUNCTION numcomp (TEXT,TEXT)
2  RETURNS INTEGER AS '
3  DECLARE
4      grp ALIAS FOR $1;
5      comp ALIAS FOR $2;
6      res INTEGER;
7      r RECORD;
8      subtotal INTEGER;
9  BEGIN
10     IF grp = comp THEN RETURN 1;
11     ELSE
12         res := 0;
13         FOR r IN
14             SELECT ot, ut, n FROM teile
15             WHERE ot = grp
16         LOOP
17             SELECT numcomp(r.ut, comp) INTO subtotal;
18             res := res + subtotal * r.n;
19         END LOOP;
20         RETURN res;
21     END IF;
22 END;
23 ' LANGUAGE plpgsql;
```

Abb. 78: Rekursive Funktion zur Teilebedarfsbestimmung

Die Funktion ist wieder rekursiv und arbeitet folgendermaßen: Stimmt `grp` mit `comp` überein, terminiert die Funktion sofort mit Rückgabewert 1 (Zeile 10), da ja eine Baugruppe trivialerweise genau einmal aus sich selbst besteht.

Im anderen Fall (**grp** und **comp** verschieden), werden mittels einer Query-FOR-Schleife die Namen der direkten Bestandteile von **grp** ermittelt (Zeile 13-16). Für jeden direkten Bestandteil **r** wird **numcomp** wieder aufgerufen, um die Anzahl **subtotal** der dort benötigten Komponenten **comp** zu ermitteln (Zeile 17); diese Zahl wird jeweils mit der Anzahl der in **grp** benötigten direkten Bestandteile **r** multipliziert (Zeile 18). Die Summe dieser Einzelwerte ergibt dann den gesuchten Wert.

Die rekursive Abarbeitung der hierarchischen Struktur ist in Abb. 79 für den Aufruf

`numcomp('Steckdosenleiste', 'Schraube')`

graphisch dargestellt.

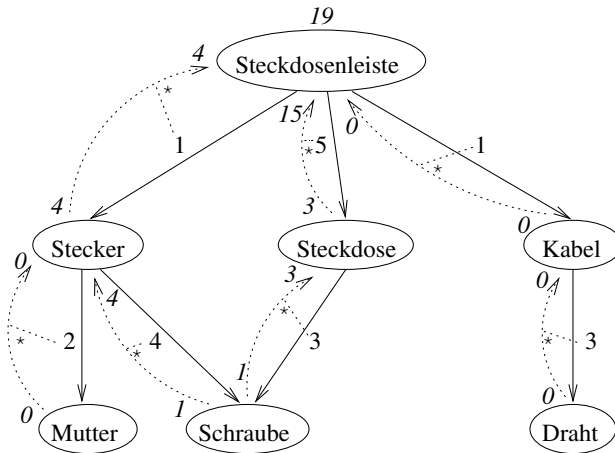


Abb. 79: Abarbeitung der hierarchischen Struktur mit **numcomp**

Beispiel 6.13.: Die Generierung der gewünschten Stückliste der etwa für die Steckdosenleiste benötigten Teile kann (wenn auch nicht besonders effizient) durch

`SELECT ut, numcomp('Steckdosenleiste', ut) FROM teile;`
erfolgen. Wir bemerken hier allerdings in der Ausgabe mehrfach vorhandene Datensätze, da ja in der Beispieldatenbank etwa das Teil 'Schraube' zweimal als Unterteil vorkommt.

Damit tatsächlich für jedes verschiedene Unterteil nur einmal die benötigte Anzahl berechnet und ausgegeben wird, erzeugen wir zunächst mit

CREATE VIEW utd AS SELECT DISTINCT ut FROM teile;
 eine View, die eine Liste von verschiedenen Unterteilen liefert. Formulieren wir dann

SELECT ut, numcomp('Steckdosenleiste', ut) FROM utd;
 so erhalten wir die Stückliste ohne Duplikate:

ut	numcomp
Draht	3
Kabel	1
Mutter	2
Schraube	19
Steckdose	5
Stecker	1



6.4. Trigger

(Datenbank-)Trigger sind mit bestimmten Tabellen verknüpfte PL/pgSQL-Programme, die immer dann ausgeführt („abgefeuert“) werden, wenn die Tabelle durch INSERT-, UPDATE- oder DELETE-Anweisungen geändert wird.

Häufig reichen die Mechanismen, die Constraints zur Überprüfung der Zulässigkeit einer Tabellenänderung anbieten, nicht aus. U. a. in solchen Fällen werden Trigger verwendet.

Beispiel 6.14.: Es soll sichergestellt werden, dass jeder Mitarbeiter höchstens zwei Projekten zugeordnet wird.

Die Überprüfung dieser Zusatzbedingung ist mit Constraints nicht möglich. Ein Trigger, der vor dem Einfügen eines Datensatzes in die Tabelle **zuordnung** ausgeführt wird, löst das Problem. Der Trigger müsste dann folgende Aktionen ausführen:

- Ermittle die Anzahl der Einträge, die für die neu einzufügende Personalnummer bereits in **zuordnung** vorhanden sind.
- Ist die Anzahl ≥ 2 (d. h. es existieren schon zwei oder mehr Zuordnungen vor dem Einfügen), melde einen Fehler und brich die Änderung ab.



Trigger werden mit CREATE TRIGGER (Abb. 80) erzeugt. Das Konzept eines Triggers ist SQL-Standard. PostgreSQL implementiert einen Teil des Standards. Anstelle der im Standard vorgesehenen Folge von

SQL-Anweisungen, die einen Trigger definieren, muss in PostgreSQL ein Trigger stets über eine benutzerdefinierte Funktion spezifiziert werden.

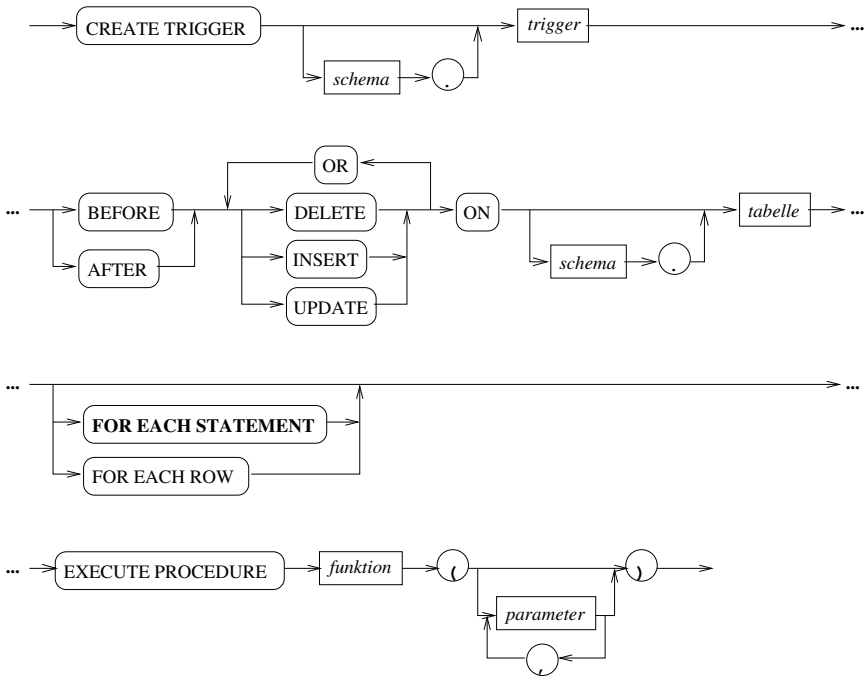


Abb. 80: Syntax des CREATE TRIGGER-Statements

Hierbei bedeutet:

<i>trigger</i>	Name des Triggers.
BEFORE	Der Trigger wird ausgeführt, bevor die Änderungsanweisung ausgeführt wird.
AFTER	Der Trigger wird ausgeführt, nachdem die Änderungsanweisung ausgeführt wurde.
DELETE	Der Trigger wird ausgeführt, wenn die Änderungsanweisung eine DELETE-Anweisung ist.
INSERT	Der Trigger wird ausgeführt, wenn die Änderungsanweisung eine INSERT-Anweisung ist.

UPDATE	Der Trigger wird ausgeführt, wenn die Änderungsanweisung eine UPDATE-Anweisung ist.
ON <i>tabelle</i>	Gibt die Tabelle an, mit der der Trigger assoziiert werden soll.
FOR EACH STATEMENT	Definition eines Statement-Triggers. Dieser wird für die gesamte Änderungsanweisung stets einmal ausgeführt, unabhängig davon, wie viele Datensätze geändert werden (auch wenn konkret keine Datensätze geändert werden). Standardmäßig werden Statement-Triggers definiert.
FOR EACH ROW	Definition eines Zeilentriggers. Dieser wird für <i>jeden</i> geänderten Datensatz genau einmal ausgeführt.
<i>funktion</i>	Name einer in PL/pgSQL definierten Triggerfunktion. Diese muss einen Wert vom speziellen (Pseudo-) Datentyp TRIGGER zurückliefern.
<i>parameter</i>	Der Funktion können optional Parameter übergeben werden, die als Stringlitterale anzugeben sind. In diesem Buch wird auf diese Möglichkeit nicht weiter eingegangen.

Grundsätzlich sollte jede Triggerfunktion mit

```
RETURN new;
```

beendet werden. Bei BEFORE-Zeilentriggern lässt sich über diesen Mechanismus der durch das auslösende INSERT- oder UPDATE-Statement eingefügte bzw. geänderte Datensatz modifizieren, da letztlich der mit RETURN zurückgelieferte Datensatz in die Tabelle übernommen wird.

Ein Sonderfall ist

```
RETURN NULL;
```

bei BEFORE-Zeilentriggern. Hierdurch wird die vom auslösenden Statement für diese Zeile vorzunehmende Operation nicht ausgeführt, ohne dass jedoch eine Ausnahmebedingung (und damit der Abbruch der Transaktion) veranlasst wird.

Beispiel 6.15.: Der oben vorgestellte Trigger, der eine Zuordnung eines Mitarbeiters zu höchstens 2 Projekten sicherstellen soll, wird wie in Abb. 81 gezeigt definiert.

```

1  CREATE OR REPLACE FUNCTION f_maxproj ()
2  RETURNS trigger AS '
3
4  DECLARE
5      nump INTEGER;
6
7  BEGIN
8
9      SELECT COUNT(projid) INTO nump FROM zuordnung
10     WHERE pid = new.pid;
11
12     IF nump >= 2 THEN
13         RAISE EXCEPTION
14         ''Zu viele Projekte fuer pid %'', new.pid;
15     END IF;
16
17     RETURN new;
18
19 END;
20 ' LANGUAGE plpgsql;
21
22 CREATE TRIGGER t_maxproj
23 BEFORE INSERT OR UPDATE
24 ON zuordnung
25 FOR EACH ROW
26 EXECUTE PROCEDURE f_maxproj ();

```

Abb. 81: Definition eines Triggers zur Zuordnungsbeschränkung

In den Zeilen 1–14 wird zunächst die Funktion definiert, die der Trigger aufrufen soll, während in den Zeilen 15–19 die eigentliche Triggerdefinition erfolgt.



Bei Zeilentriggern kann man auf den alten und den geänderten Datensatzinhalt über zwei RECORDs zugreifen, deren implizite Definition folgender expliziter Definition entspricht:

old *tabelle*%ROWTYPE

enthält den alten Datensatz (nicht sinnvoll bei INSERT-Triggern) und

new *tabelle*%ROWTYPE

enthält den geänderten (bzw. bei INSERT-Triggern den neuen) Datensatz (nicht sinnvoll bei DELETE-Triggern). (Zur Semantik des %ROWTYPE-Konstrukts siehe Abschnitt 6.3.1.

Beispiel 6.16.: Wird nach der Definition des Triggers `t_maxproj` versucht, dem Mitarbeiter 411 das (dritte) Projekt 8 zuzuordnen, etwa durch

```
INSERT INTO zuordnung VALUES (411,8);
```

führt dies erwartungsgemäß zu der Fehlermeldung

Zu viele Projekte fuer pid 411
und die Einfuegung wird nicht vorgenommen.

Zu beachten ist, dass Trigger (im Unterschied zu Constraints) keine Überprüfung bereits vorhandener Tabelleninhalte vornehmen, sondern nur auf Anweisungen wirken, die nach ihrer Definition gegeben werden. So wäre es im Beispiel kein Fehler, wenn bei Definition von `t_maxproj` ein Mitarbeiter schon 3 Projekten zugeordnet wäre.

Außer der Zulassungsüberprüfung von Änderungsoperationen haben Trigger noch weitere Anwendungsgebiete:

- Automatische Erzeugung von weiteren (abgeleiteten) Attributwerten. Z. B. kann durch einen Trigger erreicht werden, dass bei manueller Änderung des Familienstandes eines Mitarbeiters in einer Personaltabelle auch das Steuerklassen-Attribut geändert wird.
- Implementierung von komplexen Sicherheitsmechanismen. Beispiel: Die Personaltabelle soll nur zu bestimmten Zeiten geändert werden dürfen.
- Transparente Protokollierung von Änderungen, z. B. soll in eine separate Tabelle eingetragen werden, wer wann Änderungen an der Personaltabelle vorgenommen hat.

Beispiel 6.17.: In der Tabelle

```
AENDERUNGEN (zeit TIMESTAMP, username TEXT,
              operation TEXT, pid INTEGER)
```

soll bei jeder Änderung von **personal** Datum/Zeit und die Benutzerkennung desjenigen eingetragen werden, der die Änderung vorgenommen hat. Weiterhin soll erfasst werden, welche Änderung an welchem Datensatz (gegeben durch die eindeutige Personalnummer) vorgenommen wurde. Dieser Trigger ist in Abb. 82 aufgelistet.

Hierbei ist `TG_OP` eine spezielle nur im Kontext eines Triggers vorhandene Systemvariable, die angibt, welche Operation ('INSERT', 'UPDATE' oder 'DELETE') die Ausführung des Triggers ausgelöst hat.

Nun führen wir die folgenden SQL-Statements aus:

- Einfügen eines neuen Mitarbeiters (siehe Beispiel auf S. 61):

```
INSERT INTO personal (pid,nachname,vorname,  
                    strasse,ort)  
VALUES(427, 'Müller', 'Michael', 'Hauptstr. 5',  
      'München');
```
- Gehaltserhöhung um 2% für Mitarbeiter, die vor dem 01.01.1991 eingestellt wurden:

```

1  CREATE OR REPLACE FUNCTION f_personal_protokoll ()
2  RETURNS trigger AS '
3
4  BEGIN
5      IF TG_OP = 'INSERT' THEN
6          INSERT INTO aenderungen
7              values (CURRENT_TIMESTAMP, CURRENT_USER,
8                     TG_OP, new.pid);
9      ELSE /* TG_OP = UPDATE OR TG_OP = DELETE */
10         INSERT INTO aenderungen
11             values (CURRENT_TIMESTAMP, CURRENT_USER,
12                    TG_OP, old.pid);
13     END IF;
14     RETURN new;
15 ' LANGUAGE plpgsql;
16
17 CREATE TRIGGER t_personal_protokoll
18 AFTER INSERT OR UPDATE OR DELETE
19 ON personal
20 FOR EACH ROW
21 EXECUTE PROCEDURE f_personal_protokoll ();

```

Abb. 82: Trigger zur Erstellung eines Änderungsprotokolls

```

UPDATE personal SET gehalt=gehalt*1.02
WHERE einstellung < '19910101';

```

– Löschen des Mitarbeiters mit Personalnummer 128:

```
DELETE from personal WHERE pid=128;
```

Die Tabelle AENDERUNGEN hat dann den in Abb. 83 gezeigten Inhalt.

AENDERUNGEN

zeit	username	operation	pid
TIMESTAMP	TEXT	TEXT	INT
26.05.2004 21:27:52.823146	rank	INSERT	427
26.05.2004 21:28:12.465115	rank	UPDATE	107
26.05.2004 21:28:12.465115	rank	UPDATE	57
26.05.2004 21:29:12.477056	rank	DELETE	128

Abb. 83: Vom Trigger geschriebene Protokolldatensätze



Ein Beispiel, wie mit Triggern bei Änderungen an der Datenbank automatisch die Konsistenz wiederhergestellt werden kann, wird im folgenden angegeben.

Beispiel 6.18.: Ändert sich die Nummer eines Projektes in der Tabelle PROJEKT, müssen – damit die Datenbank-Konsistenz erhalten bleibt – auch die entsprechenden Projektnummern in der Tabelle **zuordnung** geändert werden. Mit Hilfe des Triggers in Abb. 84 geschieht dies automatisch.

```
1  CREATE OR REPLACE FUNCTION f_new_proj ()
2  RETURNS trigger AS '
3  BEGIN
4      UPDATE zuordnung
5      SET projid = new.projid
6      WHERE projid = old.projid;
7      RETURN new;
8  END;
9  ' LANGUAGE plpgsql;

10 CREATE TRIGGER t_new_proj
11 AFTER UPDATE
12 ON projekt
13 FOR EACH ROW
14 EXECUTE PROCEDURE f_new_proj ();
```

Abb. 84: Konsistenzsicherung mit einem Trigger

Wenn in der Datenbank kein FOREIGN KEY-Constraint definiert ist, der auf die PROJEKT-Tabelle verweist, funktioniert der Trigger so wie erwartet. Wurde allerdings wie in Abschnitt 4.4.3 vorgeschlagen ein Constraint für folgenden Zusammenhang

$$\{\text{ZUORDNUNG.Projid}\} \leftarrow \{\text{PROJEKT.Projid}\}$$

definiert, so kommt es z. B. bei Ausführung von

```
UPDATE projekt SET projid = 108 WHERE projid=8;
```

zu folgender Fehlermeldung:

```
Key (projid)=(8) is still referenced from table "zuordnung".
```

Dies ist nicht verwunderlich, da ja das UPDATE-Statement erst die PROJEKT-Tabelle ändert, bevor der Trigger die entsprechende Änderung in der Tabelle ZUORDNUNG vornehmen kann. Damit der Trigger trotzdem funktioniert, muss bei der Definition des Constraints

dafür gesorgt werden, dass er erst am Ende einer Transaktion überprüft wird:

```
ALTER TABLE zuordnung ADD FOREIGN KEY (projid)
REFERENCES projekt (projid)
DEFERRABLE INITIALLY DEFERRED;
```

□

Die Ausführung eines Triggers kann die Ausführung anderer Trigger nach sich ziehen. Dies ist etwa dann der Fall, wenn der Trigger eine Tabelle ändert, auf der ebenfalls ein Trigger definiert ist.

Bei der Definition von Triggern, die Änderungen an Tabellen vornehmen, ist im Zusammenhang mit der Datenbanksicherheit Vorsicht geboten, wie folgendes Beispiel zeigt: Ein Benutzer A hat das INSERT-Privileg auf einer Tabelle T1, aber keine Zugriffsrechte auf die Tabelle T2. Auf T1 sei ein INSERT-Trigger definiert, der T2 ändert. Führt A eine INSERT-Anweisung auf T1 aus, wird auch T2 durch den Trigger geändert, obwohl A keine Privilegien zum direkten Ändern von T2 besitzt. Dieses Verhalten sollte in einer Mehrbenutzerumgebung unbedingt berücksichtigt werden.

Trigger können mit

```
DROP TRIGGER trigger ON tabelle;
```

gelöscht werden. Die ON-Klausel ist dabei nicht SQL-standardkonform, da im SQL-Standard Triggernamen global eindeutig sein müssen, während es in PostgreSQL durchaus zwei verschiedene Trigger gleichen Namens geben kann, die sich auf unterschiedliche Tabellen beziehen.

7. Datenbankzugriff in höheren Programmiersprachen

Es gibt Datenbank Anwendungen, die sich nicht alleine mit den Datenbank-Sprachen SQL oder PL/pgSQL lösen lassen, sondern außer komfortablem Zugriff auf eine Datenbank auch die Möglichkeiten einer prozeduralen Programmiersprache (z. B. PASCAL, C, Perl) benötigen. Wir werden hier zwei verschiedene Ansätze kennenlernen, SQL-Anweisungen in einer prozeduralen Programmiersprache – der sog. Host-Programmiersprache – zu verwenden.

7.1. Embedded SQL

Man kann, soweit die Datenbankimplementierung dies unterstützt, in gewöhnliche Programme einer höheren Programmiersprache¹¹ SQL-Konstrukte einstreuen, deren Ausführung dann das Datenbanksystem übernimmt. Diese Art der Verwendung von SQL in einer gewöhnlichen Programmiersprache heißt **Embedded SQL** (kurz ESQL). Wir gehen hier nur auf ESQL-Konstrukte in Verbindung mit der Programmiersprache C ein. Die Ausführungen in diesem Abschnitt sollen nur einen Überblick über ESQL geben. Kenntnisse in der Programmiersprache C setzen wir in diesem Abschnitt voraus.

Embedded SQL ist im SQL-Standard definiert, die PostgreSQL-Implementierung **ECPG**, die wir hier beschreiben, ist zum größten Teil standardkonform.

Ein C-Programm mit Embedded SQL-Konstrukten (das sog. **Host-Programm**) muss zunächst durch einen speziellen Precompiler behandelt werden. Dieser ersetzt die ECPG-Konstrukte durch Aufrufe von Bibliotheksfunktionen, die von dem Datenbanksystem zur Verfügung gestellt werden. Das precompilierte Programm kann dann mit dem üblichen Compiler in ausführbare Form übersetzt werden.

7.1.1. Host- und Indikatorvariablen

Die Kommunikation zwischen der C-Programmumgebung und den SQL-Statements erfolgt über sogenannte **Hostvariablen**. Diese können gemeinsam von C und SQL benutzt werden.

¹¹ Welche dieser sog. **Host-Programmiersprachen** verwendet werden können, hängt vom DBMS ab.

Zulässige Datentypen für Hostvariablen sind die C-Datentypen `char`, `char[n]`, `int`, `short`, `long`, `float`, `double` sowie der Pseudo-Datentyp `VARCHAR[n]`. Letzterer beschreibt eine Zeichenkette mit Länge $\leq n$ und entspricht in C einer Struktur mit den Komponenten

```
unsigned short len;
unsigned char arr[n];
```

wobei `len` jeweils die tatsächliche Länge der in `arr` gespeicherten Zeichenkette angibt. Die Zeichenkette in `arr` muss nicht durch ein Nullbyte terminiert sein (wenn man auf `arr` mit den C-Stringfunktionen arbeiten will, muss man die Zeichenkette allerdings selbst durch ein Nullbyte terminieren).

Die Datentypen der Hostvariablen werden bei der Benutzung in einem SQL-Statement auf SQL-interne Datentypen abgebildet. Die Typkonversion läuft automatisch nach Regeln ähnlich den in Abschnitt 3.3.7 beschriebenen ab. Auf diese Weise können Werte relativ problemlos zwischen C und SQL ausgetauscht werden.

Das einzige Problem stellt die Tatsache dar, dass es in SQL den speziellen Wert NULL gibt, der in C keine Entsprechung hat. Um nun trotzdem auch NULL-Werte austauschen zu können, gibt es **Indikatorvariablen**. Dabei handelt es sich um Variablen vom Typ `short`, die mit einer Hostvariable assoziiert sind; ihr Wert bestimmt jeweils, ob der Wert der Hostvariablen von SQL ignoriert und als NULL-Wert aufgefasst werden soll.

Für Eingabe-Hostvariablen (mit denen Werte an SQL übergeben werden sollen) werden die Werte der korrespondierenden Indikatorvariablen folgendermaßen interpretiert:

- < 0 Der tatsächliche Wert der Hostvariable wird ignoriert, er wird als NULL angesehen.
- ≥ 0 Der Wert der Hostvariablen wird verwendet.

Für Ausgabe-Hostvariablen (mit denen Werte von SQL an C zurückgeliefert werden) lautet die Interpretation der Werte der korrespondierenden Indikatorvariablen wie folgt:

- < 0 Es wurde ein NULL-Wert zurückgeliefert.
- 0 Die Hostvariable enthält einen gültigen Wert.
- > 0 Der Wert wurde abgeschnitten, da die Hostvariable nicht lang genug war, um ihn ganz aufnehmen zu können. Der Wert der Indikatorvariablen gibt in diesem Fall die benötigte Länge an.

Host- und Indikatorvariablen werden im C-Programm wie gewöhnliche Variablen definiert, die Definition muss jedoch eingeleitet werden mit

```
EXEC SQL BEGIN DECLARE SECTION;
```

und beendet werden mit

```
EXEC SQL END DECLARE SECTION;
```

Host- und Indikatorvariablen werden in C-Programmen wie gewöhnliche C-Variable verwendet. In SQL muss dem Namen der Hostvariablen ein Doppelpunkt vorangestellt werden; ist die Hostvariable mit einer Indikatorvariablen assoziiert, muss das Konstrukt

:hostvariable :indikatorvariable

benutzt werden.

7.1.2. Embedded SQL-Statements

Alle ECPG-Konstrukte werden durch die einleitenden Schlüsselworte `EXEC SQL` als solche kenntlich gemacht. Das in Abb. 85 gezeigte Syntaxdiagramm stellt die wichtigsten ECPG-Statements dar, die im folgenden – soweit noch nicht bekannt – etwas genauer beschrieben werden.

An-/Abmelden und Transaktionskontrolle

Bevor man ECPG-Statements ausführen kann, muss man sich zunächst am Datenbanksystem anmelden. Dies geschieht in ECPG mit dem `CONNECT`-Statement, das als Parameter eine Hostvariable mit dem sog. Connectstring und dann eine Hostvariable mit dem Usernamen sowie optional eine weitere Hostvariable mit dem zugehörigen Passwort erhält. Der letzte Parameter kann weggelassen werden, wenn kein Passwort erforderlich ist.

Das `EXEC SQL CONNECT`-Statement ist DBMS-spezifisch und hier für ECPG angegeben. Der Connectstring für eine Verbindung zu einer PostgreSQL-Datenbank sieht wie folgt aus:

- Wenn die Verbindung zu einem Datenbank-Server erfolgen soll, der auf dem gleichen Rechner läuft wie die ECPG-Applikation, die die Verbindung herstellt:

`unix:postgres://localhost/datenbankname`

- Wenn die Verbindung zu einem anderen Rechner erfolgen soll:

`tcp:postgres://hostname/datenbankname`

Die Beendigung einer Transaktion erfolgt entweder mit einem `COMMIT`- oder einem `ROLLBACK`-Statement (siehe Abschnitt 5.1). Man beachte, dass in ECPG auch ohne das Vorliegen eines Transaktionskontextes eine Übernahme der durch die ausgeführten Kommandos vorgenommenen Änderungen in die Datenbank nur dann erfolgt,

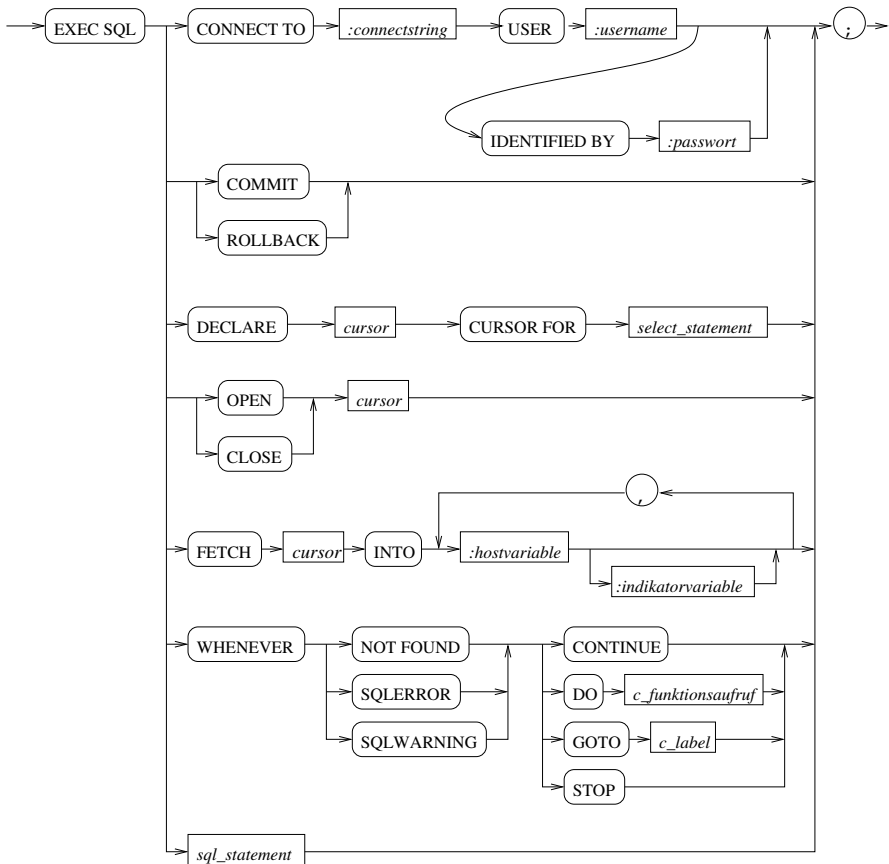


Abb. 85: Syntax von Embedded SQL-Statements

wenn vor der Abmeldung mit EXEC SQL DISCONNECT bzw. dem Programmende ein EXEC SQL COMMIT stattfindet.

Fehlerbehandlung

Für die Behandlung von Fehlern, die bei der Ausführung von ECPG-Konstrukten auftreten, gibt es verschiedene Methoden. Wir stellen hier nur die einfachste Methode – Verwendung des WHENEVER-Statements – dar.

Mit einem WHENEVER-Statement kann man für folgende Fälle spezielle Aktionen definieren:

- Bei einem SELECT...INTO- oder FETCH-Statement wurde kein Datensatz (mehr) gefunden ("NOT FOUND").

- Bei der Ausführung eines ECPG-Statements trat ein Fehler auf (“SQLERROR”). Dieser Fall ist nicht im SQL-Standard enthalten.
- Bei der Ausführung eines ECPG-Statements wurde eine Warnung gemeldet (“SQLWARNING”).

Für jeden dieser Fälle kann eine der folgenden Aktionen festgelegt werden:

- Ignorieren des Fehlers. Fortfahren mit dem nächsten Statement des Programms (“CONTINUE”). Dies ist das Standardverhalten, wenn keine andere Aktion spezifiziert wurde.
- Aufruf einer C-Funktion (“DO”). Nach der Ausführung der Funktion wird das Programm mit dem nächsten Statement fortgesetzt. Die angegebene Funktion muss parameterlos sein. Nicht im SQL-Standard.
- Sprung zu einem Label im C-Programm (“GOTO”). Die Programmausführung wird bei diesem Label fortgesetzt.
- Programmabbruch (“STOP”). Es erfolgt zusätzlich ein impliziter ROLLBACK und Abmeldung von der Datenbank. Nicht im SQL-Standard.

Man beachte, dass aufgrund der Implementierung des Precompilers für ein ECPG-Statement im Fehlerfall jeweils die Aktion durchgeführt wird, die durch das physikalisch (nicht logisch) vorhergehende WHENEVER-Statement definiert ist.

7.1.3. Ein Beispiel

Beispiel 7.1.: Wir zeigen hier als motivierendes Beispiel ein C-Programm mit Embedded SQL-Statements, das die Daten eines neuen Mitarbeiters erfragt und den Datensatz in die Datenbank der Beispiel-firma einträgt. Der Quellcode des Programms ist in Abb. 86 zu sehen.

In den Zeilen 7–11 werden die Hostvariablen definiert. Für die Verwendung von Zeichenketten bietet Embedded SQL den Spezialdatentyp VARCHAR, der in C als Struktur, die die Länge und die aktuelle Zeichenkette enthält, realisiert ist.

Die Zeilen 13–20 definieren die Funktion `sqlerror`, die bei Auftreten eines SQL-Fehlers ausgeführt werden soll. Die Ausführung dieser Funktion im Fehlerfall wird in Zeile 23 festgelegt. Das Auslesen des Fehler-textes in Zeile 17 über die Variable `sqlca` ist nicht im SQL-Standard definiert, wird aber auch in anderen DBMS so implementiert.

```

1  #include <stdio.h>
2  #include <string.h>
3  #define READVARIABLE(prompt, str) printf(prompt); gets(str.arr); \
4                                str.len = strlen(str.arr);
5  #define SKIPLINE while ((c = getchar()), c != '\n');
6  EXEC SQL BEGIN DECLARE SECTION;
7      VARCHAR nachname[21]; VARCHAR vorname[16]; VARCHAR strasse[31];
8      VARCHAR ort[21]; VARCHAR abteilung[21];
9      float gehalt; int pers_id, abt_id;
10     char *connectstring = "unix:postgresql://localhost/firma";
11     char *user = "rank";
12 EXEC SQL END DECLARE SECTION;
13 void sqlerror ()
14 {
15     EXEC SQL WHENEVER SQLERROR CONTINUE;
16     printf ("\nDatenbank-Fehler:\n");
17     printf ("%70s\n", sqlca.sqlerrm.sqlerrmc);
18     EXEC SQL DISCONNECT;
19     exit (1);
20 }
21 int main ()
22 { int c;
23     EXEC SQL WHENEVER SQLERROR DO sqlerror();
24     EXEC SQL CONNECT TO :connectstring USER :user;
25     printf ("Eintrag eines neuen Mitarbeiters:\n\n");
26     READVARIABLE ("Nachname: ", nachname); READVARIABLE ("Vorname: ", vorname);
27     READVARIABLE ("Strasse: ", strasse); READVARIABLE ("Wohnort: ", ort);
28     printf ("Gehalt: "); scanf ("%f", &gehalt); SKIPLINE;
29     EXEC SQL WHENEVER NOT FOUND GOTO notfound;
30     do {
31         READVARIABLE ("Abteilung: ", abteilung);
32         EXEC SQL SELECT aid INTO :abt_id FROM abteilung
33             WHERE bezeichnung = :abteilung;
34         break;
35     notfound:
36         printf ("Abteilung existiert nicht - Eingabe wiederholen!\n");
37     } while (1);
38     printf ("\nAbteilungs-Id = %d\n", abt_id);
39     EXEC SQL SELECT MAX(pid) INTO :pers_id FROM personal;
40     pers_id++;
41     printf ("Neue Personalnummer = %d\n", pers_id);
42     EXEC SQL INSERT INTO personal
43         (pid, nachname, vorname, strasse, ort, aid, gehalt, einstellung)
44         VALUES (:pers_id, :nachname, :vorname, :strasse, :ort,
45             :abt_id, :gehalt, CURRENT_DATE);
46     printf ("Einfuegen erfolgt.\n");
47     EXEC SQL COMMIT;
48     EXEC SQL DISCONNECT;
49     exit (0);
50 }

```

Abb. 86: Ein C-Programm mit ECPG-Konstrukten

Zeile 24 stellt die Verbindung zur Datenbank her. Hier ist im allgemeinen Fall in der Hostvariablen `:connectstring` ein wie oben beschriebener Connectstring anzugeben.¹² In der Hostvariablen `:user` geben wir den (DBMS-) Benutzernamen an, mit dem die Anmeldung an der Datenbank erfolgen soll.

In den Zeilen 26–28 werden die Mitarbeiterdaten eingelesen.

Zeile 29 legt fest, dass bei Auftreten einer NOT FOUND-Ausnahmebedingung die Programmausführung an der mit `notfound` bezeichneten Stelle des Programms fortgesetzt werden soll.

Die Zeilen 31–33 lesen einen Abteilungsnamen ein und holen mit einem Embedded SQL-Statement die zugehörige Abteilungsnummer aus der Tabelle ABTEILUNG. Bei Eingabe einer nichtexistenten Abteilung kommt es zur Ausnahmebedingung NOT FOUND, die hier zur Ausgabe einer Fehlermeldung (Zeile 36) und erneuter Anforderung eines Abteilungsnamens führt.

In den Zeilen 39–41 wird die höchste bereits vergebene Personalnummer ermittelt, um 1 erhöht und als Personalnummer des neuen Mitarbeiters verwendet.

Der Eintrag für den neuen Mitarbeiter wird in den Zeilen 42–45 erzeugt.

Die Zeile 47 macht die vorgenommenen Änderungen permanent (COMMIT), die folgende Zeile baut die Verbindung zur Datenbank wieder ab.



7.2. Die DBI-Schnittstelle für Perl

Wir beschreiben nun eine Methode, wie man aus Programmen in der Sprache Perl SQL-Anweisungen absetzen kann.

7.2.1. Eigenschaften von Perl und DBI

Perl (“Practical Extraction and Report Language”) ist eine in den 80er-Jahren im Unix-Umfeld entstandene Programmiersprache, mit

¹² Wir gehen hier – und bei allen weiteren Beispielen, bei denen eine Verbindung zur Datenbank hergestellt wird – davon aus, dass die Datenbank “firma” heißt, sich auf dem gleichen Rechner wie die Applikation befindet (“localhost”) und die Anmeldung an die Datenbank als Benutzer “rank” lokal ohne Passwort erfolgen kann.

der man auf besonders einfache Weise jegliche Art von textuellen Daten analysieren und bearbeiten kann. Darüber hinaus handelt es sich bei Perl um eine vollwertige und universelle höhere Programmiersprache, deren Popularität nicht nur bei der Anbindung von Datenbanken, sondern auch bei der Erzeugung von dynamischen WWW-Seiten – wir werden darauf in Abschnitt 8 näher eingehen – immer mehr steigt.

Perl ist frei verfügbar (es handelt sich auch hier um sog. Open Source-Software, bei der der Quellcode zur Verfügung steht¹³) und läuft auf allen Unix-ähnlichen Betriebssystemen. Implementierungen für zahlreiche andere Betriebssystemplattformen sind ebenfalls kostenlos verfügbar. Deshalb können Perl-Programme meist ohne jeglichen Portierungsaufwand auf fast allen gängigen Hardware- und Betriebssystemplattformen eingesetzt werden. Für die Ausführung von Perl-Programmen wird kein expliziter Compilierungsschritt benötigt – die Programme werden interpretiert.¹⁴ Der Programmablauf ist trotzdem recht effizient.

Für die folgenden Ausführungen setzen wir voraus, dass der Leser mit den Grundlagen von Perl vertraut ist – eine Einführung findet sich beispielsweise in [ScCh2001]. Die Beispielprogramme dürften jedoch mit den zusätzlichen Erläuterungen auch Lesern mit Kenntnissen in einer anderen höheren Programmiersprache grundsätzlich verständlich sein.

DBI (“database interface”) ist ein Perl-Modul, das sich in den letzten Jahren zu einem de-facto-Standard für den Zugriff auf Datenbanken von Perl aus entwickelt hat. DBI stellt eine datenbankunabhängige (d. h. insbesondere herstellerunabhängige) Schnittstelle zur Verfügung. DBI bedient sich zur Umsetzung von Datenbankzugriffen auf ein spezifisches Datenbanksystem zusätzlicher DBD- (“database driver”)-Module, die speziell auf das jeweils verwendete Datenbanksystem zugeschnitten sind. Ebenso wie Perl sind auch DBI und die diversen DBD-Module frei verfügbar.

Wir werden in diesem Abschnitt überblicksmäßig die wichtigsten Aspekte von DBI behandeln, ohne auch nur annähernd den Anspruch

¹³ Eine ausführliche Erläuterung des Begriffs “Open Source” findet sich beispielsweise in [Op1999].

¹⁴ Tatsächlich wird vor der Ausführung automatisch ein interner Zwischencode generiert. Dies geschieht jedoch transparent für den Benutzer.

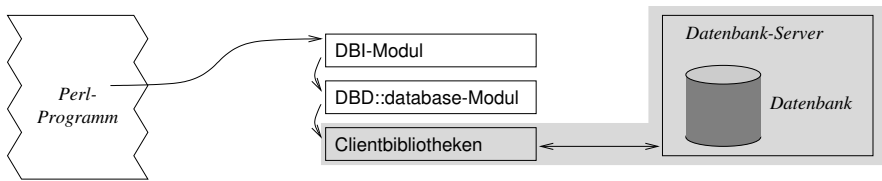


Abb. 87: Ablauf eines Datenbankzugriffs mit DBI

auf Vollständigkeit erheben zu wollen. Eine ausführliche Erläuterung aller Features von DBI findet sich in [DeBu2000].

Die Graphik in Abb. 87 verdeutlicht das Abstraktionsniveau der DBI-Schnittstelle. Hier ist zu erkennen, dass auch ein `DBD::database`-Modul¹⁵ nicht direkt auf die Datenbank zugreift, sondern Clientbibliotheken des entsprechenden Datenbanksystems benutzt.¹⁶ Das bedeutet natürlich, dass die DBMS-spezifische Clientsoftware vorhanden sein muss.

7.2.2. Datenbankzugriffe mit DBI

Das in Abb. 88 gezeigte Programm illustriert die grundlegende Vorgehensweise bei Ausführung von Datenbankzugriffen mit DBI. Es zeigt gleichzeitig, wie einfach die Verwendung der DBI-Schnittstelle ist. Unser Programm soll die Namen aller Mitarbeiter auf dem Bildschirm anzeigen.

Schon aus diesem kurzen Programm wird deutlich, wie Datenbankzugriffe mit Perl unter Verwendung von DBI ablaufen. Zeile 1 gehört eigentlich nicht zum Perl-Programm, sondern dient auf Unix-Systemen als Hinweis an das Laufzeitsystem, dass das Programm durch den `/usr/bin/perl` installierten Perl-Interpreter abzuarbeiten ist.¹⁷ Zeile 2 bindet das DBI-Modul zur Verwendung im Programm ein.

¹⁵ *database* ist z. B. `Oracle`, `Informix`, `mysql` oder eben `Pg` (für PostgreSQL).

¹⁶ Es gibt auch DBD-Module für physisch besonders einfach strukturierte Datenbanken, die direkt – ohne Clientsoftware – auf die Datenbank zugreifen können, z. B. `DBD::CSV` für Datenbanken, die in Form von Textdateien mit durch Kommata getrennten Attributwerten vorliegen.

¹⁷ Die verwendete Option `-w` weist den Perl-Interpreter an, bei syntaktisch korrekten, aber möglicherweise fehlerhaften Konstrukten Warnmeldungen auszugeben. Diese sehr nützliche Option sollte man stets verwenden.


```
1  #! /usr/bin/perl -w
2  use DBI;
3  $dbh = DBI->connect ("dbi:Pg:dbname=firma", "rank");
4  $sth = $dbh->prepare
5      ("SELECT nachname, vorname FROM personal");
6  $sth->execute ();
7  while (($n,$v) = $sth->fetchrow_array()) {
8      print "$v $n\n";
9  }
10 $dbh->disconnect ();
```

Abb. 88: Ein einfacher Datenbankzugriff mit DBI

Die Anweisung in Zeile 3 stellt die Verbindung zur Datenbank her. Die drei Parameter der Methode `DBI->connect` sind

- Beschreibung der Datenbank, auf die zugegriffen werden soll,
- Benutzername und
- Passwort¹⁸ – optional, muss nur angegeben werden, wenn ein Passwort für den Verbindungsaufbau zur Datenbank erforderlich ist.

Der erste Parameter ist eine Zeichenkette, die wie folgt aufgebaut ist:

`dbi:datenbanksystem:datenbankspezifikation`

In unserem Beispielprogramm wird also eine Verbindung zu einer PostgreSQL-Datenbank mit dem Namen “firma” aufgebaut. Genauer ist *datenbanksystem* der Name des zu verwendenden DBD::*datenbanksystem*-Moduls, *datenbankspezifikation* ist von dem verwendeten Datenbanksystem abhängig. In PostgreSQL lautet die Spezifikation im einfachsten Fall

`dbname=datenbankname`

sie kann aber auch weitere Angaben, z. B. den Namen des Rechners (Host), auf dem der Datenbank-Server läuft, enthalten.

Das Resultat der Methode `DBI->connect` ist ein sogenanntes Datenbank-Handle, das für die folgenden Zugriffe auf die Datenbank benötigt wird und deshalb in der Variablen `$dbh` gespeichert wird.

In den Zeilen 4 und 5 wird eine SQL-Anweisung als normale Zeichenkette an das Datenbanksystem geschickt (“vorbereitet”). Um dies zu erreichen, wird das Datenbank-Handle verwendet – es besitzt eine Methode `prepare`, die genau die gewünschte Aktion ausführt. Das Resultat ist ein sogenanntes Statement-Handle, über das weitere Aktionen

¹⁸ siehe Fußnote zum Beispiel aus Abschnitt 7.1.3 auf S. 149.

mit der vorbereiteten SQL-Anweisung ausgeführt werden können. Wir speichern dieses Handle daher in der Variablen `$sth`. Wir benutzen das Handle dann in Zeile 6, um das Datenbanksystem anzuweisen, diese SQL-Anweisung auszuführen.

Da die ausgeführte SELECT-Anweisung eine Ergebnistabelle liefert, die aus mehreren Zeilen bestehen kann, stellt sich die Frage, wie man auf die Ergebnistabelle im Perl-Programm zugreifen kann. Analog etwa zum Cursorprinzip in PL/pgSQL werden die Datensätze der Ergebnistabelle einfach zeilenweise ausgelesen. Hierzu dient die Methode `fetchrow_array` des Statement-Handle. Diese liefert jeweils eine weitere Zeile der Ergebnistabelle als Perl-Array, solange noch Daten vorhanden sind. Andernfalls wird ein undefinierter Wert geliefert (der in Perl als logisch falsch interpretiert wird).

Deshalb können wir in Zeile 7 in der Bedingung einer `while`-Schleife jeweils den nächsten Datensatz holen und – wir wissen ja aufgrund der Form der SELECT-Abfrage, dass er aus zwei Werten bestehen muss – in die Variablen `$n` bzw. `$v` schreiben. Soweit noch ein Datensatz der Ergebnistabelle vorhanden ist, enthält dann `$n` den Nachnamen und `$v` den Vornamen eines Mitarbeiters. Ist kein weiterer Datensatz mehr vorhanden, wird aufgrund des gelieferten undefinierten Wertes die Schleife verlassen. In der Schleife, also in Zeile 8, geben wir jeweils den Vornamen und den Nachnamen des Mitarbeiters aus.

Zeile 10 bildet das Gegenstück zu Zeile 3 – hier wird die Verbindung zur Datenbank wieder abgebaut.

Die erfolgreiche Ausführung dieses Programms mit unserer Musterdatenbank FIRMA würde dann folgende Ausgabe liefern:

```
Markus Meyer  
Hilde Huber  
Steffi Schmidt  
Friedrich Frisch  
Karl Klement  
Bernhard Berger
```

7.2.3. Fehlerprüfung

Wir hatten in unserem Beispiel keine Fehlerprüfung vorgenommen. Selbstverständlich kann es beim Zugriff auf eine Datenbank zu den verschiedensten Fehlersituationen kommen, etwa weil die Verbindung mit der Datenbank fehlschlägt oder weil das Datenbanksystem eine

Anweisung nicht ausführen kann. Nehmen wir beispielsweise an, wir hätten uns bei der Angabe der SELECT-Anweisung vertippt und etwa geschrieben:

```
SELECT nachname, vorname FROM persona
```

Dies muss zu einem Fehler führen, da in unserer Musterdatenbank keine Tabelle mit dem Namen “persona” existiert. Tatsächlich bricht das derart modifizierte Programm mit einer Fehlermeldung ähnlich der folgenden ab (`dbiex.pl` ist der Name, den wir dem Programm gegeben haben):

```
DBD::Pg::st execute failed: ERROR: relation "persona"
does not exist at ./dbiex_error.pl line 7.
DBD::Pg::st fetchrow_array failed: no statement
executing at ./dbiex_error.pl line 8.
```

Wir bemerken, dass die fehlerhafte Anweisung bereits in Zeile 7 bemerkt wird, der Programmabbruch erfolgt jedoch erst in Zeile 8, nämlich dann, wenn das Programm versucht, mit dem Statement-Handle, das die Anweisung in Zeile 4–5 liefern sollte, weiterzuarbeiten. Das klappt aber nicht, da aufgrund der Fehlersituation kein gültiges Statement-Handle, sondern ein undefinierter Wert geliefert wurde, mit dem nicht mehr weitergearbeitet werden kann.

Dankenswerterweise gibt es in DBI die Möglichkeit einer automatischen Fehlerprüfung, und zwar in zwei Abstufungen:

- bei Fehlern im Datenbankzugriff wird lediglich eine Warnung ausgegeben, das Programm läuft aber weiter,
- bei Fehlern im Datenbankzugriff wird das Programm mit einer Fehlermeldung abgebrochen.

Das standardmäßige Verhalten von DBI ist die Ausgabe von Warnungen ohne Programmabbruch, wie wir das auch bei unserem Programm in Zeile 7 gesehen haben. Dass das Programm trotzdem in Zeile 8 mit einem Fehler abbricht und nicht etwa in eine Endlosschleife läuft, liegt daran, dass die `fetchrow_array`-Methode wegen des Fehlers keinen gültigen Datensatz zurückliefert, sondern einen undefinierten Wert, der aufgrund unserer Schleifenbedingung zum Abbruch der Schleife führt.

An dieser Stelle sei angemerkt, dass bei der Verwendung von DBI mit anderen Datenbanksystemen der Fehler möglicherweise an einer anderen Stelle bemerkt wird, z. B. bereits beim Aufruf der `prepare`-Methode in Zeile 4. Programme, die datenbankunabhängig sein sollen, müssen dieses Verhalten entsprechend berücksichtigen.

Man kann die automatische Fehlerprüfung von DBI aber auch ganz abschalten, um selbst die Kontrolle über die Fehlerbehandlung zu haben. Das empfiehlt sich dringend bei Produktivprogrammen, da dort auf den spezifischen Fehler geeignet reagiert werden muss. Näheres hierzu findet sich in der Dokumentation zum DBI-Modul¹⁹ oder auch in [DeBu2000].

7.2.4. Unterstützung von Transaktionen

DBI unterstützt Transaktionen in dem gleichen Umfang wie das verwendete Datenbanksystem. Standardmäßig führt DBI allerdings nach jeder ausgeführten SQL-Anweisung ein implizites COMMIT durch (sog. `AutoCommit`-Einstellung). Dies lässt sich jedoch abschalten; es stehen dann für jedes Datenbank-Handle die Methoden `commit` und `rollback` für den expliziten Abschluss einer Transaktion zur Verfügung. Eine Transaktion wird dann stets mit dem ersten Statement nach `commit` bzw. `rollback` oder mit dem ersten Statement nach Herstellung der Datenbankverbindung begonnen.²⁰

Wir werden in Abschnitt 8.9 nochmals darauf zurückkommen. Für ausführlichere Informationen zur `AutoCommit`-Funktionalität sei auf die DBI-Dokumentation bzw. auf [DeBu2000] verwiesen.

7.2.5. Werteübergabe zwischen Perl und SQL

Wie wir schon in dem Programm aus Abb. 88 gesehen hatten, geschieht die Übergabe von Werten aus SQL an Perl ohne Zuhilfenahme von Hostvariablen wie in Embedded SQL.²¹ Für die umgekehrte Richtung – also die Übergabe von Werten aus Perl an eine SQL-Anweisung – verwendet DBI das Konzept des “Bindens” von Werten an spezielle “Platzhalter” in einer SQL-Anweisung.

¹⁹ Auf Systemen, wo dieses Modul installiert ist, kann man mit dem Kommando

`perldoc DBI`

die Online-Dokumentation aufrufen.

²⁰ In PostgreSQL ist somit ein `START TRANSACTION`-Statement zum Beginnen einer Transaktion nicht notwendig.

²¹ DBI bietet zwar ein Konstrukt an, das so ähnlich wie Hostvariablen funktioniert; dieses ist jedoch nicht notwendig, um Daten zwischen Perl und SQL auszutauschen. Wir werden dieses Konstrukt daher hier auch nicht näher betrachten.

Wie das funktioniert, wollen wir an einem Programm zeigen, das in unserer Musterdatenbank den Ort einer Abteilung ändern soll. Dabei werden der Name der zu modifizierenden Abteilung sowie der neue Ort interaktiv vom Benutzer abgefragt. Das Programm ist in Abb. 89 aufgelistet.

```
1  #! /usr/bin/perl -w
2  use DBI;
3  $dbh = DBI->connect ("dbi:Pg:dbname=firma", "rank");
4  print "Abteilung? "; $abt = <STDIN>; chomp ($abt);
5  print "Neuer Ort? "; $ort = <STDIN>; chomp ($ort);
6  $sth = $dbh->prepare
7      ("UPDATE abteilung SET ort=? WHERE bezeichnung=?");
8  $rows = $sth->execute ($ort,$abt);
9  print "Geaenderte Datensaeetze: $rows\n";
10 $dbh->disconnect ();
```

Abb. 89: Übergabe von Daten an eine SQL-Anweisung mit DBI

Wir verbinden uns zunächst wieder zur Datenbank (Zeile 3) und fragen dann in den Zeilen 4 und 5 die benötigten Eingaben – Abteilungsname und neuer Ort – ab.²²

Die für die Aktualisierung der Datenbank benötigte UPDATE-Anweisung wird in den Zeilen 6 und 7 an SQL übergeben. Hier fällt auf, dass sich an den Stellen, wo die Werte stehen müssten, Fragezeichen befinden. Dabei handelt es sich um Platzhalter; die dafür einzusetzen- den Werte werden erst später bei der tatsächlichen Ausführung der Anweisung spezifiziert.

Tatsächlich erhält nun der Aufruf der `execute`-Methode in Zeile 8 zwei Parameter, nämlich die an Stelle der Platzhalter (wir hatten zwei Platzhalter verwendet) einzusetzenden Werte. Wird also z. B. bei Ablauf dieses Programms für den Abteilungsnamen “Produktion” und für den neuen Ort “Freising” eingegeben, wird tatsächlich die SQL-Anweisung

²² Für den nicht so gut mit Perl vertrauten Leser: Die `chomp`-Funktion entfernt ein am Ende einer Zeichenkette stehendes Zeilentrennzeichen. Beim Einlesen von Werten übernimmt Perl nämlich – anders als etwa C – stets den Zeilentrenner, der die Eingabe beendet.

UPDATE abteilung

SET ort='Freising' WHERE bezeichnung='Produktion';

ausgeführt. Man beachte, dass sich DBI automatisch um das sogenannte “Quoting” kümmert, d. h. es setzt ggf. Werte in die vom Datenbanksystem verlangten Anführungszeichen.

Der Rückgabewert der `execute`-Methode ist die Anzahl der durch die SQL-Anweisung geänderten Datensätze, die wir zur Kontrolle in Zeile 9 auf dem Bildschirm ausgeben.

7.2.6. Ein komplettes Programm

Wir wollen zeigen, wie das in Embedded SQL formulierte Programm von Abschnitt 7.1.3 – Abfrage der Daten eines neuen Mitarbeiters und Einfügen in die Datenbank – in Perl/DBI aussieht. Die Perl/DBI-Version ist in Abb. 90 gezeigt.

Im Vergleich zu unseren bisherigen kleineren Programmen werden keine neuen DBI-Konstrukte verwendet. Das Unterprogramm in den Zeilen 3–7 dient zur Ausgabe einer Eingabeaufforderung und Einlesen der Benutzereingabe. In der C-Version hatten wir diese Aufgabe über eine Makrodefinition gelöst.

In Zeile 8 wird die Verbindung zur Datenbank hergestellt. Wir verlassen uns auch bei diesem Programm wieder auf die Standard-Fehlerbehandlung von DBI, die bei Problemen im Datenbankzugriff eine Warnung ausgibt, ohne das Programm abzubrechen.

In den Zeilen 10–14 werden die benötigten Daten vom Benutzer abgefragt. Man beachte, dass wir hier das Gehalt als ganz normalen Perl-Wert einlesen; da Perl nicht zwischen numerischen Werten und Zeichenketten unterscheidet, könnte der Benutzer hier auch einen nicht-numerischen Wert eingeben. An dieser Stelle lassen wir dies zunächst zu.

Da der Benutzer einen Abteilungsnamen eingeben soll, wir aber für die Einfügeoperation die Abteilungs-Id benötigen, müssen wir aus dem Abteilungsnamen die Id ermitteln (Zeilen 15–24). Hier kann es natürlich passieren, dass der Benutzer einen nicht existenten Namen eingibt. Ist dies der Fall, wird der Benutzer nochmals nach einem Abteilungsnamen gefragt. Dies wird so lange iteriert, bis ein gültiger Abteilungsname eingegeben wurde.

Hier ist bemerkenswert, dass wir die benötigte SQL-Abfrage nur einmal vorbereiten müssen (in den Zeilen 15–16), obwohl wir sie

```

1  #! /usr/bin/perl -w
2  use DBI;
3  sub readvarchar ($) {
4      my ($prompt) = @_; my $str;
5      print $prompt; $str = <STDIN>; chomp($str);
6      return $str;
7  }
8  $dbh = DBI->connect ("dbi:Pg:dbname=firma", "rank");
9  print "Eintrag eines neuen Mitarbeiters:\n\n";
10 $nachname = readvarchar ("Nachname:  ");
11 $vorname  = readvarchar ("Vorname:   ");
12 $strasse  = readvarchar ("Strasse:   ");
13 $ort      = readvarchar ("Wohnort:   ");
14 $gehalt   = readvarchar ("Gehalt:    ");
15 $sth = $dbh->prepare ("SELECT aid FROM abteilung
16                       WHERE bezeichnung = ?");
17 do {
18     $abteilung = readvarchar ("Abteilung: ");
19     $sth->execute ($abteilung);
20     ($abt_id) = $sth->fetchrow_array();
21     if (!defined($abt_id)) {
22         print "Abteilung existiert nicht - Eingabe wiederholen!\n";
23     }
24 } while (!defined($abt_id));
25 print "Abteilungs-Id = $abt_id\n";
26 $sth = $dbh->prepare ("SELECT MAX(pid) FROM personal");
27 $sth->execute();
28 ($pers_id) = $sth->fetchrow_array();
29 $pers_id++;
30 print "Neue Personalnummer = $pers_id\n";
31 $sth = $dbh->prepare ("INSERT INTO personal
32                       (pid,nachname,vorname,strasse,ort,
33                       aid,gehalt,einstellung)
34                       VALUES (?, ?, ?, ?, ?, ?, ?, CURRENT_DATE)");
35 $rows = $sth->execute ($pers_id, $nachname, $vorname,
36                       $strasse, $ort, $abt_id, $gehalt);
37 if ($rows) {
38     print "Einfuegen erfolgt.\n";
39 } else {
40     print "Fehler beim Einfuegen.\n";
41 }
42 $dbh->disconnect ();

```

Abb. 90: Ein komplettes Programm in Perl/DBI

ggf. mehrfach mit verschiedenen Werten für den Abteilungsnamen ausführen (in Zeile 19). Dies ist effizienter, als wenn wir die Abfrage jedesmal von neuem vorbereiten und ausführen würden, da das Datenbanksystem dadurch die Möglichkeit hat, die bereits analysierte Struktur der Abfrage weiterzuverwenden.

Ob der Benutzer einen existierenden Abteilungsnamen eingegeben hat, stellen wir fest, indem wir versuchen, einen Datensatz der Ergebnistabelle zu lesen (Zeile 20). Ist die Ergebnistabelle leer, wird kein Datensatz geliefert, und die Variable `$abt_id` erhält einen undefinierten Wert. Diese Tatsache wird dann in Zeile 21 überprüft und auch in Zeile 24 als Iterationskriterium für die Schleife verwendet.

In den Zeilen 26–28 ermitteln wir die bisher höchste verwendete Personalnummer. Der um 1 erhöhte Wert (Zeile 29) wird dann die Personalnummer des neuen Mitarbeiters.

In den Zeilen 31–36 erfolgt die tatsächliche Einfügung des neuen Mitarbeiters. Der von der `execute`-Methode gelieferte Rückgabewert – er gibt bei fehlerfreier Ausführung die Anzahl der Änderungen in der Tabelle an – ist für die Fehlerkontrolle wichtig. Tritt nämlich ein Fehler auf, liefert `execute` einen undefinierten Wert. Dies kann in unserem Programm z. B. dann auftreten, wenn als Gehalt kein numerischer Wert eingegeben wurde. Dann wird die Ausführung von `execute` fehlschlagen und von DBI eine Warnung auf dem Bildschirm ausgegeben werden. Wir prüfen nun in Zeile 37 noch den Rückgabewert ab (ein undefinierter Wert wird von Perl als logisch falsch eingestuft) und geben eine entsprechende Rückmeldung auf dem Bildschirm aus.²³

7.3. Portabilität der Zugriffsverfahren

Wir haben nun zwei verschiedene Methoden vorgestellt, um von höheren Programmiersprachen aus mit SQL auf Datenbanken zugreifen zu können.

Der Zugriff über Embedded SQL ist im SQL-Standard festgeschrieben und erlaubt die Einstreuung von SQL-Konstrukten in Programme höherer Programmiersprachen. Welche Programmierspra-

²³ In der Praxis wird man natürlich bereits beim Einlesen des Gehalts sicherstellen, dass es sich um einen numerischen Wert handelt. Wir haben hier nur aus Gründen der Übersichtlichkeit des Programmcodes auf diese Prüfung verzichtet.

chen unterstützt werden, liegt im Ermessen des jeweiligen Datenbankherstellers. Die Verwendung von Embedded SQL-Programmen erfordert den Einsatz spezieller – datenbankspezifischer – Precompiler, die die Embedded SQL-Konstrukte in Aufrufe entsprechender Routinen der Clientbibliotheken umwandeln. Beim Compilieren und Binden (Linken) solcher Programmen müssen die datenbankspezifischen Clientbibliotheken in den ausführbaren Code eingebunden werden, was den Programmerstellungsprozess mitunter etwas unhandlich macht.

Hingegen ist DBI eine datenbankunabhängige Schnittstelle für den Zugriff von Datenbanken aus Perl-Programmen heraus. Sind DBI und die datenbankspezifischen DBD-Module auf einem System installiert, können entsprechende Perl-Programme ohne jeglichen Compilierungsaufwand ablaufen. Somit gestaltet sich die Benutzung von DBI gegenüber Embedded SQL wesentlich einfacher, ist allerdings nur in der Sprache Perl möglich.

Abgesehen von diesen entwicklungsumgebungsspezifischen Portierungsfragen ist natürlich auch interessant, welche Änderungen im Quellcode bei einem Wechsel des Datenbanksystems notwendig sind. Soweit die Datenbanksysteme hinsichtlich Embedded SQL dem SQL-Standard entsprechen, dürfte hauptsächlich die EXEC SQL CONNECT-Anweisung anzupassen sein. Bei Perl/DBI ist es ähnlich: Unter der Voraussetzung, dass für das Ziel-Datenbanksystem ebenfalls ein DBD-Modul existiert, ist im günstigsten Fall nur eine Anpassung des `connect`-Aufrufs erforderlich.

Leider ist es damit oft nicht getan, egal ob man die Embedded SQL- oder die Perl/DBI-Variante gewählt hat. Die Probleme liegen im spezifischen SQL-Dialekt, den ein bestimmtes Datenbanksystem unterstützt. Macht man beim Zugriff auf die Datenbank reichlich von “SQL-Spezialitäten” Gebrauch, die nicht standardisiert sind, sind ggf. sehr viele SQL-Anweisungen an ein anderes Datenbanksystem anzupassen.

Auch wenn man sich nur auf einfache SQL-Anweisungen beschränkt, gibt es erfahrungsgemäß an einer Stelle trotzdem Probleme – wenn nämlich Datums- und Zeitangaben mit den von der Datenbank angebotenen Typen verwaltet werden sollen. Da in den gängigen höheren Programmiersprachen keine äquivalenten Typen existieren (das trifft auch für Perl zu), ist man auf die – stets datenbankspezifischen – Vorgehensweisen für die Übertragung von Datums- und Zeitangaben zwischen Datenbank und höherer Programmiersprache angewiesen.

Wenn Portierbarkeit ein wichtiges Kriterium ist, sollte man genau überlegen, ob man Datums- und Zeitangaben auf Datenbankebene überhaupt mit den spezifischen Typen der Datenbank darstellt. Eine stets portierbare Möglichkeit der Darstellung von Datums- und Zeitangaben ist die Verwendung einer Zeichenkette, mit der man beispielsweise den Zeitpunkt 2. April 2001, 19:52:43 Uhr als

20010402195243

darstellen kann (also das Jahr mit 4 Stellen, dann Monat und Tag mit jeweils 2 Stellen, Stunde, Minute, Sekunde ebenfalls mit jeweils 2 Stellen). Entsprechend kann man natürlich reine Datums- oder reine Zeitangaben mit 8- bzw. 6-stelligen Zeichenketten darstellen. In diesem Fall kommt dann dem Host-Programm die Aufgabe zu, diese Zeichenketten jeweils bei Bedarf in eine “lesbare” (sprich: benutzerfreundliche) Darstellung umzuwandeln (und umgekehrt). Dies ist in Perl aber sehr leicht durchführbar.

Ein weiterer Fallstrick für die Portierung von Datenbankanbindungen ist eine unterschiedliche Unterstützung des Transaktionskonzeptes von verschiedenen Datenbanksystemen. Der Portierungsaufwand wird sicherlich minimiert, wenn man sich von vornherein hinsichtlich Transaktionen auf das minimal notwendige beschränkt und sich im Zweifelsfall nicht auf Besonderheiten eines bestimmten Datenbanksystems verlässt.

Man kann jedoch zusammenfassend sagen, dass sowohl Embedded SQL als auch Perl/DBI in vielen Bereichen eine Portierung zwischen verschiedenen Datenbanksystemen überschaubar gestalten. Natürlich hat man, wenn man nicht (nur) die Datenbank, sondern auch die Hardware- oder Betriebssystemplattform wechselt, im Falle von Embedded SQL mit den üblichen Portierungsschwierigkeiten des Host-Programms zu kämpfen. Hier genießt Perl den klaren Vorteil einer plattformübergreifenden Verfügbarkeit.

8. WWW-Integration von Datenbanken

Wie bereits im Abschnitt 3.3 bemerkt, wird ein Endbenutzer im Normalfall nicht direkt durch Absetzen von SQL-Anweisungen auf eine Datenbank zugreifen. Üblicherweise erhält ein Benutzer vom Datenbankprogrammierer eine mehr oder weniger komfortable Benutzeroberfläche (engl. user interface) zur Verfügung gestellt.

8.1. Kommandozeilen- und graphische Benutzeroberflächen

Bei einer Benutzeroberfläche kann es sich im einfachsten Fall um eine kommandozeilenorientierte Oberfläche (engl. command line interface, CLI) handeln – die in Abschnitt 7 vorgestellten Programme zum Eintrag eines neuen Mitarbeiters in unsere Musterdatenbank fallen beispielsweise in diese Kategorie.

Heutzutage gelten allerdings kommandozeilenorientierte Oberflächen als nicht mehr unbedingt zeitgemäß. Mit der stetig zunehmenden Leistungsfähigkeit von Computerhardware haben sich seit Ende der 80er-Jahre immer mehr graphische Benutzeroberflächen (engl. graphical user interface, GUI) durchgesetzt.

Leider ist man mit der Implementierung einer graphischen Oberfläche oft auf ein bestimmtes Betriebssystem festgelegt. Außerdem ist die GUI-Entwicklung auch bei Einsatz entsprechender Entwicklungswerkzeuge eine nichttriviale Aufgabe.

Zwischen kommandozeilenorientierten Textinterfaces und graphischen Benutzeroberflächen gibt es einen grundlegenden programmiertechnischen Unterschied. Eine kommandozeilenorientierte Oberfläche ist programmgesteuert, d. h. das Programm, das die Oberfläche implementiert, hat die völlige Kontrolle über den Ablauf (Abb. 91).

Hingegen sind graphische Oberflächen ereignisgesteuert, d. h. für verschiedene Benutzeraktionen (z. B. Drücken einer Taste, Eingabe in ein Feld, Mausklick auf einen Button, Anwahl eines Menüpunktes) werden spezielle Programmteile ausgeführt. Das Programm, das die Oberfläche implementiert, kann diese Benutzeraktionen nicht voraussehen und auch keine dahingehenden Annahmen machen (Abb. 92).

Diese Nichtlinearität von GUIs im Vergleich zu CLIs macht die GUI-Implementierung wesentlich komplizierter.

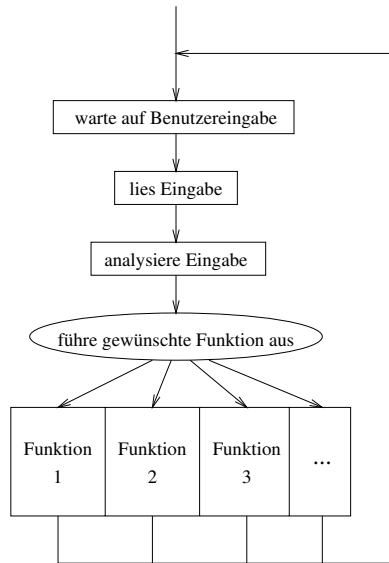


Abb. 91: Programmgesteuerter Ablauf einer Oberfläche

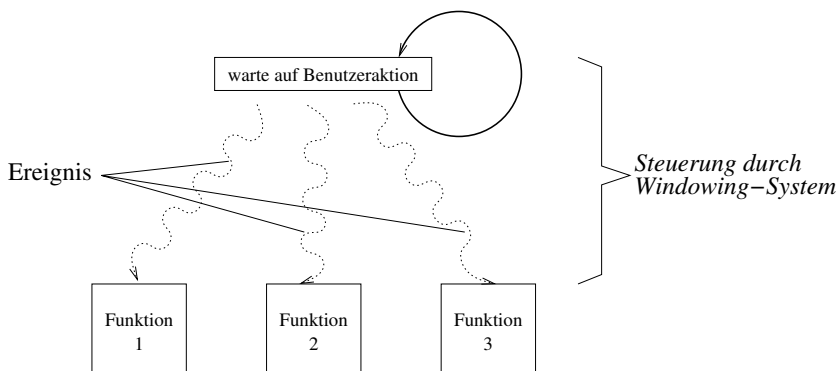


Abb. 92: Ereignisgesteuerter Ablauf einer Oberfläche

Für die meisten kommerziellen Datenbanksysteme gibt es Entwicklungstools, die speziell auf die Erstellung von GUIs zur Bedienung von Datenbanken zugeschnitten sind und somit die GUI-Programmierung wesentlich erleichtern.

Andererseits führt der steigende Vernetzungsgrad von Rechnersystemen zu einer immer weitergehenden Trennung von Benutzeroberfläche und “eigentlicher” (von der Oberfläche losgelösten) Applikation. Am Ende dieses Trennungsprozesses steht der sog. **Thin Client**,

auf dem nur noch die Benutzeroberfläche dargestellt wird.

Insbesondere hinsichtlich Datenbankanbindung gibt es bei herkömmlicher GUI- (oder auch CLI-) Programmierung Grenzen der Separationsmöglichkeiten in Oberflächen- und Applikationsteil.

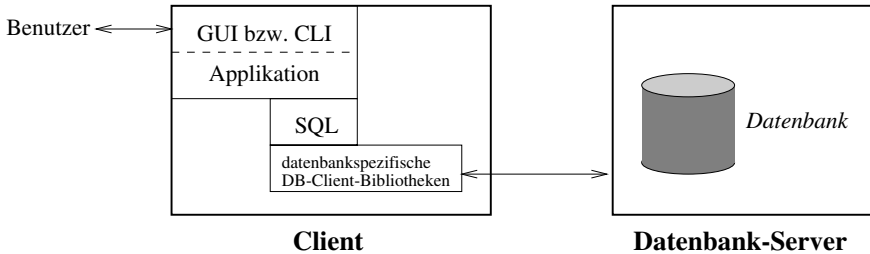


Abb. 93: Fat Client und Datenbank-Server

In Abb. 93 ist deutlich zu erkennen, dass für Zugriffe auf einen Datenbank-Server spezifische Datenbank-Clientsoftware auf dem Client installiert sein muss (sog. **Fat Client**-Modellierung).

Für eine Thin Client-Modellierung müssen GUI und Applikation möglichst gut getrennt werden, wie dies durch Abb. 94 vorgeschlagen wird. Hierbei können Applikations- und Datenbank-Server zwei getrennte Rechner sein; dies ist jedoch nicht unbedingt erforderlich.

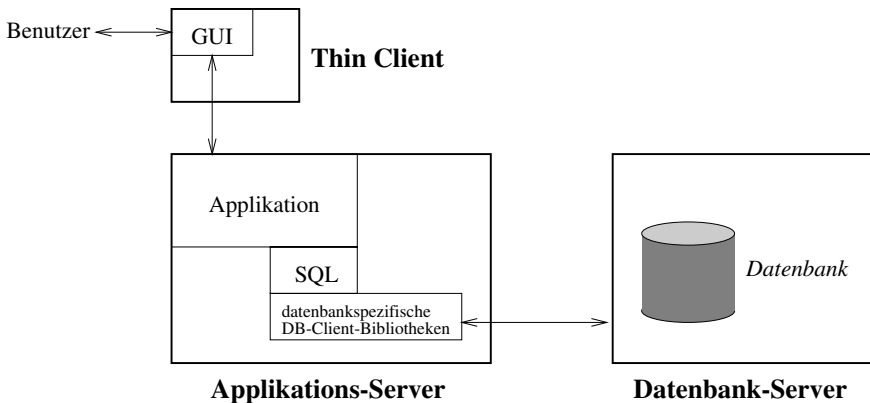


Abb. 94: Thin Client, Applikations- und Datenbank-Server

In der Welt der Unix-Betriebssysteme kann mit Hilfe des X-Windows-Systems auf einfache Weise eine Trennung von Applikationsserver und GUI-Anzeigerechner erreicht werden. Dies geschieht, indem die graphische Darstellung der Applikation nicht an den Rechner gebunden ist, auf dem die Applikation selbst läuft, sondern via Netzwerk auf einem beliebigen anderen Rechner erfolgen kann (Abb. 95). Auf letzterem muss die sog. X-Server-Software dafür sorgen, dass die vom Applikationsserver produzierte Graphikausgabe auf dem Bildschirm angezeigt wird.

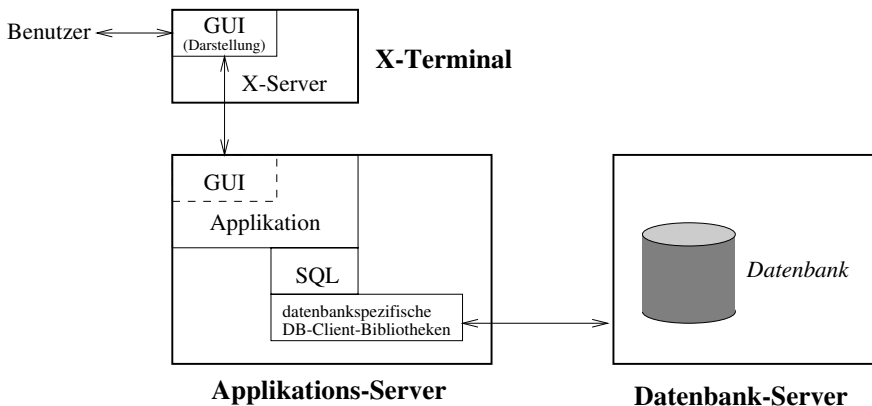


Abb. 95: X-Terminal, Applikations- und Datenbank-Server

Sogenannte X-Terminals sind Rechner, auf denen ausschließlich die X-Server-Software zur Anzeige von X-Windows-Applikationen abläuft.²⁴

8.2. Benutzeroberflächen im World Wide Web

Eine gute Benutzeroberfläche sollte idealerweise überall – also plattform- und betriebssystemunabhängig – verfügbar sein. Wie im letzten Abschnitt erläutert, ist dies für herkömmliche graphische Oberflächen kaum gegeben. Mit dem Aufkommen des World Wide Web (WWW) hat man jedoch die Möglichkeit, eine Benutzeroberfläche

²⁴ CLIs unter Unix können auf jedem anderen Rechner ausgeführt werden, der über einen Telnet-Client verfügt, da zu einem Unix-Rechner via Netzwerk Sitzungen auf Kommandozeilenebene aufgebaut werden können. Somit ist für die Netzwerkanbindung von CLIs unter Unix kein besonderer Hard- oder Softwareaufwand erforderlich.

über sog. Web-Formulare abzubilden, die dann die folgenden sehr wünschenswerten Eigenschaften besitzt:

- Darstellung durch auf den meisten Plattformen verfügbare WWW-Browser.
- Unterstützung des Thin Client-Modells – auf Benutzerseite muss lediglich ein WWW-Browser verfügbar sein.
- Standardisierte Sprache zur Darstellung von WWW-Seiten: HTML (“Hypertext Markup Language”).
- Standardisiertes Protokoll zum Datenaustausch zwischen WWW-Browser und WWW-Server: HTTP (“Hypertext Transfer Protocol”).
- Standardisiertes Verfahren zum Datenaustausch zwischen WWW-Browser (Benutzer) und Applikation: CGI (“Common Gateway Interface”).

Eine im WWW realisierte Applikation besteht im einfachsten Fall aus einer Kollektion von sog. **CGI-Skripten**, die Formulareingaben verarbeiten und als Ausgabe weitere HTML-Seiten produzieren, die im WWW-Browser des Benutzers angezeigt werden.²⁵

Wir wollen im Rahmen dieses Buches nur serverbasierte Applikationen betrachten, bei denen der WWW-Browser auf Benutzerseite ausschließlich HTML-Seiten anzeigen und Formulareingaben an den WWW-Server übermitteln soll. Es gibt darüber hinaus Techniken, bei denen auch der WWW-Browser sog. **aktive Inhalte** etwa in Form von Java- oder JavaScript-Programmen ausführt und somit die Applikation nicht mehr rein serverbasiert ist. Obwohl diese Technik größere Freiheit bei der Gestaltung einer Benutzeroberfläche gewährt, macht sie die Erstellung einer solchermaßen “verteilten” Applikation wesentlich komplizierter. Außerdem werden in den Browser-Implementierungen immer wieder z. T. schwerwiegende Sicherheitslücken entdeckt, so dass eine Verwendung aktiver Inhalte in sicherheitskritischen Umgebungen fraglich scheint.

²⁵ Außer der hier vorgestellten Technik der Realisierung einer Applikation im WWW über CGI-Skripten gibt es noch andere Techniken. Beispielsweise erlaubt die Sprache PHP eine direkte Einbettung des Programmcodes in HTML-Seiten und wird daher der CGI-Programmierung mitunter vorgezogen. Jedoch ist die Flexibilität der PHP-Variante etwas geringer.

8.3. Architektur einer WWW-Oberfläche mit CGI-Skripten

Bei einem CGI-Skript handelt es sich um ein Programm in einer mehr oder weniger beliebigen Programmiersprache, dessen Ausführung vom WWW-Serverprozess beim Aufruf einer bestimmten WWW-Adresse durch den WWW-Browser angestoßen wird (Abb. 96). Dieses Skript kann vom WWW-Browser Benutzereingaben übermittelt bekommen, falls es aus einem Web-Formular heraus aufgerufen wurde.

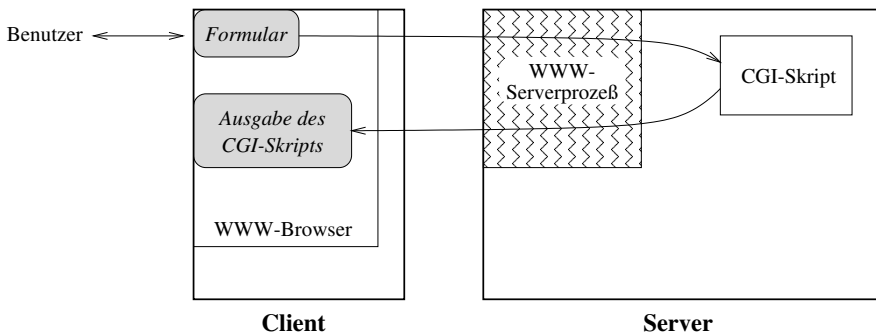


Abb. 96: Ablauf der Ausführung eines CGI-Skripts

Die Ausgaben des CGI-Skripts werden vom WWW-Server als neue WWW-Seite zurück an den Browser übermittelt, was bedeutet, dass ein CGI-Skript in den meisten Fällen HTML-Code als Ausgabe produzieren wird. So entstandene WWW-Seiten heißen **dynamisch**, da sie bei jedem Aufruf des CGI-Skripts neu erzeugt werden und der Inhalt jedes Mal ein anderer sein kann. Hingegen handelt es sich bei statischen WWW-Seiten um HTML-Code, der bereits in fester Form (als Datei) auf dem WWW-Server vorliegt und beim Aufruf unverändert an den Browser übermittelt wird.

Obwohl CGI-Skripten in einer beliebigen auf dem WWW-Server unterstützten Programmiersprache verfasst sein können, wird oftmals die Sprache Perl verwendet – insbesondere die Anbindung von Datenbanken ist mit Perl sehr einfach möglich, wie wir in Abschnitt 7.2 gesehen haben. Perl erleichtert die Programmierung von CGI-Skripten zusätzlich, da es ein spezielles CGI-Modul für Perl gibt, das dem Programmierer sowohl die Auswertung der über ein Formular enthaltenen Benutzereingaben als auch die Generierung von HTML-Ausgabe abnehmen kann. Wir wollen daher im folgenden die Programmierung von

CGI-Skripten mit Perl unter Verwendung der CGI- und DBI-Module an charakteristischen Beispielen vorstellen.

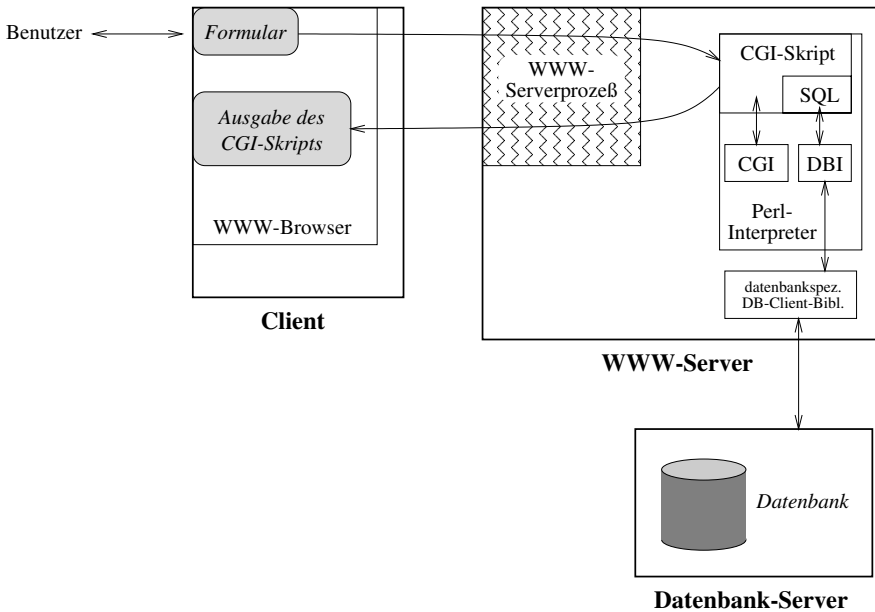


Abb. 97: Datenbankzugriff aus dem WWW mit CGI und DBI

Bei einer per WWW via CGI angebundenen Datenbank ist das Gesamtsystem wie in Abb. 97 dargestellt strukturiert. Die Anbindung von Datenbanken mit dem DBI-Modul haben wir ja schon in Abschnitt 7.2 behandelt, so dass wir uns hier vor allem auf die Interaktion zwischen Browser (Benutzer) und CGI-Skript konzentrieren wollen.

Wir illustrieren im folgenden unsere Ausführungen wieder mit dem bereits bekannten Beispiel des Eintragens eines neuen Mitarbeiters in unsere Musterdatenbank. In diesem Fall soll dies natürlich per WWW unter Verwendung von Web-Formularen und CGI-Skripten geschehen.

8.4. Sammeln von Eingaben mit Web-Formularen

Bei einem Web-Formular handelt es sich um eine WWW-Seite, die durch spezielle HTML-Anweisungen („Tags“) die Fähigkeit erhält, Be-

nutzereingaben zu ermöglichen.²⁶ Ein Web-Formular kann verschiedene Elemente besitzen, mit denen Eingaben vorgenommen bzw. Aktionen veranlasst werden können. Die folgende Auflistung stellt lediglich eine Auswahl dar:

- *Textfelder* erlauben die Eingabe beliebiger Zeichenketten.
- *Checkboxes* ermöglichen die An- oder Abwahl vorgegebener Optionen.
- *Radio-Buttons* ermöglichen die Auswahl einer Option aus verschiedenen Möglichkeiten.
- *Submit-Buttons* veranlassen die Übermittlung der Eingaben des Formulars an den WWW-Server, die von diesem wiederum an ein im Formular spezifiziertes CGI-Skript weitergereicht werden. Die Ausgabe dieses CGI-Skripts wird anstelle des Formulars als neue WWW-Seite angezeigt.
- *Reset-Buttons* löschen alle im Formular gemachten Eingaben bzw. stellen die Standardbelegung der Eingaben wieder her. Ansonsten passiert nichts, das Formular wartet dann weiterhin auf Eingaben.

Die Formularelemente werden auf HTML-Ebene als HTML-Tags spezifiziert. Jedem Formularelement ist ein Name und ein Wert zugeordnet. Auf diese Weise kann das von dem Formular aufgerufene CGI-Skript die Parameterwerte ähnlich wie Variablenwerte bestimmen.

Da wir uns im Rahmen dieses Buches auf die WWW-Anbindung von Datenbanken konzentrieren wollen, erläutern wir hier nicht genauer, wie die Übertragung von Formulareingaben an ein CGI-Skript technisch abläuft. Der interessierte Leser sei hierzu auf entsprechende Literatur, z. B. [GuGuBi2000] verwiesen. Dort finden sich auch Ausführungen zu Sicherheitsaspekten bei der Realisierung von CGI-Skripten.

Wir wollen nun ein Formular betrachten, das die zum Eintragen eines neuen Mitarbeiters benötigten Daten vom Benutzer erfassen soll. Dieses könnte wie in Abb. 98 gezeigt aussehen.

Der HTML-Code, der dieses Formular erzeugt, ist in Abb. 99 aufgelistet. Der hier interessante Code befindet sich in den Zeilen 5–16; der Rest ist standardmäßiges HTML.

²⁶ Wir setzen hier voraus, dass der Leser mit den Grundlagen von HTML vertraut ist. Eine ausführliche Darstellung von HTML findet sich beispielsweise in [MuKe2002].

Abb. 98: Web-Formular zum Eintragen eines neuen Mitarbeiters

```

1  <HTML>
2  <HEAD><TITLE>Neuanlage Mitarbeiter</TITLE></HEAD>
3  <BODY>
4  <H1>Eintrag eines neuen Mitarbeiters</H1>
5  <FORM ACTION="neumitwww2.pl">
6  <TABLE>
7  <TR><TD>Nachname: </TD>      <TD><INPUT TYPE="text" NAME="nachname" ></TD></TR>
8  <TR><TD>Vorname: </TD>        <TD><INPUT TYPE="text" NAME="vorname" ></TD></TR>
9  <TR><TD>Straße: </TD>          <TD><INPUT TYPE="text" NAME="strasse" ></TD></TR>
10 <TR><TD>Wohnort: </TD>          <TD><INPUT TYPE="text" NAME="ort" ></TD></TR>
11 <TR><TD>Gehalt: </TD>           <TD><INPUT TYPE="text" NAME="gehalt" ></TD></TR>
12 <TR><TD>Abteilung: </TD>       <TD><INPUT TYPE="text" NAME="abteilung" ></TD></TR>
13 </TABLE>
14 <INPUT TYPE="submit" NAME="aktion" VALUE="Mitarbeiter anlegen">
15 <INPUT TYPE="reset" VALUE="Eingaben löschen">
16 </FORM>
17 </BODY>
18 </HTML>

```

Abb. 99: HTML-Code zum Erzeugen des Web-Formulars

Damit das Formular “schön” gegliedert ist, haben wir die Beschreibungen der Eingabefelder und die Felder selbst in einer Tabelle organisiert – so stehen die Eingabefelder ordentlich untereinander. Für die Funktion des Formulars ist die Verwendung einer Tabellenstruktur allerdings unerheblich.

Ein Web-Formular wird mit dem `<FORM>`-Tag eingeleitet; dieser besitzt hier ein Attribut `ACTION`. Mit diesem Attribut wird die WWW-Adresse des CGI-Skripts angegeben, das beim Anklicken eines Submit-Buttons ausgeführt werden soll und das auch die Formulardaten erhält. Die WWW-Adresse muss eine gültige URL (“Uniform Resource Loca-

tor”) sein; da wir hier nur einen Dateinamen angegeben haben, bezieht er sich auf eine Datei (das CGI-Skript), die serverseitig im gleichen Verzeichnis abgelegt ist wie das Formular.

Die einzelnen Eingabefelder werden in den Zeilen 7–12 jeweils mit einem INPUT-Tag festgelegt. Das Attribut TYPE legt die Art des Formularelements fest (hier jeweils ein Textfeld), NAME bestimmt den Namen des Formularelements. Mit einem weiteren Attribut VALUE könnten wir noch Standardwerte für die Eingabefelder festlegen, z. B. eine Vorgabe für den Wohnort.

Zeile 14 definiert einen Submit-Button; der “Wert” wird im Browser als Beschriftung des Buttons angezeigt. In Zeile 15 spezifizieren wir einen Reset-Button, mit dem alle Formulareingaben gelöscht werden können.

8.5. Auswertung von Formulareingaben mit CGI-Skripten

Wie sieht nun das CGI-Skript `neumitwww2.pl` aus, das Eingaben im Formular aus Abb. 98 auswertet und den neuen Mitarbeiter in die Datenbank einträgt? Wir hatten weiter oben schon angedeutet, dass wir das Skript in Perl unter Verwendung des DBI- sowie des CGI-Moduls programmieren wollen. Bei dem CGI-Modul handelt es sich im Prinzip um eine Perl-Programmbibliothek, die dem Programmierer das Einlesen der Formulareingaben abnimmt und außerdem die Erzeugung der vom Skript zu liefernden WWW-Seite wesentlich vereinfacht.

In Abb. 100 ist ein Listing des CGI-Skripts `neumitwww2.pl` abgebildet. Zeile 2 aktiviert ein sogenanntes *Pragma*, hier die Notwendigkeit, Variablen vor Gebrauch zu definieren. Dies ist insbesondere bei CGI-Skripten sehr sinnvoll.²⁷

²⁷ Der Grund dafür liegt in der Tatsache, dass es bei manchen WWW-Servern möglich ist, Perl-Programme mit einem bereits gestarteten Perl-Interpreter ausführen zu lassen. Bei erfolgter Ausführung des Skripts wird der Interpreter nicht beendet, sondern kann ein weiteres Skript ausführen. Werte von Variablen, die nicht lokal für das Skript definiert sind, bleiben dann erhalten und beeinträchtigen möglicherweise den Ablauf folgender Skripten. Da Perl (ohne `use strict`) die Eigenschaft hat, bei Verwendung einer undefinierten Variable automatisch eine globale Variable zu erzeugen, wäre die unkontrollierte Verwendung von Variablennamen gefährlich für die Stabilität der WWW-Serverumgebung.

```

1  #!/usr/bin/perl -wT
2  use strict;
3  use CGI qw(:standard);
4  use CGI::Carp qw(fatalsToBrowser);
5  use DBI;
6
7  my ($i,%p,$dbh,$sth,$abt_id,$pers_id,$rows);
8
9  print header,
10     start_html ("Neuanlage Mitarbeiter - Ergebnis");
11
12  foreach $i ("nachname","vorname","strasse","ort",
13             "gehalt","abteilung") {
14     $p{$i} = param($i);
15 }
16
17 $dbh = DBI->connect ("dbi:Pg:dbname=firma", "rank");
18 $sth = $dbh->prepare ("SELECT aid FROM abteilung
19                      WHERE bezeichnung = ?");
20 $sth->execute ($p{abteilung});
21 ($abt_id) = $sth->fetchrow_array ();
22
23 if (!defined($abt_id)) {
24     print "Abteilung existiert nicht!",
25     end_html;
26     exit;
27 }
28
29 $sth = $dbh->prepare ("SELECT MAX(pid) FROM personal");
30 $sth->execute ();
31 ($pers_id) = $sth->fetchrow_array ();
32 $pers_id++;
33
34 $sth = $dbh->prepare ("INSERT INTO personal
35                      (pid,nachname,vorname,strasse,ort,
36                      aid,gehalt,einstellung)
37                      VALUES (?, ?, ?, ?, ?, ?, ?, CURRENT_DATE)");
38 $rows = $sth->execute ($pers_id, $p{nachname}, $p{vorname},
39                      $p{strasse}, $p{ort}, $abt_id, $p{gehalt});
40
41 if ($rows) {
42     print "Neuer Mitarbeiter mit Personal-Id $pers_id ";
43     print "und Abteilungs-Id $abt_id eingef&uuml;gt.";
44 } else {
45     print "Fehler beim Einf&uuml;gen.";
46 }
47
48 print end_html;

```

Abb. 100: CGI-Skript zum Eintragen eines neuen Mitarbeiters

In Zeile 3 wird das CGI-Modul geladen. Dieses Modul bietet eine objektorientierte und eine "konventionelle" Schnittstelle an. Wir verwenden hier aus Gründen der Einfachheit die konventionelle Schnittstelle, mit der auf das CGI-Modul über gewöhnliche Funktionsaufrufe zugegriffen werden kann; dies geben wir bei der `use`-Anweisung über die Klausel `qw(:standard)` an.

Mit der Anweisung in Zeile 4 wird während der Entwicklung eines

CGI-Skripts eine leichtere Fehlersuche ermöglicht: Fehler, die bei der Ausführung des Skripts entstehen, werden dann im WWW-Browser des Benutzers angezeigt. Standardmäßig wird nämlich eine eventuelle Fehlerausgabe nur in die Protokolldatei des WWW-Servers geschrieben, was die Fehlersuche schwieriger gestaltet.²⁸

In Zeile 5 wird in bekannter Weise das DBI-Modul geladen. In Zeile 6 definieren wir die benötigten lokalen Variablen – dies ist wegen `use strict` in Zeile 2 erforderlich.

Wie bereits angedeutet, soll unser CGI-Skript als Ausgabe eine HTML-Seite produzieren. Diese müssen wir nicht “von Hand” erzeugen, sondern wir können entsprechende Funktionen des CGI-Moduls aufrufen. In den Zeilen 7 und 8 werden damit die üblichen Tags für den Anfang einer HTML-Seite erzeugt. Der Aufruf von `start_html` produziert den im Kopfbalken des Browsers angezeigten Text, also hier

Neuanlage Mitarbeiter – Ergebnis

Nun lesen wir die vom Formular übergebenen Werte ein. Diese erhält man durch Aufruf der Funktion `param`, wobei diese als Parameter den Namen des Formularfeldes, dessen Wert wir bestimmen wollen, benötigt. In den Zeilen 9–12 werden die Werte aller Felder unseres Formulars in den Hash `%p` übertragen.²⁹ Nach Ausführung dieser Zeilen enthält

`$p{feldname}`

jeweils den Wert des Formularfeldes mit dem Namen *feldname*.

Der Rest des Skripts läuft strukturmäßig ganz ähnlich wie unser kommandozeilenorientiertes Skript aus Abb. 90, allerdings mit dem Unterschied, dass wir bei Ausgaben berücksichtigen müssen, dass diese Bestandteil einer HTML-Seite werden, die der Benutzer im Browser angezeigt erhält.

Betrachten wir etwa die Zeilen 18–22: Wir haben festgestellt, dass die im Formular eingegebene Abteilung nicht existiert. Wir können nun nicht den Benutzer wie in der kommandozeilenorientierten Ver-

²⁸ In der Produktionsversion eines CGI-Skripts sollte man allerdings aus Sicherheitsgründen dem Benutzer keine internen Fehlermeldungen anzeigen.

²⁹ Für Perl-Neulinge: Ein Hash kann vereinfacht als Array angesehen werden, nur dass die Elemente nicht nur mit Zahlen, sondern beliebigen Zeichenketten – sogenannten Schlüsseln, hier den Parameternamen – indiziert werden können.

sion nochmals innerhalb des Skripts zur Neueingabe auffordern und dies so lange wiederholen, bis eine existierende Abteilung eingegeben wird. Eine Fehlerbehandlung dieser Art muss mit Formularen und CGI-Skripten etwas anders erfolgen – wir werden uns weiter unten ausführlicher mit dieser Thematik auseinandersetzen. An dieser Stelle begnügen wir uns damit, dem Benutzer mit einem simplen Hinweis anzuzeigen, dass die eingegebene Abteilung nicht existiert.

Dieser Hinweis wird in Zeile 19 als gewöhnlicher Text ausgegeben. Bevor wir das Skript beenden, müssen wir noch dafür sorgen, dass die HTML-Seite korrekt mit den entsprechenden Tags beendet wird, wofür der Funktionsaufruf `end_html` als Parameter von `print` sorgt. Bei einer nicht existenten Abteilung erhält der Benutzer also die in Abb. 101 dargestellte WWW-Seite angezeigt.

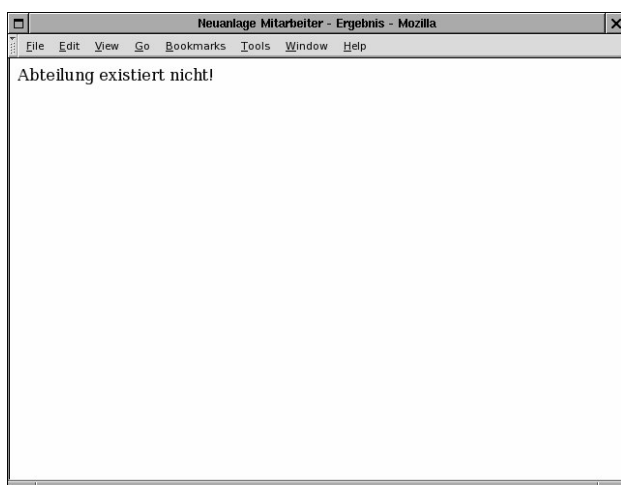


Abb. 101: Ergebnisseite des Skripts bei nicht existenter Abteilung

Dies ist nicht sonderlich benutzerfreundlich, da der Benutzer dann nur mit Hilfe der Browser-Navigationsfunktionen wieder zur Formularseite zurückkommen kann, aber wir werden wie gesagt weiter unten eine komfortablere Version des Skripts entwerfen.

Die zweite Stelle, an der in unserem Skript Ausgaben erzeugt werden, ist die Rückmeldung an den Benutzer über den Erfolg der Einfügeoperation. Im positiven Fall geben wir eine entsprechende Meldung mit Personal-Id und Abteilungs-Id des neu erzeugten Datensatzes aus, ansonsten die Meldung über den Fehler.

In Zeile 39 wird die HTML-Seite mit den richtigen Tags abgeschlossen.

8.6. Erzeugen von Formularen mit CGI-Skripten

Nachdem das CGI-Modul auch zum Erzeugen von HTML-Seiten verwendet werden kann, besteht kein Grund, sich bei der Erstellung des Eingabeformulars selbst mit HTML-Tags “herumzuschlagen”. Lassen wir doch das CGI-Modul diese Arbeit tun und formulieren das Formular ebenfalls als CGI-Skript (Abb. 102), das wir unter dem Namen `neumitwww.pl` speichern wollen.

```
1  #! /usr/bin/perl -w
2  use strict;
3  use CGI qw(:standard);
4  use CGI::Carp qw(fatalsToBrowser);
5
6  print header,
7      start_html ("Neuanlage Mitarbeiter"),
8      h1 ("Eintrag eines neuen Mitarbeiters"),
9      start_form (-action => "neumitwww2.pl"),
10     table(
11         Tr(td("Nachname: "),      td(textfield (-name => "nachname"))),
12         Tr(td("Vorname: "),      td(textfield (-name => "vorname"))),
13         Tr(td("Stra&szlig;e: "), td(textfield (-name => "strasse"))),
14         Tr(td("Wohnort: "),      td(textfield (-name => "ort"))),
15         Tr(td("Gehalt: "),      td(textfield (-name => "gehalt"))),
16         Tr(td("Abteilung: "),    td(textfield (-name => "abteilung"))),
17     ),
18     submit (-name => "aktion", -value => "Mitarbeiter anlegen"),
19     reset (-value => "Eingaben löschen"),
20     end_form, end_html;
```

Abb. 102: CGI-Skript zum Erzeugen des Web-Formulars

Dieses Skript erzeugt den in Abb. 99 dargestellten HTML-Code.³⁰ Nun mag man einwenden, dass sich hier gegenüber der direkten Formulierung des HTML-Codes keine große Vereinfachung ergeben hat. Der große Vorteil einer skriptgesteuerten Formularerzeugung ist jedoch, dass man die Möglichkeit hat, auch die Formularseite dynamisch zu gestalten. Dies findet Anwendung bei der Interaktion mit

³⁰ Tatsächlich ist der vom Skript erzeugte HTML-Code geringfügig anders als der handgestrickte. Die Unterschiede haben mit der Art der Datenübertragung vom Formular zum Aktions-CGI-Skript zu tun, auf die wir hier nicht eingehen wollen. Jedenfalls verhält sich das von diesem Skript erzeugte Formular für den Benutzer genau so wie das statische Formular.

Web-Formularen – u. a. auch für eine komfortable Fehlerbehandlung –, mit der wir uns im folgenden befassen wollen.

8.7. Interaktion mit Web-Formularen

Im Vergleich zu einem konventionellen GUI findet der programmtechnische Umgang mit Web-Formularen in linearer Weise statt, da es nur ein einziges Ereignis gibt, das eine Aktion auslöst, nämlich das “Abschicken” des Formulars durch Klicken auf einen Submit-Button. Demzufolge kann auch eine Konsistenz- und Fehlerprüfung der Eingaben erst stattfinden, wenn das Formular abgeschickt wurde und das Aktionsskript ausgeführt wird.

Im Gegensatz dazu können konventionelle GUIs diese Prüfungen unmittelbar vornehmen, z. B. dann, wenn der Benutzer den Cursor in ein anderes Eingabefeld setzt. Diese weniger große Flexibilität von Web-Formularen wird aber durch ihre Plattformunabhängigkeit weitgehend aufgewogen.

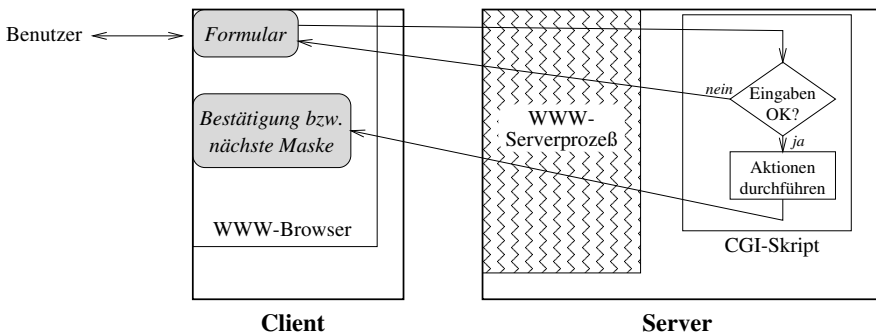


Abb. 103: Fehlerprüfung bei Web-Formularen

Ein typischer Ablauf der Auswertung eines Web-Formulars kann daher so aussehen wie in Abb. 103 gezeigt. Das beim Abschicken des Formulars ausgeführte CGI-Skript wird zunächst die Eingaben des Benutzers auf Fehlerfreiheit und Konsistenz prüfen. Im negativen Fall wird der Benutzer nochmals ein Formular angezeigt bekommen, das ihm Gelegenheit gibt, die unzulässigen Eingaben zu verbessern.

Nur bei Fehlerfreiheit der Eingaben wird das Skript die eigentlich beabsichtigte Aktion ausführen und dann dem Benutzer eine Meldung über das Ergebnis anzeigen. Wenn es sich um eine größere Applikation

handelt, wird der Benutzer stattdessen ein neues Formular angezeigt bekommen, in dem er die nächsten Eingaben machen muss.

Die eigentliche Aktion kann auch fehlschlagen; in unserem Beispiel könnten beim Einfügen in die Datenbank Fehler auftreten. Dann muss auch dies dem Benutzer angezeigt werden; der weitere Ablauf hängt dann nicht nur von der Art des Fehlers, sondern auch von dem Umfang der Fehlerbehandlung ab, die man realisieren möchte.

Wir haben in den letzten Absätzen immer von “dem” Skript gesprochen – natürlich können mehrere Skripten beteiligt sein, beispielsweise nach folgender Struktur: Das erste Skript erzeugt das initiale Formular, das zweite bildet das Aktionsskript für dieses Formular, wertet also die Eingaben aus (und muss ggf. nochmals ein Formular erzeugen) und führt die gewünschte Aktion aus. Beim Auftreten von Fehlern, die nicht auf Benutzereingaben zurückzuführen sind, könnte der Benutzer ein weiteres Formular angezeigt bekommen, das als Aktionsskript ein drittes Skript hat usw.

Allerdings gibt es auch den programmiertechnischen Ansatz, alle Aktivitäten, die mit einem Formular (einer Eingabemaske) zu tun haben, in nur einem Skript zu halten. Dieses wird dann zwar umfangreicher, besitzt aber den Vorteil, dass der Code, der dem gleichen Zweck dient, auch in der gleichen Datei steht, was die Struktur klarer macht. Man sollte hier immer daran denken, dass in der Praxis bei einer per WWW angebundenen größeren Datenbankapplikation nicht nur eine Eingabemaske, sondern deren viele realisiert werden müssen.

Wir werden aus diesem Grund für unser Beispiel des Eintragens eines neuen Mitarbeiters in die Musterdatenbank mit Fehlerkontrolle ein einziges Skript zur Realisierung verwenden.

Unsere WWW-Oberfläche soll nun so aussehen, dass beim ersten Aufruf des CGI-Skripts ein Formular wie in Abb. 98 angezeigt wird. Wenn der Benutzer nach Bearbeitung des Formulars auf den “Mitarbeiter anlegen”-Button klickt, wird das gleiche CGI-Skript wieder aufgerufen. Es muss nun feststellen, dass Eingaben vorliegen und diese überprüfen.

Im Beispiel wollen wir sicherstellen, dass zumindest ein Nachname eingegeben wurde und die eingegebene Abteilung existiert. (Wir prüfen hier nicht separat auf eine nicht eingegebene Abteilung, da die Abteilung mit “leerer” Bezeichnung ohnehin nicht existiert.) Wird einer dieser beiden Fehlerfälle festgestellt, wird keine Änderung an der Datenbank vorgenommen, sondern der Benutzer bekommt das Formular

nochmals präsentiert, allerdings mit Hinweisen, an welchen Stellen er falsche Eingaben gemacht hat.

In Abb. 104 ist gezeigt, wie der Eintrag eines neuen Mitarbeiters unter “Ausschöpfung” aller Fehlerfälle ablaufen soll. In a) wird das Formular abgeschickt, ohne dass ein Nachname eingegeben wurde. Der Benutzer bekommt daraufhin nochmals das gleiche Formular präsentiert, allerdings mit dem Hinweis, dass er einen Nachnamen eingeben muss.

Dies tut der Benutzer in b); beim Abschicken dieses Formulars stellt das System fest, dass es die Abteilung “Herstellung” nicht gibt und weist den Benutzer bei erneuter Anzeige des Formulars auf diese Tatsache hin. In c) hat der Benutzer den Abteilungsnamen durch “Produktion” ersetzt; nun kann das Einfügen des neuen Mitarbeiters erfolgen.

Wir wollen uns nun die Realisierung dieser Oberfläche als CGI-Skript ansehen. Der Code ist in Abb. 105 und Abb. 106 gezeigt (auf die grau unterlegten Zeilen werden wir in Abschnitt 8.9 nochmals besonderen Bezug nehmen). Es fällt zunächst auf, dass der Code wesentlich umfangreicher ist als unsere erste Version der Oberfläche aus Abb. 100 und Abb. 102. Dieses Verhalten entspricht aber der allgemeinen Beobachtung im Bereich der Softwareentwicklung, dass eine komfortable Fehlerbehandlung einen wesentlichen Teil des Codes ausmachen kann. Wir wollen uns im folgenden diesen Code genauer ansehen.

Die ersten Zeilen des Skripts sind im wesentlichen schon aus der Vorversion von Abb. 102 bekannt. Allerdings taucht in Zeile 3 das zusätzliche Konstrukt `*table` auf. Damit erreichen wir, dass uns zwei zusätzliche Funktionen `start_table` und `end_table` zur Verfügung stehen, mit denen wir Beginn- und Ende-Tags für Tabellen erzeugen können. Dies war in der Vorversion nicht notwendig, da wir die Tabelle mit den Eingabefeldern sozusagen “in einem Rutsch” als Argument der `Tr`-Funktion erzeugt hatten. Wir wollen aber nun die Formularfelder parameterisieren, d. h. Namen und Bezeichnungen aus einer Variablen auslesen.

Zeile 6 definiert die vom Skript verwendeten lokalen Variablen:

<code>\$i</code>	ist eine Schleifenvariable.
<code>%p</code>	In diesem Hash werden die vom Formular gelieferten Parameter abgelegt.
<code>\$dbh</code>	Datenbank-Handle für die Verbindung zur Datenbank.
<code>\$sth</code>	Statement-Handle für die jeweils aktive SQL-Anweisung.
<code>\$abt_id</code>	ist die aus der Datenbank ermittelte Abteilungs-Id.

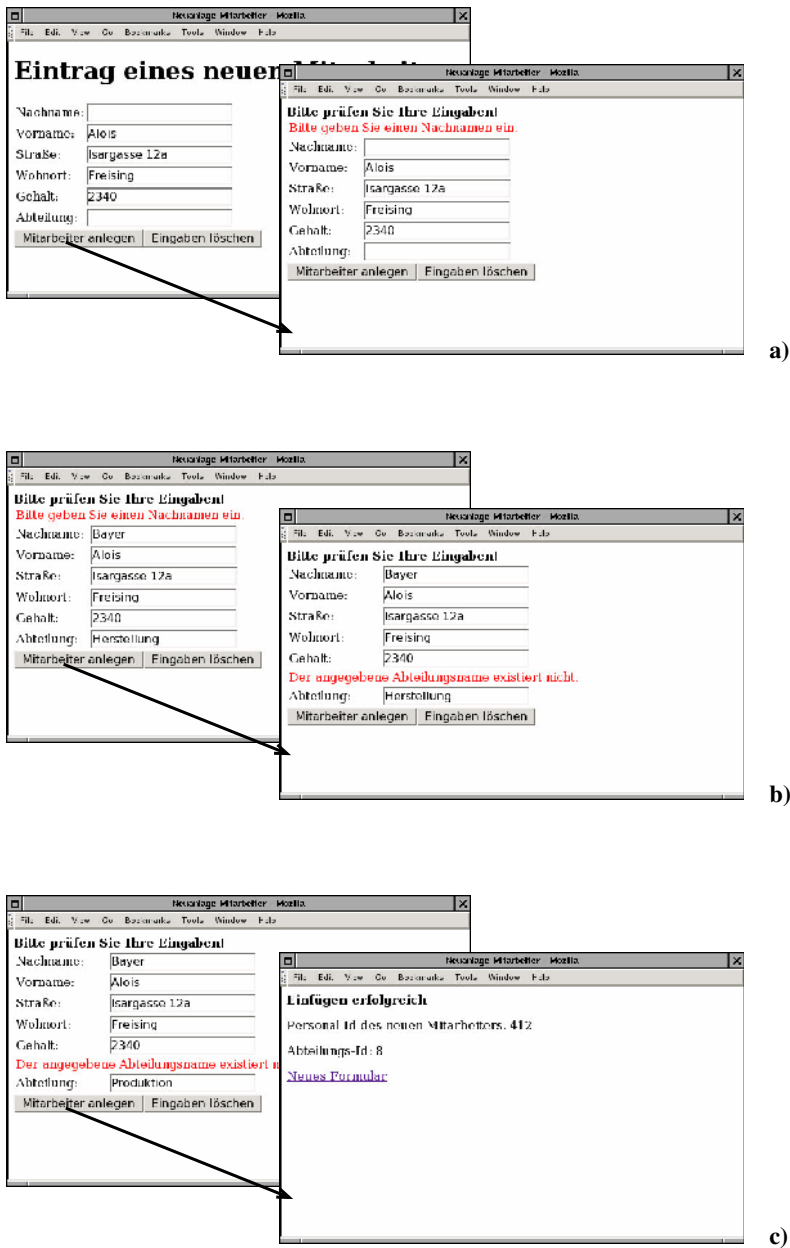


Abb. 104: Ablauf der Korrektur einer fehlerbehafteten Eingabe

```

1  #! /usr/bin/perl -wT
2  use strict;
3  use CGI qw(:standard *table);
4  use CGI::Carp qw(fatalsToBrowser);
5  use DBI;
6
7  my ($i,$p,$dbh,$sth,$abt_id,$pers_id,$rows,@fields,%fields,%messages);
8
9  # Ausgabe Formular, ggf. mit Hinweisen
10 sub makeform {
11     my ($k, $v);
12     my @f = @fields;
13
14     if (%messages)
15     { print b("Bitte pr&uuml;fen Sie Ihre Eingaben!"); }
16     print start_form (), start_table();
17     while (@f) {
18         ($k,$v,@f) = @f;
19         if ($messages{$k}) {
20             print Tr(td({-colspan=>2}, font({-color=>"red"},
21                                     $messages{$k})));
22         }
23         print Tr(td("$v:"), td(textfield(-name => $k)));
24     }
25     print end_table(),
26         submit (-name => "aktion", -value => "Mitarbeiter anlegen"),
27         reset (-value => "Eingaben löschen"),
28         end_form;
29 }
30
31 # Festlegen der Formularfelder
32 @fields = ("nachname" => "Nachname",
33            "vorname"   => "Vorname",
34            "strasse"   => "Stra&szlig;e",
35            "ort"       => "Wohnort",
36            "gehalt"    => "Gehalt",
37            "abteilung" => "Abteilung");
38 %fields = @fields;
39
40 # Ausgabe Seitenanfang
41 print header, start_html ("Neuanlage Mitarbeiter");
42
43 # Pruefen, ob Erstaufruf des Formulars
44 if (!defined(param("aktion"))) {
45     print h1 ("Eintrag eines neuen Mitarbeiters");
46     makeform;
47     print end_html;
48     exit;
49 }
50
51 # Formulareingaben holen
52 foreach $i (keys(%fields)) {
53     $p{$i} = param($i);
54 }
55
56 # Kein Nachname eingegeben?
57 if (!defined($p{nachname}) || $p{nachname} =~ /^ *$/) {
58     $messages{nachname} = "Bitte geben Sie einen Nachnamen ein.";
59     makeform;
60     print end_html;
61     exit;
62 }

```

Abb. 105: Web-Formular mit Fehlerbehandlung (Anfang)

```

55 # mit Datenbank verbinden
56 $dbh = DBI->connect ("dbi:Pg:dbname=firma", "rank");
57 # Abteilungs-Id aus Name ermitteln
58 $sth = $dbh->prepare ("SELECT aid FROM abteilung
59                       WHERE bezeichnung = ?");
60 $sth->execute ($p{abteilung});
61 ($abt_id) = $sth->fetchrow_array ();
62 # Abteilungsname nicht vorhanden?
63 if (!defined($abt_id)) {
64     $dbh->disconnect();
65     $messages{abteilung} =
66         "Der angegebene Abteilungsname existiert nicht.";
67     makeform;
68     print end_html;
69     exit;
70 }
71 # hoechste Personalnummer ermitteln
72 $sth = $dbh->prepare ("SELECT MAX(pid) FROM personal");
73 $sth->execute ();
74 ($pers_id) = $sth->fetchrow_array ();
75 # neue Personalnummer berechnen
76 $pers_id++;
77 # Einfuegen des neuen Datensatzes
78 $sth = $dbh->prepare ("INSERT INTO personal
79                       (pid,nachname,vorname,strasse,ort,
80                       aid,gehalt,einstellung)
81                       VALUES (?, ?, ?, ?, ?, ?, ?, CURRENT_DATE)");
82 $rows = $sth->execute ($pers_id, $p{nachname}, $p{vorname},
83                       $p{strasse}, $p{ort}, $abt_id, $p{gehalt});
84 # Einfuegeresultat analysieren
85 if ($rows) {
86     print b("Einf&uuml;gen erfolgreich"), p;
87     print "Personal-Id des neuen Mitarbeiters. $pers_id", p;
88     print "Abteilungs-Id: $abt_id";
89 } else {
90     print b("Einf&uuml;gen fehlgeschlagen!"), p;
91     print "Bitte &uuml;berpr&uuml;fen Sie Ihre Eingaben ",
92           "bzw. Ihr System.";
93 }
94 # Datenbankverbindung abbauen
95 $dbh->disconnect();
96 # Link auf neue Formularinstanz erzeugen
97 print p, a({-href => url()}, "Neues Formular");
98 # HTML-Seite abschliessen
99 print end_html;

```

Abb. 106: Web-Formular mit Fehlerbehandlung (Fortsetzung)

<code>\$pers_id</code>	speichert die Personal-Id des neuen Mitarbeiters.
<code>\$rows</code>	speichert die Anzahl der Änderungen beim Einfügen des neuen Mitarbeiters und wird für die Fehlerkontrolle verwendet.
<code>@fields</code>	In diesem Array werden die Namen und die Bezeichnungen der Formularfelder abgelegt, und zwar in der Reihenfolge <i>feldname1</i> , <i>feldbezeichnung1</i> , <i>feldname2</i> , <i>feldbezeichnung2</i> , ...
<code>%fields</code>	Dieser Hash wird aus dem Array <code>@fields</code> erzeugt und enthält die gleichen Informationen, wobei der Feldname der Schlüssel ist, so dass $\text{\$fields}\{\textit{feldname}\} = \textit{feldbezeichnung}$ gilt. Wir benötigen diese Informationen sowohl als Array als auch als Hash, weil man zwar auf die Einträge eines Hash einfacher zugreifen kann, aber andererseits ein Hash die Elemente nicht in einer garantierten Reihenfolge ablegt. Wir benötigen aber eine definierte Reihenfolge, damit nicht überraschenderweise bei Anzeige des Formulars beispielsweise das Feld "Straße" vor dem Feld "Nachname" steht.
<code>%messages</code>	Dieser Hash wird verwendet, um feldbezogene Fehlermeldungen abzulegen, die beim Erzeugen des Formulars vor dem betreffenden Feld angezeigt werden sollen.

Das Erzeugen des Formulars geschieht in einem Unterprogramm `makeform` in den Zeilen 8–26. Die in Zeile 9 definierten lokalen Variablen `$k` und `$v` werden jeweils zum Ablegen eines Feldnamens und der zugehörigen Feldbezeichnung verwendet. In Zeile 10 wird eine lokale Kopie des Arrays `@fields` in der Variable `@f` angelegt, da das Erzeugen des Formulars das Array mit den Feldnamen und -bezeichnungen zerstört.

Die Zeilen 11 und 12 dienen dazu, eine Information über aufgetretene Fehler auszugeben. Dies geschieht nur dann, wenn der Hash `%messages` Einträge enthält. Die Information soll im Fettdruck angezeigt werden, daher wird sie als Argument der vom CGI-Modul zur Verfügung gestellten Funktion `b` verwendet. Damit werden die für den Fettdruck benötigten HTML-Tags erzeugt.

In Zeile 13 werden die für den Beginn des Formulars und der Tabelle mit den Eingabefeldern benötigten Tags erzeugt. Überraschenderweise besitzt `start_form` kein Argument. Damit wird erreicht, dass als Ak-

tionsskript wieder das gleiche Skript aufgerufen wird – wir brauchen hier also keinen Skriptnamen anzugeben.

In den Zeilen 14–21 werden aus dem Array `@f` die Formularfelder mit Bezeichnungen erzeugt. Die Zeile 15 dient hierbei dazu, jeweils die ersten beiden Elemente aus dem Array abzuspalten. `$k` enthält dann den Namen des aktuellen Formularfeldes, `$v` die Bezeichnung. Als nächstes wird in Zeile 16 geprüft, ob für das aktuelle Formularfeld eine Fehlermeldung vorliegt. Falls ja, wird diese in den Zeilen 17 und 18 ausgegeben. Die etwas kryptische Form der Ausgabe dient dazu, HTML-Tags zu erzeugen, die einerseits dazu dienen, die Fehlermeldung über beide Tabellenspalten (Formularbezeichnung und Formularfeld) gehen zu lassen und somit das Tabellenformat nicht zu zerstören; zum anderen soll die Fehlermeldung vom Browser in roter Schrift angezeigt werden.

In den Zeilen 22–25 wird das Endetag für die Tabelle erzeugt. Weiterhin werden der Submit- und der Reset-Button generiert und das Endetag für das Formular ausgegeben.

Das Hauptprogramm beginnt in Zeile 27. In Zeile 28ff. werden die Namen und Bezeichnungen der Formularfelder definiert, indem sie in das Array `@fields` eingetragen werden. Die hier verwendete Notation `=>` dient zur Unterstreichung der Zusammengehörigkeit der links und rechts von `=>` stehenden Konstrukte für den Leser des Codes. Für Perl ist `=>` gleichbedeutend mit einem gewöhnlichen Komma. Zeile 34 erzeugt aus dem Array den Hash.

Der Seitenanfang (Zeile 36) soll immer gleich sein, egal, ob das Skript zum ersten Mal oder als Aktion eines abgeschickten Formulars aufgerufen wird: Die Start-Tags für die HTML-Seite werden erzeugt, der Text “Neuanlage Mitarbeiter” soll in der Kopfzeile des Browsers angezeigt werden.

Der weitere Ablauf des Skripts bestimmt sich nach dem Umstand, ob es zum ersten Mal oder als Aktion eines abgeschickten Formulars ausgeführt wird. Beim erstmaligen Aufruf ist lediglich das Formular zu erzeugen – weitere Aktionen sind nicht erforderlich, denn es liegen ja noch keine Benutzereingaben vor. Wie erkennt man aber nun, ob das Skript durch ein abgeschicktes Formular aufgerufen wurde? Man macht sich hier die Tatsache zunutze, dass auch der Submit-Button als Formularfeld angesehen wird, dessen Name und Wert als Parameter an das Skript übermittelt wird.

Wir prüfen daher in Zeile 38 einfach, ob der Parameter “aktion” – das ist der Name des Submit-Buttons in unserem Formular – an das Skript übergeben wurde. Wenn das nicht der Fall ist, muss es sich um den ersten Aufruf unseres Skripts handeln. Wir erzeugen dann eine Überschrift (Zeile 39), generieren das Formular durch Aufruf unseres Unterprogramms (Zeile 40), geben das Endetag für die Seite aus (Zeile 41) und beenden in Zeile 42 das CGI-Skript.

Ab Zeile 44 wissen wir, dass wir Formulareingaben zu verarbeiten haben. Wir übertragen in den Zeilen 45–47 die Eingaben in den Hash %p. In Zeile 49 prüfen wir, ob überhaupt ein Nachname eingegeben wurde. Dazu stellen wir sicher, dass der Parameter überhaupt vorhanden ist und dass sein Wert nicht nur aus lauter Leerzeichen besteht. Wenn kein Nachname eingegeben wurde, setzen wir in Zeile 50 die im Formular anzuzeigende Fehlermeldung, erzeugen dann das Formular (Zeile 51), geben das Endetag für die Seite aus (Zeile 52) und beenden in Zeile 53 das CGI-Skript.

Für die Prüfung, ob der eingegebene Abteilungsname gültig ist, benötigen wir eine Datenbankabfrage. Wir stellen daher eine Verbindung zur Datenbank her (Zeile 56) und versuchen, die Abteilungs-Id aus dem eingegebenen Abteilungsnamen zu ermitteln (Zeilen 58–61). Gelingt dies nicht, wird analog zur Fehlerbehandlung beim Nachnamen die anzuzeigende Fehlermeldung gesetzt, das Formular und das Endetag ausgegeben und das CGI-Skript beendet (Zeilen 63–70).

Man beachte, dass in Zeile 64 explizit die Datenbankverbindung abgebaut wird. Das geschieht zwar überlicherweise ohnehin, wenn das Skript beendet wird, aber in Umgebungen, wo CGI-Skripten von einer ständig laufenden Instanz des Perl-Interpreters ausgeführt werden, muss dies nicht notwendigerweise so sein. Man sollte also sichergehen, dass die Verbindung zur Datenbank beendet wird, indem man sie stets explizit vor Beendigung des Skripts abbaut.

Die Zeilen 72–83 sind in der gleichen Form schon im Skript aus Abb. 100 aufgetreten. Hier wird die neue Personalnummer ermittelt und der neue Mitarbeiter eingetragen. In Zeile 85ff. prüfen wir, ob das Einfügen erfolgreich war und geben dem Benutzer ein dementsprechendes Feedback. Konnte der Mitarbeiter eingefügt werden, teilen wir dem Benutzer die Personal-Id und die Abteilungs-Id mit (Zeilen 86–88), ansonsten erhält der Benutzer eine allgemeine Fehlermeldung (Zeilen 90–92). (In einer Produktivversion sollte man eine ausführlichere Fehleranalyse vornehmen. Aus Gründen der Kompaktheit des Codes haben

wir hier darauf verzichtet.) Die Funktion `p`, die in den Zeilen 86, 87 und 90 aufgerufen wird, kommt aus dem CGI-Modul und erzeugt ein HTML-Tag für den Beginn eines neuen Absatzes.

Der Rest des Skripts baut die Verbindung zur Datenbank ab (Zeile 95) und erzeugt einen HTML-Link, den der Benutzer anklicken kann, um das Formular nochmals neu aufzurufen (Zeile 97), damit ein weiterer Mitarbeiter eingegeben werden kann. Der hier verwendete Funktionsaufruf `url()` liefert die WWW-Adresse unseres CGI-Skripts. Klickt der Benutzer auf den Link, wird das Skript ohne Parameter aufgerufen, es wird also ein völlig neues Formular erzeugt. In Zeile 99 wird schließlich die HTML-Seite abgeschlossen.

8.8. Effizienz von CGI-Skripten

Der Aspekt der Effizienz von CGI-Skripten kann unter verschiedenen Gesichtspunkten betrachtet werden:

- Geschwindigkeit: Die Zeit, die ein CGI-Skript zur Abarbeitung benötigt, bestimmt auch die Zeit, bis der Benutzer im WWW-Browser das Resultat (die Ausgabe des Skripts) angezeigt bekommt.
- Ressourcennutzung: Wie stark belastet die Ausführung eines CGI-Skripts den WWW-Server? Dies ist ein wichtiges Kriterium, da ja mehrere Benutzer gleichzeitig auf das Skript zugreifen können (siehe dazu auch den nächsten Abschnitt).

Offensichtlich hängen diese beiden Gesichtspunkte eng zusammen, denn je höher der Ressourcenverbrauch von CGI-Skripten, umso länger die Zeitdauer der Ausführung im Mehrbenutzerbetrieb. Natürlich ist die Geschwindigkeit wesentlich von der Komplexität der durchgeführten Aktionen abhängig, aber es gibt einen gewissen Overhead, der jedesmal bei Ausführung eines Skripts anfällt:

- Der Perl-Interpreter muss gestartet werden und das Skript in einen Zwischencode übersetzt werden. Dies benötigt eine gewisse Zeit.
- Die Verbindung zur Datenbank muss hergestellt und auch wieder abgebaut werden. Dies bedeutet insbesondere, dass eine Datenbank-Transaktion den Ablauf eines CGI-Skripts nicht überleben kann, da Transaktionen spätestens mit Abbau der Datenbankverbindung beendet werden.

Das erste Problem des fortwährenden Aufrufs des Perl-Interpreters kann in entsprechenden Systemumgebungen dadurch gelöst werden, dass bereits vom WWW-Server ein Perl-Interpreter gestartet wird und

ständig läuft.³¹ Dieser Interpreter bekommt dann CGI-Skripten vom WWW-Server zur Ausführung übergeben. Da der Interpreter nach Ausführung eines Skripts nicht beendet wird, bleibt auch der erzeugte Zwischencode erhalten, so dass bei der nächsten Ausführung des gleichen Skripts die Compilierungsphase entfällt.

Systemumgebungen wie eben beschrieben können um sog. **persistente** Datenbankverbindungen erweitert werden.³² Hier kann eine Datenbankverbindung auch nach Beendigung eines Skripts bestehen bleiben und muss beim erneuten Aufruf des Skripts nicht wieder neu aufgebaut werden. Damit wäre auch das zweite Problem des Overheads für den Verbindungsauf- und -wiederabbau bei Datenbanken gelöst.

Allerdings können sich auch damit Transaktionen nicht über mehrere Skripten bzw. Skriptaufrufe erstrecken.³³ Der Grund dafür liegt darin, dass üblicherweise auf einem WWW-Server nicht nur ein Serverprozess abläuft, sondern mehrere, damit nicht nur eine Anforderung (engl. Request) gleichzeitig bearbeitet werden kann (Abb. 107). Die Zuordnung der eingehenden Requests zu "freien" Serverprozessen kann nicht in sinnvoller Weise beeinflusst werden.

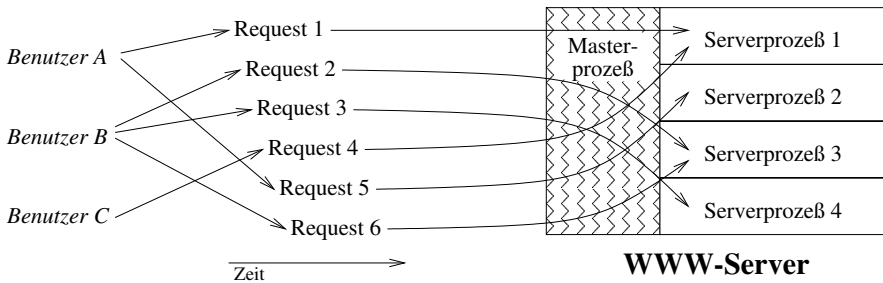


Abb. 107: Bearbeitung von Requests durch vier WWW-Serverprozesse

Im Beispiel von Abb. 107 setzt Benutzer A zwei Requests ab, die von den Serverprozessen Nr. 1 und Nr. 2 bearbeitet werden. Da je-

³¹ Dies ist z. B. mit dem Apache-Webserver in Verbindung mit der Servererweiterung `mod_perl` möglich. Informationen hierzu finden sich unter <http://perl.apache.org/>.

³² In Verbindung mit dem Apache-Webserver leistet dies das Perl-Modul `Apache::DBI`.

³³ Dies gilt zumindest für das `Apache::DBI`-Modul.

der Serverprozess einen eigenen Speicherbereich für Variablen besitzt, ist z. B. von Prozess 2 kein Zugriff auf die Variablen von Prozess 1 möglich. Technisch wäre es zwar durchaus denkbar, einen gemeinsamen Speicherbereich für gemeinsam genutzte Variablen festzulegen, jedoch müsste hier beim Zugriff der gegenseitige Ausschluss sichergestellt werden, was die Skriptprogrammierung um ein Vielfaches komplizierter machen würde.

Es gibt jedoch noch eine weitere Schwierigkeit, die die Verteilung einer Transaktion über mehrere CGI-Skripten (bzw. mehrere Aufrufe des gleichen Skripts) in Frage stellt. Betrachten wir das Beispiel in Abb. 108, wo der Benutzer zwei Formulare hintereinander ausfüllen muss. Mit dem Abschicken des ersten Formulars wird eine Transaktion gestartet und erst mit dem Abschicken des zweiten Formulars wieder beendet.

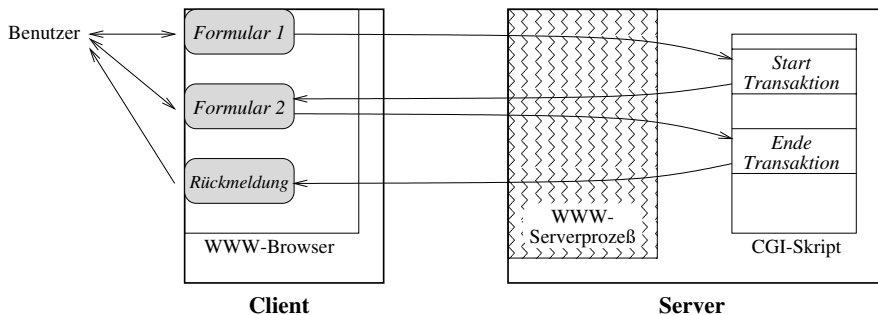


Abb. 108: Über mehrere Formulare verteilte Transaktion

Schickt nun der Benutzer das zweite Formular nicht mehr ab, bleibt die Transaktion offen und hindert möglicherweise andere Benutzer daran, auf die Datenbank zuzugreifen (Abb. 109). Dieses Problem könnte durch einen separaten "Überwacherprozess" gelöst werden, der Transaktionen, die eine gewisse Zeitspanne offen sind, mit einem Rollback beendet. Jedoch ist damit ein nicht unerheblicher Implementierungsaufwand verbunden.

Generell (nicht nur bei der WWW-Anbindung von Datenbanken) stellt es eine zweifelhafte Strategie dar, den Abschluss einer Transaktion vom Warten auf eine Benutzereingabe abhängig zu machen. Die Länge einer Transaktion sollte auch aus Performanzgründen auf das notwendige Mindestmaß beschränkt werden.

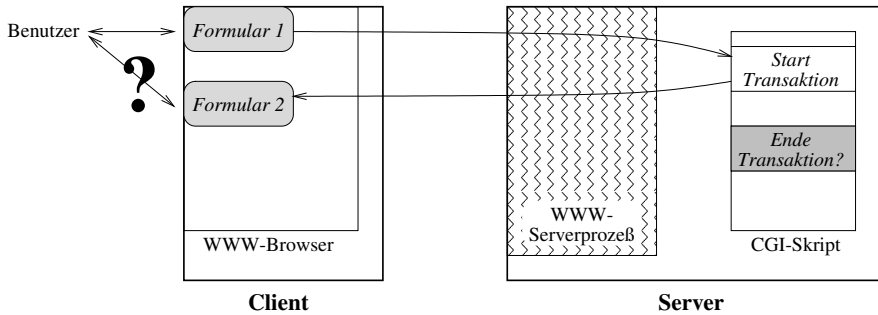


Abb. 109: Nicht komplettierte Transaktion

8.9. CGI-Skripten im Mehrbenutzerbetrieb

Betrachten wir noch einmal das CGI-Skript in Abb. 105 und Abb. 106. Dieses Skript funktioniert perfekt, solange es nur jeweils von einem Benutzer gleichzeitig benutzt wird. Wie stellt sich die Situation dar, wenn mehrere Benutzer gleichzeitig einen neuen Mitarbeiter eintragen wollen?

Ein kritischer Fall ergibt sich genau dann, wenn das durch einen Benutzer 1 gestartete CGI-Skript die höchste momentan vorhandene Personalnummer ermittelt hat und das durch den Benutzer 2 etwas später gestartete CGI-Skript (genauer: die neue Instanz des gleichen CGI-Skripts) dies ebenfalls tut, bevor das CGI-Skript von Benutzer 1 den neuen Datensatz eintragen (und die Transaktion beenden) konnte (Abb. 110).

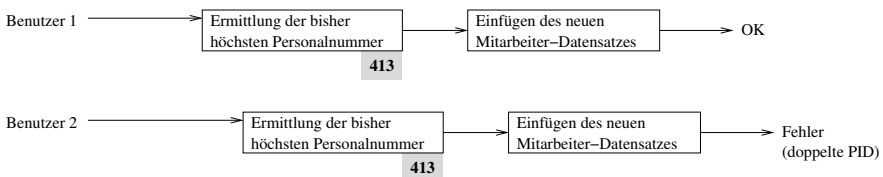


Abb. 110: Fast gleichzeitiger Zugriff auf ein CGI-Skript durch zwei Benutzer

Hier würde also dann versucht, zwei Datensätze mit der gleichen PID anzulegen. Das Attribut PID sollte aber sinnvollerweise ein Primärschlüssel der Tabelle PERSONAL sein. Wenn wir die Überwachung dieser Primärschlüsseigenschaft durch eine Integritätsbe-

dingung sichergestellt haben, wird die langsamere Instanz des Skripts einen Fehler produzieren.³⁴

Dieses Verhalten fällt in die Kategorie einer sog. **race condition**. Nur der Gewinner des “Rennens” kann die Transaktion ordnungsgemäß beenden. Race conditions stellen eine hartnäckige Fehlerquelle in Systemen dar, da sie im allgemeinen recht selten auftreten und daher nur schwer durch Tests entdeckt werden können.

In unserem Beispiel erstreckt sich der “kritische Bereich” des Skripts auf die Zeilen 73 bis 83 (der grau unterlegte Bereich in Abb. 106 auf S. 181). Hier wird die in Abb. 110 gezeigte Situation nur sehr selten auftreten, da von der Ermittlung der höchsten Personalnummer bis zum Einfügen des neuen Datensatzes (und anschließender Beendigung der Transaktion) nur sehr wenig Zeit vergeht. Wir können diese race condition aber künstlich simulieren, indem wir in den Skriptcode zwischen die Zeilen 76 und 77 die Anweisung

```
sleep(5);
```

einfügen. Damit legt das Skript an dieser Stelle eine Pause von 5 Sekunden ein, womit der kritische Bereich eine sehr viel größere Zeitspanne zur Abarbeitung benötigen wird.

Bedienen nun zwei Benutzer dieses Formular und Benutzer 1 schickt das Formular zur Zeit $t + 0s$ ab, während Benutzer 2 das Formular 2 Sekunden später, also zur Zeit $t + 2s$ abschickt, haben wir genau die Situation künstlich herbeigeführt, dass sich Instanz 1 des Skripts noch im kritischen Bereich befindet, während Instanz 2 ebenfalls den kritischen Bereich betritt. Da wir die Verzögerung jeweils konstant mit 5 Sekunden Dauer herbeiführen, wird immer Instanz 1 der Gewinner des “Rennens” sein (Abb. 111).

Um solche Situationen zu vermeiden, muss sichergestellt werden, dass jeweils nur eine Instanz des Skripts gleichzeitig den kritischen Bereich betritt. Dies lässt sich im Beispiel durch geeignete Sperren leicht erreichen: Zu Beginn des kritischen Bereichs muss auf die Tabelle PERSONAL eine Exklusiv-Sperre gesetzt werden (vgl. hierzu Abschnitt 5.2).

³⁴ Dieses Problem betrifft natürlich nicht nur die CGI-Version unseres Mitarbeitereinfügeprogramms. Der gleiche Fehler kann auch bei den kommandozeilenorientierten Programmversionen in Abb. 86 und Abb. 90 auftreten. Bei diesen Versionen steht allerdings der Mehrbenutzeraspekt weniger im Vordergrund, da Kommandozeilenprogramme hauptsächlich für administrative Zwecke eingesetzt, also im allgemeinen höchstens von einer Person gleichzeitig genutzt werden.

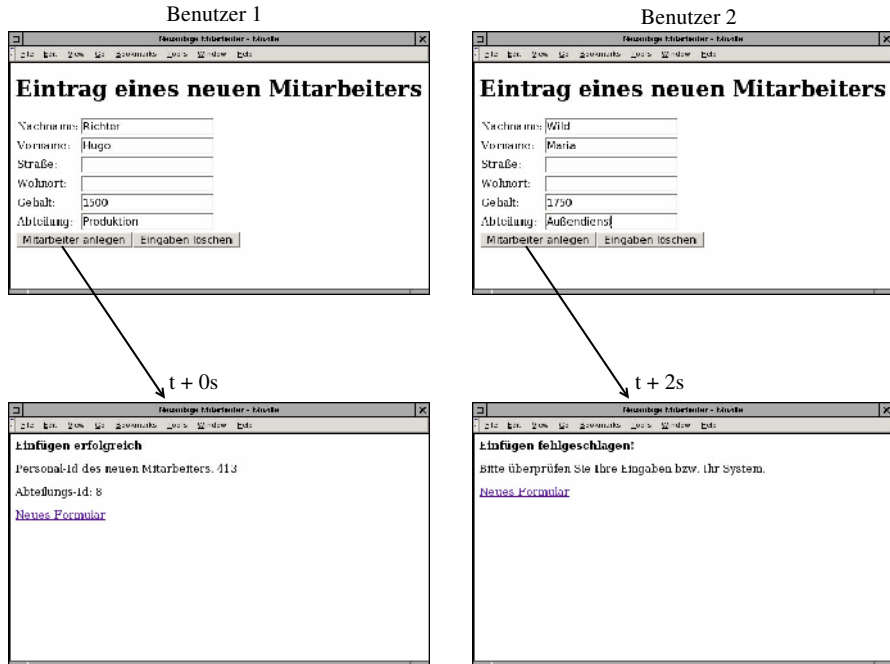


Abb. 111: Konsequenzen einer “race condition”

Bei Beendigung der Transaktion nach Zeile 83 – hier wäre dann entsprechend ein Aufruf der `commit`-Methode einzufügen – wird die Sperre automatisch gelöscht.

Damit dies in DBI funktioniert, muss beim Verbinden zur Datenbank die `AutoCommit`-Funktionalität von DBI abgeschaltet werden. Die Zeile 56 lautet dann

```
$dbh = DBI->connect ("dbi:Pg:dbname=firma",
    "rank", "", { AutoCommit => 0 });
```

Wir zeigen in Abb. 112 den kompletten Programmcode, der den Code von Abb. 106 in einer Mehrbenutzerumgebung ersetzen sollte. Die geänderten bzw. eingefügten Zeilen sind grau hinterlegt. Die Zeile 77 mit der `sleep`-Anweisung sollte im Produktionsbetrieb natürlich entfernt werden.

In dieser Version funktioniert das CGI-Skript auch, wenn viele Benutzer gleichzeitig darauf zugreifen. Wie beim Umgang mit Sperren üblich, werden konkurrierende Transaktionen, die auf eine gesperrte Tabelle zugreifen wollen, so lange angehalten, bis die Sperre wieder

```

55 # mit Datenbank verbinden
56 $dbh = DBI->connect ("dbi:Pg:dbname=firma", "rank", "",
57                      { AutoCommit => 0 });

58 # Abteilungs-Id aus Name ermitteln
59 $sth = $dbh->prepare ("SELECT aid FROM abteilung
60                      WHERE bezeichnung = ?");
61 $sth->execute ($p{abteilung});
62 ($abt_id) = $sth->fetchrow_array ();

63 # Abteilungsname nicht vorhanden?
64 if (!defined($abt_id)) {
65     $dbh->disconnect();
66     $messages{abteilung} =
67         "Der angegebene Abteilungsname existiert nicht.";
68     makeform;
69     print end_html;
70     exit;
71 }

72 # hoechste Personalnummer ermitteln
73 $sth = $dbh->prepare ("SELECT MAX(pid) FROM personal");
74 $dbh->do ("LOCK TABLE personal IN EXCLUSIVE MODE");
75 $sth->execute ();
76 ($pers_id) = $sth->fetchrow_array ();
77 sleep(5);

78 # neue Personalnummer berechnen
79 $pers_id++;

80 # Einfuegen des neuen Datensatzes
81 $sth = $dbh->prepare ("INSERT INTO personal
82                      (pid,nachname,vorname,strasse,ort,
83                      aid,gehalt,einstellung)
84                      VALUES (?, ?, ?, ?, ?, ?, ?, CURRENT_DATE)");
85 $rows = $sth->execute ($pers_id, $p{nachname}, $p{vorname},
86                      $p{strasse}, $p{ort}, $abt_id, $p{gehalt});
87 $dbh->commit();

88 # Einfuegeresultat analysieren
89 if ($rows) {
90     print b("Einf&uuml;gen erfolgreich"), p;
91     print "Personal-Id des neuen Mitarbeiters. $pers_id", p;
92     print "Abteilungs-Id: $abt_id";
93 } else {
94     print b("Einf&uuml;gen fehlgeschlagen!"), p;
95     print "Bitte &uuml;berpr&uuml;fen Sie Ihre Eingaben ",
96         "bzw. Ihr System.";
97 }

98 # Datenbankverbindung abbauen
99 $dbh->disconnect();

100 # Link auf neue Formularinstanz erzeugen
101 print p, a({-href => url()}, "Neues Formular");

102 # HTML-Seite abschliessen
103 print end_html;

```

Abb. 112: "Mehrbenutzerfähiges"
Web-Formular mit Fehlerbehandlung

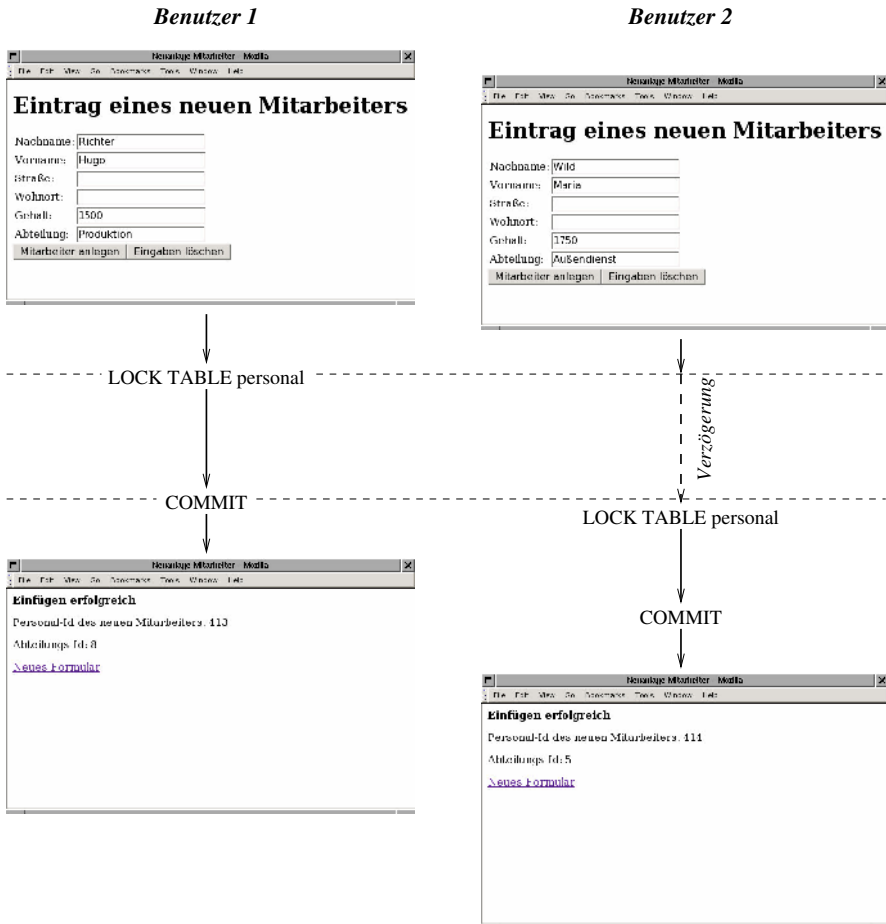


Abb. 113: Konkurrierende Transaktionen bei Web-Formularen

freigegeben wird. Dieses Verhalten ist für unser Beispielprogramm nochmals in Abb. 113 gezeigt.

8.10. Implementierung umfangreicher Anwendungen

Wir haben in den letzten Abschnitten beschrieben, wie man Datenbanken mit einer WWW-Oberfläche durch Verwendung von Open Source-Software versehen kann. Zur Illustration dieser Technik haben wir das verhältnismäßig einfache Beispiel des Eintragens eines neuen Mitarbeiters in unsere Musterdatenbank verwendet. Es stellt sich nun

die Frage, ob die dargestellte Realisierung in Perl und CGI auch für komplexere Anwendungen praktikabel ist.

Alternativen zur vorgestellten “Do it yourself”-Methode sind integrierte Entwicklungsumgebungen, die u. a. verschiedene vorgefertigte Oberflächenmodule bieten und die Codierungsarbeit angenehmer gestalten können. Allerdings ist zu bedenken, dass die Programmlogik der zu erstellenden Applikation in jedem Fall genau festgelegt werden muss – diesen wichtigen Schritt im Entwicklungsprozess kann einem keine noch so komfortable Entwicklungsumgebung abnehmen. Zudem handelt es sich bei integrierten Entwicklungstools zumeist um kommerzielle Softwareprodukte, die fast immer nur auf bestimmten Plattformen verfügbar sind.

Hingegen sichert das Open Source-Konzept (etwa der Sprache Perl und des Apache-Webserver) die Portierbarkeit einer Applikation auf verschiedene (Server-) Plattformen und gewährleistet die Unabhängigkeit von Softwareherstellern.

Letztendlich hängt die Entscheidung für eine bestimmte Methode zur WWW-Anbindung einer Datenbank auch von vielen anderen Faktoren ab, z. B.

- Integration in bestehende EDV-Landschaft,
- verfügbares Produktspektrum des Herstellers der eingesetzten Datenbank,
- Know-How der mit der Implementierung befassten Arbeitsgruppe,
- verfügbares Budget,
- geplanter Zeitrahmen für die Realisierung.

Im Rahmen dieses Buches kann daher keine allgemeingültige Empfehlung für eine bestimmte Methode zur WWW-Anbindung von Datenbanken gegeben werden.

9. Administrations- und Designwerkzeuge

Während wir im letzten Abschnitt einige Möglichkeiten zur Implementierung von Benutzeroberflächen für Endbenutzer vorgestellt haben, wollen wir jetzt entsprechende Oberflächen für den Datenbankadministrator bzw. -designer betrachten. Auch hier lässt sich zunächst eine Unterscheidung in Kommandozeilentools und graphische Oberflächen vornehmen; allerdings kommt es dabei nicht darauf an, die Datenbank vor dem Nutzer – der ja jetzt in der Rolle als Datenbankadministrator bzw. -designer auftritt – zu “verstecken”. Vielmehr sind die wünschenswerten Eigenschaften der in diesem Abschnitt beschriebenen Werkzeuge wie folgt zu charakterisieren:

- Vereinfachung von Routine-Administrationsaufgaben
- (Graphischer) Überblick über die relationale Datenbankstruktur mit der Möglichkeit, auch ohne SQL-Kenntnisse auf Detailinformationen zuzugreifen.
- Komfortable Erstellung und Ausführung von SQL-Abfragen
- Einfache Erzeugung von Datenbankobjekten
- Fehlersuch-Möglichkeit (“Debugging”) bei der Ausführung von Abfragen, Funktionen, Triggern

Konkrete Werkzeuge für Datenbankadministration bzw. -design sind meist spezifisch für ein bestimmtes DBMS, auch wenn die Funktionalität oft ähnlich ist. Wir stellen hier beispielhaft einige Werkzeuge vor, die speziell für PostgreSQL vorgesehen sind oder mit PostgreSQL genutzt werden können.

9.1. Kommandozeilentools

Naturgemäß wird ein reines Kommandozeilentool obigen Zielen nur sehr eingeschränkt gerecht werden – bei den meisten Datenbanksystemen ist ein solches Tool allerdings im Lieferumfang und bietet somit ohne die Installation zusätzlicher Software die Möglichkeit, auf eine Datenbank zuzugreifen. Kommandozeilentools ermöglichen hauptsächlich die interaktive Eingabe von SQL-Statements und geben das Resultat unmittelbar auf dem Bildschirm aus.

Bei PostgreSQL heißt das entsprechende Kommandozeilentool **psql**; eine Beispielsitzung unter Verwendung von **psql** ist in Abb. 114 zu sehen.

Obwohl der Komfort von Kommandozeilentools nicht besonders hoch ist, sind sie unentbehrlich

```

system% psql -U rank -d firma
Welcome to psql 7.4.2, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

firma=> select * from abteilung
firma-> where ort='München';
 aid | bezeichnung | ort
-----+-----+-----
   1 | Betriebsleitung | München
   5 | Außendienst   | München
(2 rows)

firma=> \d
          List of relations
Schema | Name      | Type  | Owner
-----+-----+-----+-----
public | abteilung | table | rank
public | personal  | table | rank
public | projekt   | table | rank
public | teile      | table | rank
public | zuordnung | table | rank
(5 rows)

firma=> \d personal
          Table "public.personal"
  Column      |      Type      | Modifiers
-----+-----+-----
 pid          | integer        | not null
 nachname     | character varying(20)
 vorname      | character varying(15)
 strasse      | character varying(30)
 ort          | character varying(20)
 einstellung  | date
 gehalt       | numeric(7,2)
 vorges_id    | integer
 aid          | integer
Indexes:
    "personal_pkey" primary key, btree (pid)

firma=> \q
system%

```

Anmelden an der Datenbank firma als Benutzer rank

Zeilenweises Eintippen einer SQL-Abfrage

Anzeige aller Tabellen/Views

Struktur der Tabelle PERSONAL

Beenden der Sitzung

Abb. 114: Eine psql-Beispielsitzung

- wenn kein grafikfähiges Terminal zur Verfügung steht, und
- wenn ein skriptgesteuerter Zugriff auf die Datenbank erfolgen soll und die in Abschnitt 7 beschriebenen Methoden nicht anwendbar sind.

9.2. Zugriff über WWW

Software, die einen administrativen Zugriff auf eine Datenbank über WWW gestattet, bringt folgende Vorteile:

- Für den Zugriff von einem vernetzten Client auf die Datenbank ist nur ein WWW-Browser erforderlich, man benötigt keine zusätzliche Software (auf dem Client).

- Plattformunabhängigkeit auf dem Client – es kann jeder vernetzte Client mit einem WWW-Browser verwendet werden.
- Graphikfähigkeit ist durch die Mittel von HTML gegeben.

Das Standardtool für den administrativen Zugriff auf eine PostgreSQL-Datenbank via WWW ist **phpPgAdmin**. Diese frei verfügbare Software ist in der Sprache PHP programmiert. PHP wird oft für die Realisierung dynamischer Webseiten (als Alternative zu CGI-Skripten) verwendet und steht auf Webserver-Installationen mit der Apache-Serversoftware (siehe Abschnitt B.5) in vielen Fällen bereits standardmäßig zur Verfügung. Die Struktur einer Datenbankinstallation betrachtet mit **phpPgAdmin** ist in Abb. 115 zu sehen. Der Webserver mit der **phpPgAdmin**-Software muss nicht identisch mit dem Datenbank-Server sein, was größere Flexibilität in einer vernetzten Serverumgebung bedeutet.

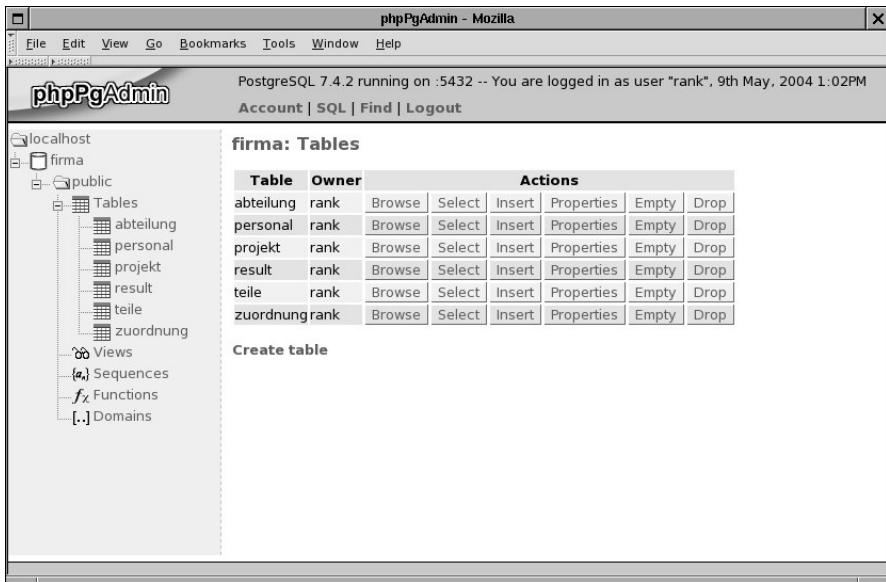


Abb. 115: Datenbankadministration mit **phpPgAdmin**

Mit **phpPgAdmin** kann man nicht nur die in Abschnitt 4.4 beschriebenen Datenbankdefinitions-Operationen vornehmen, sondern auch Datenbankabfragen ausführen, so dass **phpPgAdmin** im Prinzip ein vollwertiger Ersatz für das in Abschnitt 9.1 beschriebene Kommandozeilentool **psql** ist.

Über ein sog. *SQL-Worksheet* kann man beliebige SQL-Statements auf der Datenbank ausführen und erhält die Ergebnisse in übersichtlicher Form angezeigt. In Abb. 116 ist das SQL-Worksheet mit der auf S. 48 gezeigten Abfrage und das Ergebnis dargestellt.

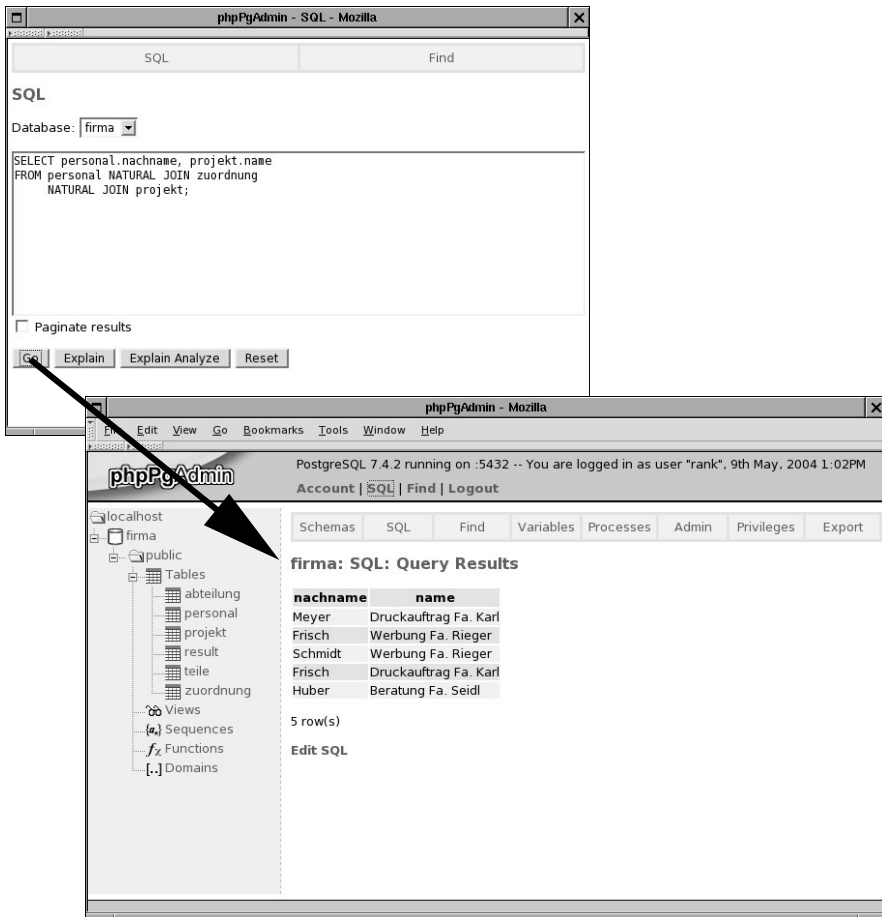


Abb. 116: Freie SQL-Statements in phpPgAdmin

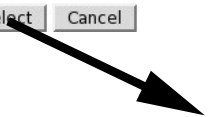
Einfache SELECT-Abfragen, die sich nur auf eine Tabelle beziehen und nur Auswahl- und Projektionsoperationen beinhalten, kann man einfach nach dem "Query by example"-Prinzip ohne SQL-Kenntnisse wie in Abb. 117 gezeigt ausführen.

firma: Tables: personal: Select

Show	Field	Type	Operator	Value
<input type="checkbox"/>	pid	integer	=	
<input checked="" type="checkbox"/>	nachname	character varying(20)	=	
<input checked="" type="checkbox"/>	vorname	character varying(15)	=	
<input type="checkbox"/>	strasse	character varying(30)	=	
<input type="checkbox"/>	ort	character varying(20)	=	
<input type="checkbox"/>	einstellung	date	=	
<input type="checkbox"/>	gehalt	numeric(7,2)	=	
<input type="checkbox"/>	vorges_id	integer	=	
<input type="checkbox"/>	aid	integer	=	5

☐ Select all fields

SelectCancel



firma: Tables: personal: Select

nachname	vorname
Huber	Hilde
Klement	Karl

2 row(s)

[Back](#) | [Edit SQL](#) | [Expand](#) | [Create view](#) | [Download](#) | [Refresh](#)

Abb. 117: Datenbankabfrage nach dem “Query by example”-Prinzip

9.3. Dedizierte graphische Tools

Im Unterschied zu den via WWW realisierten Administrationstools handelt es sich hier um eigenständige Anwendungen, die ggf. nur für bestimmte Plattformen erhältlich sind.

Wir stellen hier das Tool **pgAdmin III** vor, das kostenlos für unix-artige Betriebssysteme und Windows zur Verfügung steht. Unsere Musterdatenbank **FIRMA** ist in **pgAdmin III** wie in Abb. 118 abgebildet dargestellt.

pgAdmin III erlaubt – neben den “üblichen” Administrationsoperationen, wie sie etwa auch **phpPgAdmin** anbietet – speziell auch den Zugriff auf für das Datenbankdesign wichtige Objekte. Wir zeigen hier beispielhaft die Definition des Triggers aus Abb. 84 auf S. 141 mit **pgAdmin III**.

Zur Definition der Funktion, die vom Trigger aufgerufen werden soll,

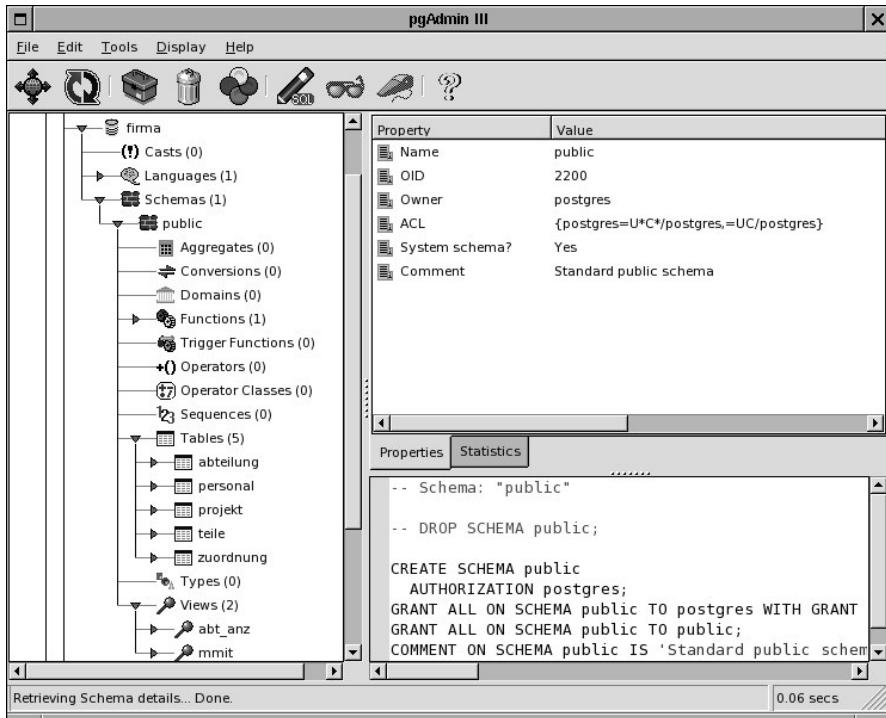


Abb. 118: Musterdatenbank FIRMA in pgAdmin III

klickt man bei “Trigger Functions” mit der rechten Maustaste und wählt “New Object” und “New Trigger Function”. Bei dem folgenden Dialog füllt man in der Karteikarte “Properties” die Felder “Name” und “Language” aus; in der Karteikarte “Parameter” schreibt man unter “Definition” eben den Code, der die Funktion definiert. Die entsprechenden Dialoge sind in Abb. 119 dargestellt.

Den Trigger auf der Tabelle PROJEKT definiert man, indem man den Punkt “Triggers” mit der rechten Maustaste anklickt und “New Object” und “New Trigger” wählt. In dem folgenden Dialog trägt man bei “Name” den Namen des Triggers ein. Da es sich um einen Zeilentrigger handelt, ist “Row trigger” zu markieren. Die “Trigger function” kann man aus dem Drop-down-Menu auswählen, bei “Fires” wählt man “AFTER” und bei “Events” “Update”. Durch Klicken auf “OK” wird die Triggerdefinition durchgeführt. Die Dialoge sind in Abb. 120 nochmals gezeigt.

Man kann aus diesem Beispiel deutlich erkennen, dass das Design ei-

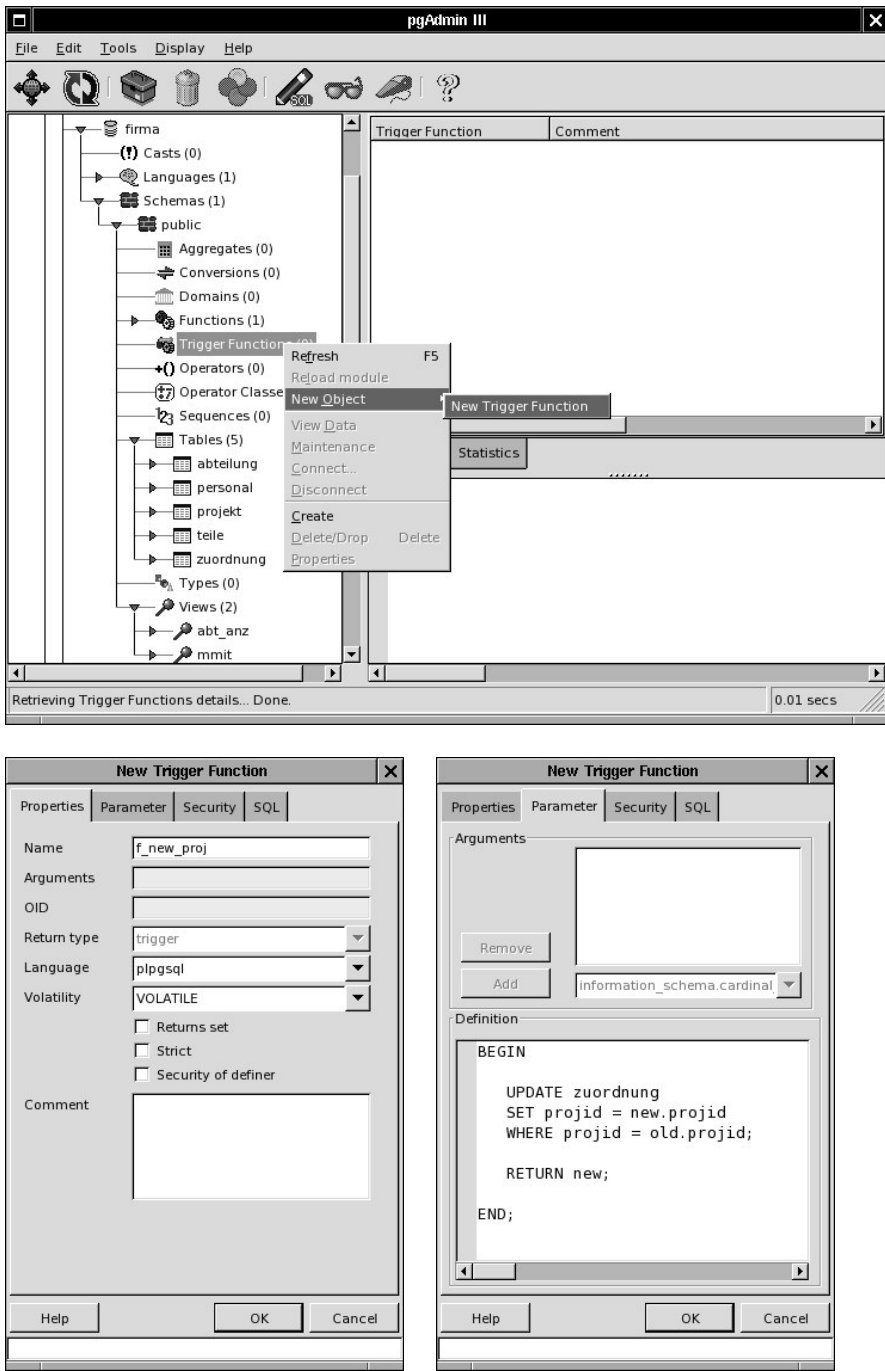


Abb. 119: Definition einer Triggerfunktion in pgAdmin III

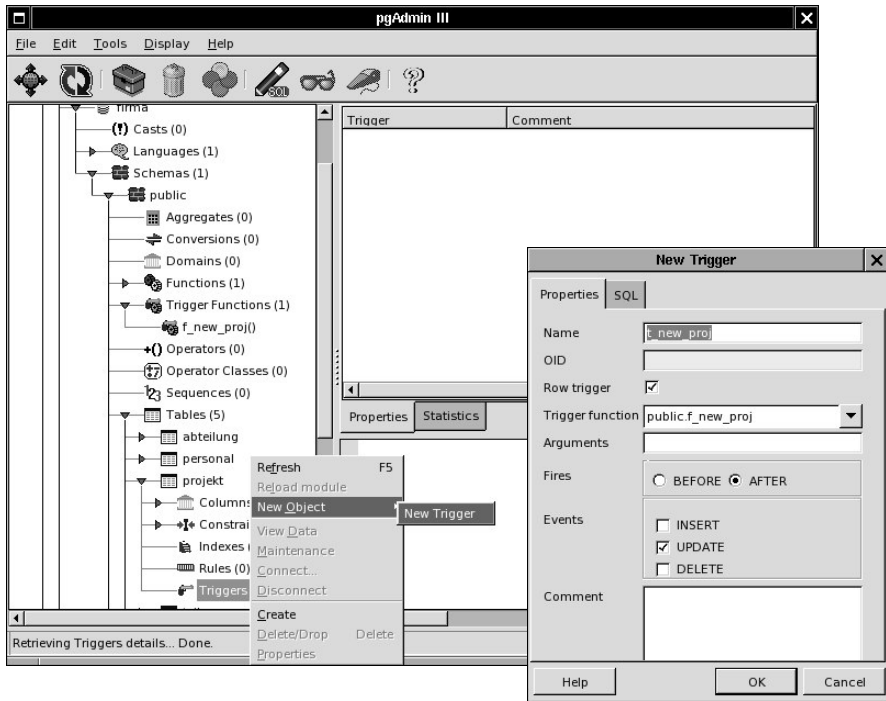


Abb. 120: Definition eines Triggers in pgAdmin III

ner Datenbank insoweit vereinfacht wird, als man die entsprechenden Definitionen von Objekten nicht in SQL formulieren muss, sondern interaktiv mit der Maus vornehmen kann. pgAdmin III erzeugt dann im Hintergrund ein korrektes SQL-Statement und schickt es an PostgreSQL.

Da sich die Bedienung von pgAdmin III beim praktischen Arbeiten damit leicht erschließt, wollen wir die Vorstellung dieses Tools im Rahmen dieses Buches nicht weiter vertiefen, sondern den Leser einladen, es selbst auszuprobieren.




Literatur

- [Co1970] E. F. Codd: A Relational Model of Data for Large Shared Data Banks; in: Communications of the ACM 13,6, Juni 1976, 377-387
- [Da2004] C. J. Date: An Introduction to Database Systems, 8th Edition; Addison-Wesley 2004
- [DeBu2000] A. Descartes, T. Bunce: Programming the Perl DBI; O'Reilly & Associates Inc. 2000
- [EiMe1999] A. Eisenberg, J. Melton: SQL:1999, formerly known as SQL3; in: SIGMOD Record 28,1, März 1999, 131-138
- [ElNa2004] R. Elmasri, S. B. Navathe: Fundamentals of Database Systems, 4nd Edition; Addison-Wesley 2004
- [GuGuBi2000] S. Guelich, S. Gundavaram, G. Birznies: CGI Programming with Perl, 2nd Edition; O'Reilly & Associates Inc. 2000
- [KeEi2004] A. Kemper, A. Eickler: Datenbanksysteme – eine Einführung, 5. Auflage; Oldenbourg 2004
- [MeSi1993] J. Melton, A. R. Simon: Understanding the New SQL - A Complete Guide; Morgan Kaufmann Publ. 1993
- [MuKe2002] C. Musciano, B. Kennedy: HTML & XHTML – The Definitive Guide, 5th Edition; O'Reilly & Associates Inc. 2002
- [Op1999] Open Source: kurz & gut; O'Reilly & Associates Inc. 1999, online unter:
http://www.ora.de/german/freebooks/os_fb
- [Pg2004] PostgreSQL 7.4.2 Documentation; The PostgreSQL Global Development Group, online unter:
<http://www.postgresql.org/docs>
- [ScCh2001] R. L. Schwartz, T. Phoenix: Learning Perl, 3nd Edition; O'Reilly & Associates Inc. 2001
- [SQL1999] Information Technology – Database Languages – SQL; ISO/IEC 9075- x -1999, wobei x die Nr. des Teildokuments (1,2,3,4,5) bezeichnet.
- [SQL2003] Information technology – Database languages – SQL; ISO/IEC 9075- x :2003, wobei x die Nr. des Teildokuments (1,2,3,4,9, 10,11,13,14) bezeichnet.

A. Syntaxdiagramme

Syntaxdiagramme dienen dazu, die richtige Schreibweise (Syntax) von Sprachkonstrukten anschaulich zu definieren.

Im Prinzip handelt es sich bei einem Syntaxdiagramm um einen We-geplan mit einem Ausgangs- und einem Endpunkt. Jeder Weg vom Ausgangs- zum Endpunkt beschreibt ein syntaktisch zulässiges Sprachkonstrukt. Elemente von Syntaxdiagrammen sind Pfeile, abgerundete Kästen bzw. Kreise und Rechtecke.

Abgerundete Kästen  enthalten ein syntaktisches Element, das zeichenweise genau so angegeben werden muss. Kreise  enthalten jeweils ein Sonderzeichen, das genau so angegeben werden muss. Rechtecke  kennzeichnen Namen von syntaktischen Elementen, die wiederum über ein weiteres Syntaxdiagramm (oder auch manchmal in Worten) definiert sind. Dieses müsste an dieser Stelle für das Rechteck substituiert werden. Pfeile \longrightarrow kennzeichnen zulässige Wege. Sie dürfen jeweils nur in Pfeilrichtung durchlaufen werden.

Betrachten wir beispielsweise das Syntaxdiagramm, das die syntaktische Einheit ‘Ziffer’ charakterisieren soll (Abb. 121).

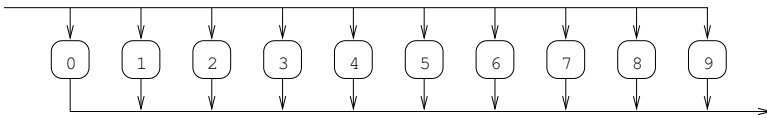



Abb. 121: Syntaxdiagramm ‘Ziffer’

Darauf aufbauend, geben wir in Abb. 122 das Syntaxdiagramm für die syntaktische Einheit ‘Gleitpunktzahl’ an, wobei für  das Syntaxdiagramm für ‘Ziffer’ einzusetzen ist.

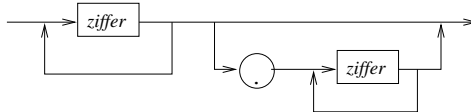


Abb. 122: Syntaxdiagramm ‘Gleitpunktzahl’

Die in Abb. 123 dargestellten Wege sind zulässige Wege vom Ausgangs- zum Endpunkt im Syntaxdiagramm ‘Gleitpunktzahl’ und somit korrekte Gleitpunktzahlen.

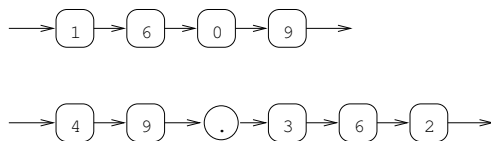


Abb. 123: Zulässige Wege im Syntaxdiagramm

Da Syntaxdiagramme oft recht umfangreich sind, müssen sie gegebenenfalls auf mehrere Zeilen umgebrochen werden. Wir verwenden die Zeichen $\rightarrow \dots$, um anzudeuten, dass ein Syntaxdiagramm an der nächsten mit $\dots \rightarrow$ bezeichneten Stelle fortgesetzt wird.

B. Software-Bezugsquellen

Wir geben im folgenden einige Verweise an, wie man die zum Arbeiten mit den Beispielen in diesem Buch geeignete Software erhalten kann.

B.1. PostgreSQL

PostgreSQL ist ein lizenzkostenfrei verfügbares relationales Datenbanksystem mit objektorientierten Konzepten. Es läuft unter vielen unix-artigen Betriebssystemen (insbesondere unter Linux) und auch unter Windows-Betriebssystemen (ab Windows NT). Für den Einsatz unter letzteren ist im Moment noch zusätzlich die Cygwin-Software erforderlich (siehe Abschnitt B.3). Eine native Unterstützung für Windows-Betriebssysteme ist in Vorbereitung und wird voraussichtlich in der nächsten PostgreSQL-Version vorhanden sein. Zum Zeitpunkt der Manuskripterstellung für dieses Buch war die PostgreSQL-Version 7.4.2 aktuell.

Weitere Informationen einschließlich Dokumentation und Download-Möglichkeit finden sich auf der offiziellen PostgreSQL-Homepage unter <http://www.postgresql.org>

Das in Abschnitt 9.2 vorgestellte Tool `phpPgAdmin` findet sich unter <http://phpPgAdmin.sourceforge.net>
das Tool `pgAdmin III` aus Abschnitt 9.3 ist zu beziehen von <http://www.pgadmin.org>

B.2. Linux

Eine komplette Systemumgebung für PostgreSQL und auch insbesondere für die in Abschnitt 8 beschriebene WWW-Anbindung von Datenbanken lässt sich gut unter dem Open Source-Betriebssystem Linux aufbauen. Man verwende am besten eine der kompletten Distributionen, die einfach zu installieren sind und viele nützliche Softwarepakete gleich mitbringen. Linux-Distributionen, die über eine verhältnismäßig umfangreiche Softwareausstattung verfügen, sind z. B.

- Slackware Linux

<http://www.slackware.com>

damit wurden das Manuskript dieser Auflage sowie die Beispiele erstellt,

- SuSE Linux

<http://www.suse.de>

und

- RedHat Enterprise Linux
<http://www.redhat.com/software/rhel/>
sowie die von RedHat gesponsorte kostenlos verfügbare Distribution
Fedora Core
<http://fedora.redhat.com/>

B.3. Cygwin für Windows

Die Cygwin-Software stellt unter Windows-Betriebssystemen eine Unix-Umgebung bereit und erlaubt somit auch den Betrieb einer PostgreSQL-Datenbank unter Windows. Der Download ist möglich unter
<http://cygwin.com>

B.4. Perl und Perl-Module

Informationen zu Perl gibt es unter
<http://www.perl.com>

Hier kann man auch den Perl-Interpreter per Download erhalten. In den oben erwähnten Linux-Distributionen ist Perl schon enthalten.

Die komplette Sammlung aller Perl-Module (also insbesondere DBI, CGI und Apache::DBI) findet sich unter
<http://cpan.perl.com>

B.5. Apache

Die offizielle Website des Apache-WWW-Servers ist
<http://httpd.apache.org>

Hier finden sich ausführliche Informationen, die Dokumentation und kostenlose Downloadmöglichkeiten (Apache ist Open Source). Die oben angegebenen Linux-Distributionen enthalten den Apache-Server bereits.

Für die in Abschnitt 8.8 beschriebene Möglichkeit, einen Perl-Interpreter in Apache direkt einzubinden, finden sich unter

<http://perl.apache.org>

weitere Informationen.

B.6. Weitere relationale Open Source-Datenbanksysteme

Außer PostgreSQL gibt es noch weitere kostenlos verfügbare Datenbanksysteme. Die folgende Auflistung ist nicht als erschöpfend anzusehen:

- Das bekannte Datenbanksystem MySQL
<http://www.mysql.com>
ist nicht für jede Anwendung geeignet, da es z. B. bis jetzt noch über keine Views verfügt. Eine native Windows-Version existiert.
- Firebird
<http://firebird.sourceforge.net>
ist die Weiterentwicklung der im Jahr 2000 als Open Source freigegebenen Datenbank InterBase.

C. WWW-Site für dieses Buch

Leser mit Zugriff auf das Internet haben die Möglichkeit, die SQL-Definitionen der in diesem Buch verwendeten Musterdatenbanken sowie längere Programmlistings aus den Abschnitten 7 und 8 per WWW abzurufen, so dass lästige Tipparbeit entfällt.

Die URL der betreffenden WWW-Site lautet

`http://www.rz.uni-passau.de/db-buch/`

Hier sind auch Linklisten auf im Text erwähnte Software sowie Aktualisierungen und eventuelle Fehlerkorrekturen zu finden.

D. Die Musterdatenbank FIRMA

Die folgenden SQL-Statements dienen zur Erzeugung der Musterdatenbank FIRMA aus Abb. 8, mit der alle Beispiele im Textteil dieses Buches erstellt wurden. Geringfügige Anpassungen auf den SQL-Dialekt des verwendeten Datenbanksystems sind evtl. erforderlich, da die SQL-Statements im PostgreSQL-Dialekt formuliert sind. Die Definition der Musterdatenbank steht alternativ auch im Internet (siehe Abschnitt C) zur Verfügung.

```
DROP TABLE zuordnung CASCADE;
DROP TABLE teile CASCADE;
DROP TABLE projekt CASCADE;
DROP TABLE personal CASCADE;
DROP TABLE abteilung CASCADE;
```

```
CREATE TABLE abteilung (
    aid integer,
    bezeichnung character varying(20),
    ort character varying(20)
);
```

```
CREATE TABLE personal (
    pid integer,
    nachname character varying(20),
    vorname character varying(15),
    strasse character varying(30),
    ort character varying(20),
    einstellung date,
    gehalt numeric(7,2),
    vorges_id integer,
    aid integer
);
```

```
CREATE TABLE projekt (
    projid integer,
    name character varying(30)
);
```

```
CREATE TABLE teile (
    ot character varying(30),
    ut character varying(30),
    n integer
);
```

```
CREATE TABLE zuordnung (
    pid integer,
```

```
    projid integer
);

INSERT INTO abteilung VALUES (1, 'Betriebsleitung', 'München');
INSERT INTO abteilung VALUES (5, 'Außendienst', 'München');
INSERT INTO abteilung VALUES (8, 'Produktion', 'Olching');

INSERT INTO personal VALUES (128, 'Meyer', 'Markus',
    'Hilblestr. 17', 'München', '1994-01-19', 4327.50, 107, 8);
INSERT INTO personal VALUES (205, 'Huber', 'Hilde',
    'Passauer Str. 2a', 'Augsburg', '1991-05-27', 4995.05, 57, 5);
INSERT INTO personal VALUES (107, 'Schmidt', 'Steffi',
    'Münchner Str. 7', 'Freising', '1990-11-02', 5722.00, 350, 8);
INSERT INTO personal VALUES (411, 'Frisch', 'Friedrich',
    'Dachauer Str. 22', 'München', '1995-09-14', 4520.67, 107, 8);
INSERT INTO personal VALUES (57, 'Klement', 'Karl',
    'Kirchfeldstr. 3', 'Bad Tölz', '1990-10-04', 6011.44, 350, 5);
INSERT INTO personal VALUES (350, 'Berger', 'Bernhard',
    'Grünaustr. 11', 'München', '1993-05-28', 8748.92, NULL, 1);

INSERT INTO projekt VALUES (3, 'Druckauftrag Fa. Karl');
INSERT INTO projekt VALUES (8, 'Beratung Fa. Seidl');
INSERT INTO projekt VALUES (11, 'Werbung Fa. Rieger');

INSERT INTO teile VALUES ('Steckdosenleiste', 'Stecker', 1);
INSERT INTO teile VALUES ('Steckdosenleiste', 'Steckdose', 5);
INSERT INTO teile VALUES ('Steckdosenleiste', 'Kabel', 1);
INSERT INTO teile VALUES ('Stecker', 'Schraube', 4);
INSERT INTO teile VALUES ('Stecker', 'Mutter', 2);
INSERT INTO teile VALUES ('Steckdose', 'Schraube', 3);
INSERT INTO teile VALUES ('Kabel', 'Draht', 3);

INSERT INTO zuordnung VALUES (128, 3);
INSERT INTO zuordnung VALUES (411, 11);
INSERT INTO zuordnung VALUES (107, 11);
INSERT INTO zuordnung VALUES (411, 3);
INSERT INTO zuordnung VALUES (205, 8);
```

Obige SQL-Statements sollten in einer Datei `firma.sql` gespeichert werden.

Es wird empfohlen, diese Musterdatenbank in PostgreSQL unter einer eigenen Datenbank-Benutzerkennung und einer separaten Datenbank anzulegen, z. B. als User "firma" und Datenbank "firma". Benutzer und Datenbank müssen zunächst wie folgt angelegt werden, wozu Superuser-Privilegien für die PostgreSQL-Installation erforderlich sind.

Verbinden Sie sich zunächst als PostgreSQL-Superuser mit der Standard-Datenbank `template1` (bei einer PostgreSQL-Standardinstallation heißt der PostgreSQL-Superuser `postgres`):

```
psql -U postgres -d template1
```

Legen Sie dann unter `psql` den Benutzer “firma” an:

```
CREATE USER firma;
```

Erzeugen Sie unter `psql` die Datenbank “firma”, wobei der Eigentümer der Benutzer “firma” sein soll:

```
CREATE DATABASE firma
```

```
WITH OWNER = firma ENCODING = 'LATIN1';
```

Beenden Sie nun die `psql`-Sitzung und melden Sie sich als Benutzer “firma” an der Datenbank “firma” an:

```
psql -U firma -d firma
```

Führen Sie unter `psql` die SQL-Statements in der oben angelegten Datei `firma.sql` aus:

```
\i firma.sql
```

Nun ist die Musterdatenbank installiert und bereit zum Üben.

E. Übungsaufgaben

Obwohl man sich durch die Lektüre von Büchern (wie etwa diesem) ein solides Wissen über Datenbanken aneignen kann, erlernt man den souveränen praktischen Umgang mit Datenbanken nur durch die aktive Beschäftigung mit Beispielen.

Zu genau diesem Zweck – den praktischen Umgang mit SQL und PL/pgSQL zu erlernen – haben wir eine Kollektion von Übungsaufgaben zusammengestellt. Die Aufgaben sind so gestellt, dass typische Praxisprobleme auftreten, mit denen man auch in größeren “Real World”-Situationen konfrontiert wird. Es werden jeweils zuerst alle Aufgaben gestellt; die kommentierten Lösungen befinden sich im darauf folgenden Abschnitt.

Die Tabelleninhalte sind nur als Beispiel anzusehen; die formulierten SQL-Befehle müssen für *jeden* – in der Situation der Aufgabenstellung sinnvollen – Tabelleninhalt das Verlangte liefern. Verwenden Sie nur die in den Aufgaben angegebene Information und nicht Information, die Sie durch Nachschauen in den Tabellen von Hand ermittelt haben.

Bitte beachten Sie, dass wir aus Platzgründen bei den den Aufgabenstellungen zugrunde liegenden Musterdatenbanken stets die – SQL-standardkompatible – Kurzform für Datentypnamen gewählt haben, also INT statt INTEGER, CHAR statt CHARACTER und VARCHAR statt CHARACTER VARYING.

E.1. Musterdatenbank THEATER

Eine Großstadt verwaltet ihre Schauspielhäuser über eine Datenbank, die aus den in Abb. 124 gezeigten Tabellen besteht.

Die Funktion der einzelnen Tabellen ist wie folgt:

THEATER	Enthält die Theater, die von der Stadt verwaltet werden. Die Spalte <code>t_id</code> enthält dabei ein eindeutiges Kürzel für jedes Theater.
PREISE	Enthält für jede Platzkategorie (eindeutiges Kürzel in Spalte <code>lage</code> , volle Bezeichnung in Spalte <code>bezeichnung</code>) den Eintrittspreis (Spalte <code>preis</code>).
PLAETZE	Enthält für jedes Theater (<code>t_id</code>) die verfügbaren Platzkategorien (<code>lage</code>), die Nummern der Reihen, in denen sich Plätze dieser Kategorie befinden

THEATER			PLAETZE			
t_id	bezeichnung	adresse	t_id	lage	reihe	anzahl
CHAR(2)	VARCHAR(40)	VARCHAR(25)	CHAR(2)	CHAR(3)	INT	INT
MK	Mozarthaus, Saal K	Lärchenallee 33	MK	PA1	1	14
MG	Mozarthaus, Saal G	Lärchenallee 33	MK	PA1	2	17
OP	Opernhaus	Kosakenring 10	MK	PA1	3	17
ST	Stadttheater	Lechstraße 2	MK	PA1	4	18
			MK	SP	5	18
			MK	SP	6	18
			MK	SP	7	18
			MK	SP	8	18
			MK	BA	1	12
			MK	BA	2	10
			MK	R1	1	30
			MK	R1T	2	28
			OP	PA1	1	22
			OP	PA1	2	27
			OP	PA1	3	28
			OP	PA1	4	27
			OP	PA2	5	28
			OP	PA2	6	27
			OP	PA2	7	27
			OP	SP	8	25
			OP	SP	9	25
			OP	LO	1	10
			OP	LO	2	9
			OP	R1	1	18
			OP	R1T	2	16
			OP	BA	1	8
			OP	R2	1	18
			ST	PA1	1	20
			ST	PA1	2	22
			ST	PA1	3	23
			ST	PA1	4	23
			ST	PA1	5	23
			ST	R1	1	30
			ST	R1T	2	26

PREISE			
lage	bezeichnung	preis	
CHAR(3)	VARCHAR(25)	NUMERIC(5,2)	
PA1	1. Parkett	75.00	
PA2	2. Parkett	68.00	
SP	Sperrsitz	61.00	
LO	Loge	90.00	
ML	Mittelloge	95.00	
BA	Balkon	75.00	
R1	1. Rang	50.00	
R1T	1. Rang Teilsicht	27.00	
R2	2. Rang	35.00	
R2T	2. Rang Teilsicht	20.00	

RESERVIERUNG			
v_id	lage	reihe	platznr
INT	CHAR(3)	INT	INT
224	SP	8	4
224	SP	8	5
244	R1	1	1
224	LO	1	1
224	LO	1	4
227	PA1	2	10
227	PA1	2	11

VORSTELLUNG					
v_id	titel	gattung	datum	uhrzeit	t_id
INT	VARCHAR(30)	VARCHAR(15)	DATE	CHAR(5)	CHAR(2)
224	Der Barbier von Sevilla	Oper	30.11.2004	19:30	OP
227	Der Zigeunerbaron	Operette	30.11.2004	20:00	ST
228	Der Zigeunerbaron	Operette	06.12.2004	19:30	ST
244	Annie, schieß los	Operette	01.12.2004	19:30	ST
220	La Traviata	Oper	02.12.2004	19:00	OP
245	Die Physiker	Schauspiel	02.12.2004	20:00	MK
259	La Traviata	Oper	08.12.2004	19:00	OP
263	Die Zauberflöte	Oper	09.12.2004	19:00	OP
277	Kammerorchester "Arte" Konzert		08.12.2004	20:00	MK

Abb. 124: Die Tabellen der Musterdatenbank THEATER

(**reihe**) sowie die Anzahl der Plätze der betreffenden Kategorie in der betreffenden Reihe (**anzahl**).

VORSTELLUNG Enthält Angaben über den Theaterspielplan, d. h. welches Stück (**titel**) wann (**datum**, **uhrzeit**) und wo (**t_id**) gegeben wird. Die Spalte **v_id** enthält für jede Vorstellung eine eindeutige Nummer. In **gattung** wird die Art des Stückes näher bezeichnet. Es wird angenommen, dass keine verschiedenen Stücke mit dem gleichen Titel existieren.

RESERVIERUNG Enthält Angaben über bereits verkaufte Eintrittskarten. Jeder Eintrag entspricht einer verkauften Karte (und damit einem belegten Platz) für eine bestimmte Vorstellungsnummer (**v_id**), eine bestimmte Platzkategorie (**lage**), eine bestimmte Reihe (**reihe**) und eine bestimmte Platznummer (**platznr**). Platznummern sind relativ zur Reihe und Platzkategorie, d. h. in jeder Reihe und jeder Platzkategorie sind die Plätze von 1 beginnend fortlaufend nummeriert.

E.1.1. Aufgaben

zu Abschnitt 2.1/4.4.3

1. Identifizieren Sie für jede Tabelle der Datenbank einen (sinnvollen) Primärschlüssel (nicht anhand der Beispieltabellen, sondern anhand der Bedeutung der Tabellen). Geben Sie die SQL-Statements zur Definition dieser Primärschlüssel an.

zu Abschnitt 3.4

2. Es sollen die Bezeichnungen der Platzkategorien ermittelt werden, die nicht mehr als EUR 50,- kosten. Formulieren Sie einen entsprechenden SQL-Befehl.
3. Formulieren Sie einen SQL-Befehl, der die Anzahl der im Dezember 2004 im Opernhaus stattfindenden Aufführungen ermittelt.
4. Erstellen Sie eine SQL-Abfrage, die eine Übersicht über die Vorstellungen im Monat Dezember 2004 liefert. Die Übersicht soll dabei folgende Angaben enthalten: Datum und Beginn der Vorstellung,

Titel des Stückes, Veranstaltungsort. Die Übersicht soll nach Datum und Uhrzeit aufsteigend sortiert sein.

5. Geben Sie einen SQL-Befehl an, der eine nach der Gesamtzahl der Plätze absteigend sortierte Liste aller Theater erstellt.
6. Erstellen Sie einen SQL-Befehl, der ermittelt, wie viele verschiedene Operetten der Vorstellungsplan zur Zeit enthält.
7. Erstellen Sie eine SQL-Abfrage, die die Gesamteinnahmen bei einer ausverkauften Vorstellung im Stadttheater berechnet.

zu Abschnitt 3.5

8. Erstellen Sie eine View

FREI (v_id, lage, anzahl)

die für jede Vorstellungsnummer und jede dort angebotene Platzkategorie die Anzahl der freien Plätze liefert. (Hinweis: Sie können sich zwei HilfsvIEWS erzeugen, die jeweils die Gesamtzahl bzw. die Zahl der reservierten Plätze pro Vorstellung und Platzkategorie berechnen.)

Unter Verwendung der View FREI soll die Anzahl der noch freien Plätze in der Platzkategorie PA1 oder LO für jeden Vorstellungstermin von “La Traviata” ermittelt werden. Formulieren Sie eine entsprechende SQL-Abfrage.

zu Abschnitt 3.6

9. Eine neue Vorstellung mit der Nr. 280 und dem Titel “Cosi fan tutte” (eine Oper), die am 10. Dezember 2004 um 19:30 Uhr im Opernhaus stattfindet, soll in die Tabelle VORSTELLUNG eingetragen werden. Formulieren Sie einen entsprechenden SQL-Befehl.
10. Die Eintrittskarte für den Platz 1 der Reihe 1 in Platzkategorie R1 für die Vorstellung von “Annie, schieß los” am 01.12.2004 wird zurückgegeben. Aktualisieren Sie die Tabelle RESERVIERUNG.
11. Die Vorstellung “Der Zigeunerbaron” am 30.11.2004 muss entfallen; für bereits verkaufte Eintrittskarten (= reservierte Plätze) wird der Eintritt zurückerstattet. Geben Sie einen SQL-Befehl an, der die Höhe des rückzuerstattenden Gesamtbetrages berechnet. Geben Sie weitere SQL-Befehle an, die die Tabellen RESERVIERUNG und VORSTELLUNG aktualisieren.

zu Abschnitt 4.2/4.3

12. Überführen Sie die Tabelle VORSTELLUNG in die dritte Normalform.

zu Abschnitt 4.4

13. Formulieren Sie einen SQL-Befehl zur Definition der Tabelle VORSTELLUNG.
14. Definieren Sie alle sinnvollen Abhängigkeits-(FOREIGN KEY) Constraints für die Tabellen der Datenbank.

zu Abschnitt 5.1/5.2

15. Es sei angenommen, dass über mehrere Arbeitsplätze auf die Datenbank zugegriffen werden kann. Bei der Platzreservierung wird einem Kunden zunächst ein Platz angeboten, den dieser dann durch Kauf der betreffenden Eintrittskarte verbindlich reservieren kann. Welche Tabellenzugriffe sind von der Ermittlung eines freien Platzes bis zur endgültigen Reservierung notwendig? Wie kann verhindert werden, dass ein freier Platz an mehreren Arbeitsplätzen gleichzeitig angeboten wird? (Das vorgeschlagene Verfahren muss nicht effektiv arbeiten.)

zu Abschnitt 6

16. Schreiben Sie eine PL/pgSQL-Funktion **resplatz**, die einen bestimmten Platz für eine bestimmte Vorstellung (gegeben durch Titel und Datum) reserviert (d. h. die Tabelle RESERVIERUNG entsprechend ändert). Die Eingabedaten sollen dabei als Parameter von **resplatz** wie folgt übergeben werden:

resplatz (titel, datum, lage, reihe, platznr)

Es wird angenommen, dass der zu reservierende Platz für die betreffende Vorstellung noch nicht reserviert ist.

17. Schreiben Sie eine PL/pgSQL-Funktion **reslage**, die für eine bestimmte Vorstellungsnummer und eine bestimmte Platzkategorie einen freien Platz (Reihe und Platznummer) ermittelt, falls ein solcher noch existiert. Die Eingabedaten sollen dabei als Parameter von **reslage** wie folgt übergeben werden:

reslage (v_id, lage)

Das Ergebnis soll als ein Datensatz in die Tabelle

PANSWER (reihe INTEGER, platznr INTEGER)

geschrieben werden, soweit es noch einen freien Platz gibt. Ansonsten soll kein Datensatz erzeugt werden.

Es sei in der Datenbank zusätzlich noch eine leere Tabelle

TMPPLATZ (**reihe** INTEGER, **platznr** INTEGER)

vorhanden. Diese Tabelle können Sie beliebig verwenden, z. B. um sich eine Liste aller Plätze (Reihe und Platznummer) der verlangten Platzkategorie in der verlangten Vorstellung anzulegen.

E.1.2. Lösungen

Beim Lösen von konkreten Datenbank-Aufgabenstellungen, die die Formulierung von SQL-Statements zum Ziel haben, ist es sinnvoll, nach folgendem Schema vorzugehen:

- (i) Klassifikation des zu formulierenden Befehls: Sollen Informationen aus der Datenbank beschafft (SELECT), Datenbankinhalte modifiziert (INSERT, UPDATE, DELETE) oder die Datenbank selbst verändert werden (DDL-Statements wie CREATE TABLE, CREATE VIEW, ALTER TABLE, ...)?
- (ii) Welche Tabellen werden für die Datenbankabfrage benötigt? Wie (über welche Joins mit welchen Attributen) müssen diese verknüpft werden?
- (iii) Welche weiteren Bedingungen für die Abfrage gibt es? Diese müssen in entsprechende WHERE-Klauseln umgesetzt werden.
- (iv) Was soll geliefert bzw. geändert werden?

Durch diese strukturierte Vorgehensweise erleichtert man sich das Erstellen von SQL-Befehlen erheblich.

Nun zu den Lösungen für die einzelnen Aufgaben. Zum besseren Verständnis der Tabelle PLAETZE ist der Platzplan für das Theater 'Mozarthaus, Saal K' in Abb. 125 graphisch dargestellt.

1. Aus der Beschreibung der THEATER-Tabelle geht hervor, dass das Attribut **t_id** ein Theater, also einen Datensatz, eindeutig bestimmt. Damit ist **{t_id}** schon einmal Superschlüssel für THEATER. Da die einzige echte Teilmenge von **{t_id}** die leere Menge **{}** ist, die niemals Schlüssel sein kann, ist **{t_id}** Schlüssel von THEATER. Weitere Schlüssel sind aus der Tabellenbeschreibung nicht ersichtlich; wir verwenden also **{t_id}** als Primärschlüssel:

ALTER TABLE theater ADD PRIMARY KEY (t_id);

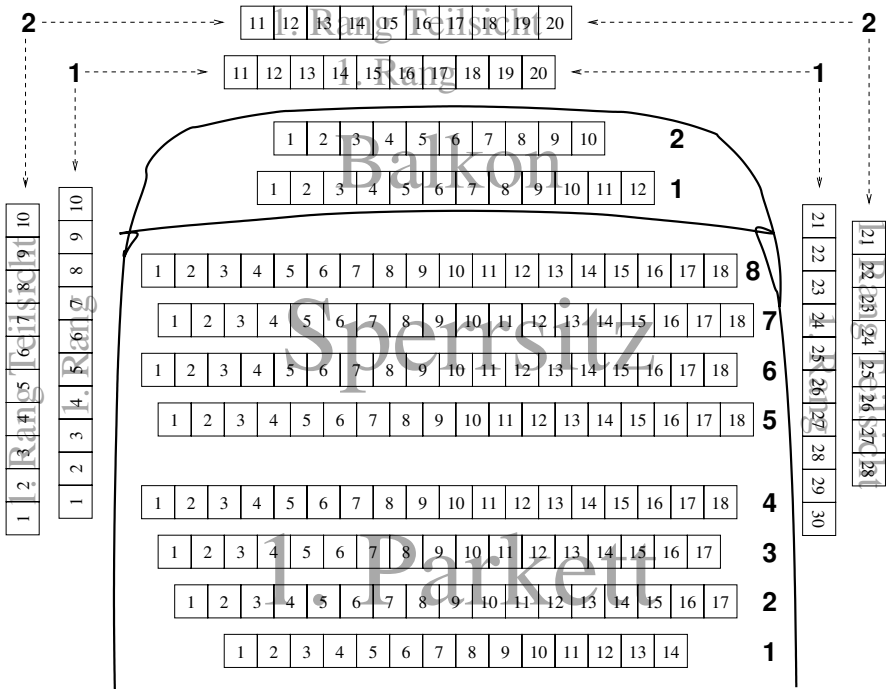


Abb. 125: Platzplan von 'Mozarthaus, Saal K'

Mit der gleichen Argumentation findet man $\{\text{lage}\}$ als Primärschlüssel von PREISE und $\{\text{v_id}\}$ als Primärschlüssel von VORSTELLUNG.

```
ALTER TABLE preise ADD PRIMARY KEY (lage);
```

```
ALTER TABLE vorstellung ADD PRIMARY KEY (v_id);
```

In der Tabelle PLAETZE befindet sich für jede verfügbare Platzreihe einer Platzkategorie in einem Theater ein eigener Datensatz, über den jeweils die Anzahl der Plätze in der betreffenden Reihe ermittelt werden kann. Für die eindeutige Identifikation eines Datensatzes in PLAETZE benötigt man offensichtlich die Attribute t_id , lage und reihe : Um die Anzahl der Plätze in einer Reihe zu erhalten, muss man wissen, in welchem Theater und in welcher Kategorie sich die Platzreihe befindet, und welche Nummer die Platzreihe hat. Aus diesem Grund bildet $\{\text{t_id}, \text{lage}, \text{reihe}\}$ einen Schlüssel von PLAETZE. Weitere Schlüssel gibt es nicht, so dass wir den einzigen Schlüssel auch als Primärschlüssel verwenden müssen.

```
ALTER TABLE plaetze
ADD PRIMARY KEY (t_id,lage,reihe);
```

Die Tabelle RESERVIERUNG enthält schließlich für jeden reservierten Platz einen eigenen Datensatz. Man erkennt unschwer, dass man für die Identifikation eines reservierten Platzes alle Angaben (Vorstellungsnummer, Platzkategorie, Reihe und Platznummer) benötigt, so dass {v_id,lage,reihe,platznr} einziger Schlüssel und daher auch Primärschlüssel von RESERVIERUNG ist.

```
ALTER TABLE reservierung
ADD PRIMARY KEY (v_id,lage,reihe,platznr);
```

2. Offensichtlich kann die benötigte Information allein aus der PREISE-Tabelle beschafft werden. Wir benötigen daher auch keinen Join, und der gesuchte SQL-Befehl lautet:

```
SELECT bezeichnung FROM preise
WHERE preis <= 50;
```

3. Die Angaben über die Zeitpunkte der Vorstellungen entnehmen wir aus der Tabelle VORSTELLUNG; die Klartextnamen der Theater stehen jedoch in der Tabelle THEATER. Wir müssen daher die beiden Tabellen durch einen Natural-Join verknüpfen; denn das zur Verknüpfung benötigte Attribut ist offensichtlich T_ID in beiden Tabellen. Der SQL-Befehl lautet dann:

```
SELECT COUNT(*)
FROM vorstellung NATURAL JOIN theater
WHERE datum >= '20041201' AND datum < '20050101'
AND bezeichnung = 'Opernhaus';
```

Durch die beiden angegebenen Bedingungen werden

- nur Vorstellungen selektiert, die am oder nach dem 1. Dezember 2004 und vor dem 1. Januar 2005, also genau im Dezember 2004 stattfinden,
- nur Vorstellungen selektiert, die im Opernhaus gegeben werden.

Da uns nicht die einzelnen Vorstellungen, sondern nur ihre Anzahl interessiert, zählen wir die Datensätze mit der COUNT-Funktion und liefern das Ergebnis als SELECT-Resultat.

4. Die Vorstellungsdaten entnehmen wir der Tabelle VORSTELLUNG, die Veranstaltungsorte der Tabelle THEATER. Gemeinsames Attribut ist t_id. Der Befehl lautet dann:

```
SELECT datum, uhrzeit, titel, bezeichnung
FROM vorstellung NATURAL JOIN theater
```

```
WHERE datum >= '20041201' AND datum < '20050101'  
ORDER BY datum, uhrzeit;
```

5. Die Platzzahl in einem Theater können wir anhand der Tabelle PLAETZE ermitteln; für die Theaternamen benötigen wir die Tabelle THEATER. Die Gesamtzahl der Plätze eines Theaters ergibt sich offensichtlich durch Addition der Werte des Attributes ANZAHL der PLAETZE-Tabelle jeweils für alle Datensätze, die zu diesem Theater gehören. Zu beachten ist, dass die Gesamtzahl der Plätze gar nicht ausgegeben, sondern nur als Sortierkriterium verwendet werden soll. Wir formulieren daher:

```
SELECT bezeichnung  
FROM plaetze NATURAL JOIN theater  
GROUP BY bezeichnung  
ORDER BY SUM(anzahl) DESC;
```

Will man alle Theater in der Liste haben (auch die, die – z. B. wegen Renovierung – nicht in der PLAETZE-Tabelle vertreten sind), muss man einen Outer-Join verwenden:

```
SELECT bezeichnung  
FROM plaetze NATURAL RIGHT JOIN theater  
GROUP BY bezeichnung  
ORDER BY COALESCE(SUM(anzahl),0) DESC;
```

6. Alle benötigten Angaben können aus der Tabelle VORSTELLUNG entnommen werden. Wir müssen die Datensätze zählen, bei denen GATTUNG den Wert 'Operette' hat, jedoch mit der Zusatzforderung, dass wir gleiche Stücke – erkennbar am gleichen Titel – nur einmal zählen. Dies erreichen wir durch die Angabe DISTINCT im Argument von COUNT:

```
SELECT COUNT(DISTINCT titel) FROM vorstellung  
WHERE gattung = 'Operette';
```

7. Die – für die Berechnung der Einnahmen benötigte – Anzahl der vorhandenen Plätze einer Platzkategorie stehen in der Tabelle PLAETZE. Den Preis für einen Platz ermitteln wir aus der Tabelle PREISE. Für die Umsetzung des Namens 'Stadttheater' benötigen wir die Tabelle THEATER. Wir berechnen dann für jede Platzreihe einer Kategorie die Einnahmen aus dieser Platzreihe (anzahl*preis) und summieren die Resultate auf:

```
SELECT SUM(anzahl*preis)  
FROM (theater NATURAL JOIN plaetze)  
JOIN preise USING(lage)
```

```
WHERE theater.bezeichnung = 'Stadttheater';
```

Man beachte, dass das Attribut mit dem Namen 'bezeichnung' sowohl in der THEATER- als auch in der PREISE-Tabelle mit unterschiedlicher Bedeutung auftritt, so dass bei dem zweiten Join explizit das Join-Attribut angegeben werden muss.

8. Dem Hinweis folgend, erstellen wir zunächst eine View GESAMT, die für eine Vorstellung und eine Platzkategorie die Gesamtzahl der verfügbaren Plätze liefert:

```
CREATE VIEW gesamt (v_id, lage, anzahl) AS
  SELECT v_id, lage, SUM(anzahl)
  FROM vorstellung NATURAL JOIN plaetze
  GROUP BY v_id, lage;
```

Ebenso erstellen wir eine weitere Hilfsview RESERVIERT, die für eine Vorstellung und eine Platzkategorie die Zahl der reservierten Plätze angibt:

```
CREATE VIEW reserviert (v_id, lage, anzahl) AS
  SELECT v_id, lage, COUNT(platznr)
  FROM reservierung
  GROUP BY v_id, lage;
```

Mittels dieser beiden Views können wir die gewünschte View FREI problemlos erzeugen: Die Zahl der freien Plätze ergibt sich aus der Gesamtzahl abzüglich der Zahl der reservierten Plätze. Hierbei müssen wir lediglich berücksichtigen, dass Kombinationen von Theatern und Platzkategorien, für die keine Platzreservierungen vorliegen, gar nicht in der RESERVIERT-View auftauchen, und somit einen Outer-Join verwenden.

```
CREATE VIEW frei (v_id, lage, anzahl) AS
  SELECT gesamt.v_id, gesamt.lage,
         gesamt.anzahl
         - COALESCE(reserviert.anzahl, 0)
  FROM gesamt LEFT JOIN reserviert USING(v_id,lage);
```

Die Formulierung der SELECT-Abfrage, die im zweiten Teil der Aufgabe verlangt ist, stellt dann kein Problem mehr dar:

```
SELECT datum, SUM(anzahl)
FROM frei NATURAL JOIN vorstellung
WHERE titel = 'La Traviata'
   AND lage IN ('PA1','LO')
GROUP BY datum;
```

9. Wir müssen den INSERT-Befehl mit einem SELECT-Statement kombinieren, da in die VORSTELLUNG-Tabelle das Theater-Kürzel eingetragen werden muss, das wir erst aus der THEATER-Tabelle ermitteln müssen:

```
INSERT INTO vorstellung
SELECT 280, 'Cosi fan tutte', 'Oper', '20041210',
      '19:30', t_id
FROM theater
WHERE bezeichnung = 'Opernhaus';
```

Das SELECT-Statement erzeugt genau einen Datensatz (mit bis auf t_id lauter konstanten Einträgen), der dann in die VORSTELLUNG-Tabelle eingefügt wird.

10. Es ist klar, dass es sich hier um eine DELETE-Anweisung handelt. In dieser muss jedoch eine Subquery verwendet werden, um aus dem Namen des Stückes und dem Termin die Vorstellungsnummer zu erhalten.

```
DELETE FROM reservierung
WHERE reihe = 1 AND platznr = 1 AND lage = 'R1'
AND v_id = (
  SELECT v_id FROM vorstellung
  WHERE titel = 'Annie, schieß los'
  AND datum = '20041201'
);
```

11. Für die Berechnung des Rückerstattungsbetrages benötigen wir offensichtlich die Tabellen RESERVIERUNG, PREISE und VORSTELLUNG (letztere, um die gegebenen Vorstellungsdaten in die Vorstellungsnummer umzusetzen).

```
SELECT SUM(preis)
FROM reservierung NATURAL JOIN preise
      NATURAL JOIN vorstellung
WHERE titel = 'Der Zigeunerbaron'
AND datum = '20041130';
```

Wir löschen dann die Reservierungen mit:

```
DELETE FROM reservierung
WHERE v_id = (
  SELECT v_id FROM vorstellung
  WHERE titel = 'Der Zigeunerbaron'
  AND datum = '20041130'
);
```

Abschließend ist die Vorstellung selbst noch zu entfernen:

```
DELETE FROM vorstellung
WHERE titel = 'Der Zigeunerbaron'
AND datum = '20041130';
```

12. Bei genauerer Betrachtung der Tabelle VORSTELLUNG stellt man fest, dass es eine funktionale Abhängigkeit gibt, die der dritten Normalform widerspricht:

$$\{\text{titel}\} \rightarrow \{\text{gattung}\}$$

Durch Dekomposition erhalten wir:

```
VORSTELLUNG1 (v_id, titel, datum, uhrzeit, t_id)
GATTUNGEN (titel, gattung)
```

13. Die für den CREATE TABLE-Befehl benötigten Parameter können direkt aus Abb. 124 abgelesen werden.

```
CREATE TABLE vorstellung (
    v_id INTEGER,
    titel CHARACTER VARYING(30),
    gattung CHARACTER VARYING(15),
    datum DATE,
    uhrzeit CHARACTER(5),
    t_id CHARACTER(2)
);
```

14. Wir geben für jede (Kind-)Tabelle an, welche Attribute von welchen Attributen in der Eltern-Tabelle abhängen. Die Angaben zu Name und Attribut der Eltern-Tabelle stehen dabei rechts vom Pfeil.

```
VORSTELLUNG: t_id ← THEATER.t_id
PLAETZE: t_id ← THEATER.t_id
         lage ← PREISE.lage
RESERVIERUNG: v_id ← VORSTELLUNG.v_id
              lage ← PREISE.lage
```

Mit Hilfe dieser Angaben lassen sich ohne Mühe SQL-Befehle für die Definition der Constraints formulieren:

```
ALTER TABLE vorstellung ADD
    FOREIGN KEY (t_id) REFERENCES theater(t_id);
ALTER TABLE plaetze ADD
    FOREIGN KEY (t_id) REFERENCES theater(t_id);
ALTER TABLE plaetze ADD
    FOREIGN KEY (lage) REFERENCES preise(lage);
ALTER TABLE reservierung ADD
```



```
FOREIGN KEY (v_id) REFERENCES vorstellung(v_id);  
ALTER TABLE reservierung ADD  
FOREIGN KEY (lage) REFERENCES preise(lage);
```

15. Der Kartenkäufer gibt an, für welche Veranstaltung (Titel, Datum) und welche Platzkategorie er Karten möchte. Es finden dann folgende Tabellenzugriffe statt:
- Für die Ermittlung der Vorstellungsnummer und der Theater-Id ist ein Lesezugriff auf die VORSTELLUNG-Tabelle erforderlich.
 - Zur Ermittlung der Lagecodes der gewünschten Platzkategorie muss auf PREISE zugegriffen werden.
 - Zur Ermittlung der freien Plätze benötigt man schließlich einen Lesezugriff auf PLAETZE und RESERVIERUNG.

Eine Auswahl freier Plätze wird nun dem Käufer angeboten. Entschcheidet er sich für einen Platz, muss ein Schreibzugriff auf die RESERVIERUNG-Tabelle durchgeführt werden.

Damit nicht zwei Käufern der gleiche Platz angeboten wird, bietet sich folgende einfache (nicht unbedingt praktikable) Lösung an: Es werden jeweils nur einem Käufer freie Plätze angeboten. Der nächste Käufer muss so lange warten, bis der erste Käufer seine Reservierung abgeschlossen hat. Dies lässt sich technisch folgendermaßen realisieren: Beim Lesezugriff auf RESERVIERUNG wird eine Exklusiv-Sperre eingerichtet. Damit können weitere Transaktionen keine freien Plätze ermitteln und anbieten, da sie dafür auch auf RESERVIERUNG zugreifen müssen. Nach dem Schreibzugriff auf die RESERVIERUNG-Tabelle muss die Exklusiv-Sperre wieder entfernt werden.

16. Die Aufgabe lässt sich einfach in zwei Schritten lösen: Zuerst beschaffen wir die Informationen, die zum Eintrag in die RESERVIERUNG-Tabelle benötigt werden, dann nehmen wir den Eintrag tatsächlich vor. Das Listing des PL/pgSQL-Programms ist in Abb. 126 gezeigt.

Lage, Reihe und Nummer des gewünschten Platzes liegen bereits als Parameter der Funktion vor. Wir müssen lediglich noch die Vorstellungsnummer aus der VORSTELLUNG-Tabelle ermitteln und in der Variablen `vnr` speichern (Zeile 13-14).

In den Zeilen 15–16 wird der Eintrag in der RESERVIERUNG-Tabelle vorgenommen.

```
1  CREATE OR REPLACE FUNCTION resplatz (  
2      CHARACTER VARYING(30), DATE, CHARACTER(3),  
3      INTEGER, INTEGER  
4  ) RETURNS void AS '  
  
5  DECLARE  
6      in_titel ALIAS FOR $1;  
7      in_datum ALIAS FOR $2;  
8      in_lage ALIAS FOR $3;  
9      in_reihe ALIAS FOR $4;  
10     in_platznr ALIAS FOR $5;  
11     vnr INTEGER;  
  
12 BEGIN  
  
13     SELECT v_id INTO vnr FROM vorstellung  
14     WHERE titel = in_titel AND datum = in_datum;  
  
15     INSERT INTO reservierung VALUES  
16         (vnr, in_lage, in_reihe, in_platznr);  
  
17     RETURN;  
  
18 END;  
19 ' LANGUAGE plpgsql;
```

Abb. 126: Reservierung eines Platzes

17. Diese etwas komplexere Aufgabe gehen wir an, indem wir das Problem folgendermaßen strukturieren:
- Ermittlung der zur Vorstellung gehörenden Theater-Id (diese wird für einen Zugriff auf die PLAETZE-Tabelle benötigt).
 - Erzeugung einer Liste mit *allen* Plätzen der gewünschten Platzkategorie im entsprechenden Theater. Für jeden vorhandenen Platz wird hierbei ein Datensatz angelegt, der aus der Reihen- sowie der Platznummer besteht.
 - Aus der im vorhergehenden Schritt erzeugten Liste suchen wir einen Platz aus, der noch nicht reserviert ist, und liefern seine Daten als Ergebnis. Gibt es keinen nichtreservierten Platz mehr, wird nichts geliefert.

Die komplette Lösung ist in Abb. 127 dargestellt. Wir beschreiben das Programm jetzt noch etwas genauer. Die in den Zeilen 6–17 definierten Variablen haben folgende Aufgaben: In `t.t_id` speichern wir das Theaterkürzel.

`p` nimmt jeweils einen Datensatz aus der PLAETZE-Tabelle auf. `fplatz` nimmt schließlich einen Datensatz aus der erzeugten Platzliste auf. Die Platzliste legen wir in der Tabelle TMPPLATZ an.

```
1  CREATE OR REPLACE FUNCTION reslage (INTEGER, CHARACTER(3))
2  RETURNS void AS '
3  DECLARE
4      in_v_id ALIAS FOR $1;
5      in_lage ALIAS FOR $2;
6      t_t_id theater.t_id%TYPE;
7      p RECORD;
8      fplatz RECORD;
9
10     f CURSOR FOR
11         SELECT * FROM tmpplatz
12         WHERE NOT EXISTS (
13             SELECT * FROM reservierung
14             WHERE v_id = in_v_id
15                 AND lage = in_lage
16                 AND reservierung.reihe = tmpplatz.reihe
17                 AND reservierung.platznr = tmpplatz.platznr
18         );
19
20 BEGIN
21     SELECT t_id INTO t_t_id FROM vorstellung
22     WHERE v_id = in_v_id;
23
24     delete from tmpplatz;
25
26     FOR p IN
27         SELECT * FROM plaetze
28         WHERE t_id = t_t_id
29             AND lage = in_lage
30     LOOP
31         FOR i IN 1 .. p.anzahl LOOP
32             INSERT INTO tmpplatz VALUES (p.reihe, i);
33         END LOOP;
34     END LOOP;
35
36     OPEN f;
37     FETCH f INTO fplatz;
38
39     IF FOUND THEN
40         INSERT INTO panswer
41         VALUES (fplatz.reihe, fplatz.platznr);
42     END IF;
43
44     RETURN;
45
46 END;
47 ' LANGUAGE plpgsql;
```

Abb. 127: Ermittlung eines freien Platzes

Der Cursor **f** (Zeilen 9–17) selektiert Einträge aus der TMPPLATZ-Tabelle, die noch nicht reservierten Plätzen entsprechen.

In den Zeilen 19 und 20 ermitteln wir zunächst das Theaterkürzel. Sodann löschen wir die TMPPLATZ-Tabelle, die wir zum Aufbau unserer Platzliste benötigen (Zeile 21).

Nun erzeugen wir für jede von der SELECT-Abfrage in Zeile 23–25 (Einträge in der PLAETZE-Tabelle für das gewünschte Theater und die gewünschte Lage) gelieferte Platzreihe die Datensätze für die entsprechenden Einzelplätze. Die äußere **for**-Schleife in den Zeilen 22–30 durchläuft jeden von der SELECT-Abfrage gelieferten Datensatz und überträgt ihn jeweils in die Variable **p.p.reihe** gibt dann die Platzreihe an, **p.anzahl** die Anzahl der Plätze in dieser Reihe. Die Nummern der Plätze laufen dann von 1 bis **p.anzahl**. Die innere **for**-Schleife in den Zeilen 27–29 erzeugt genau diese Platzdaten, die in Zeile 28 in die TMPPLATZ-Tabelle eingefügt werden.

Nach dem Zusammenbau der TMPPLATZ-Liste müssen wir uns nur noch einen Datensatz, der einem nicht reservierten Platz entspricht, aus dieser Tabelle herausgreifen. Dies geschieht in den Zeilen 31 und 32 unter Zuhilfenahme des Cursors **f**. In Zeile 33 testen wir, ob überhaupt ein geeigneter Datensatz vorhanden war. Wenn ja, setzen wir die entsprechenden Daten (Reihe und Platznummer) in die PANSWER-Tabelle ein.

E.2. Musterdatenbank HOTEL

Eine Hotelvermittlung verwaltet ihre Daten über eine Datenbank, deren Tabellen in Abb. 128 auszugsweise wiedergegeben sind.

Die Funktion der einzelnen Tabellen ist wie folgt:

LAND	Liste von Ländernamen (name) mit zugehörigem Länderkürzel (lid).
ZIEL	Enthält die Ziele, in denen das Unternehmen Hotels anbieten kann. Für jeden Ort ist eine eindeutige Nummer (zid), das Kürzel des Landes, in dem sich der Ort befindet (lid), sowie der Name des Ortes (ort) angegeben. Der Ortsname muss bezüglich verschiedener Länder nicht eindeutig sein.

BUCHUNG							UMSATZ		
kid	zid	hnr	aez	zdz	anreise	abreise	kid	jahr	betrag
INT	INT	INT	INT	INT	DATE	DATE	INT	NUMERIC(4)	NUMERIC(7,2)
275785	1903	13	2	3	04.09.2004	07.09.2004	251167	2003	1047.00
275785	2846	77	2	3	07.09.2004	09.09.2004	120509	2002	533.00
251167	4119	48	0	2	07.09.2004	11.09.2004	178990	2002	200.00
178990	3355	28	0	1	28.08.2004	01.09.2004	384133	2003	2780.00
384133	3355	28	1	0	29.08.2004	03.09.2004	384133	2004	435.00
							251167	2004	180.00

zid	hnr	hname	kat	katname	ezz	dzz	pez	pdz
INT	INT	VARCHAR(30)	INT	VARCHAR(20)	INT	INT	NUMERIC(5,2)	NUMERIC(5,2)
1742	13	Adler	3	Komfort	10	25	45.00	80.00
2007	33	Central	2	Standard	18	10	53.00	100.00
1903	13	Am Pöstlingsberg	3	Komfort	5	20	48.00	88.00
3355	27	La Mirage	4	Gehoben	17	50	95.00	175.00
1742	1	Königshof	4	Gehoben	20	30	75.00	125.00
4119	48	Nederland	1	Einfach	4	8	35.00	65.00
3355	28	Chez Claude	2	Standard	3	7	43.00	70.00
2846	77	Zum Heurigen	3	Komfort	6	17	45.00	85.00
2390	14	El Terero	5	Luxus	20	70	85.00	155.00

kid	name	vorname	adresse	plz	ort
INT	VARCHAR(30)	VARCHAR(25)	VARCHAR(30)	CHAR(5)	VARCHAR(30)
173346	Vogt	Markus	Freyunger Str. 27	94065	Waldkirchen
251167	Jäger	Elisabeth	Hofstraße 57	94034	Passau
120509	Dangl	Max	Am Gries 1	93059	Regensburg
178990	Kaufmann	Karl	Koldbruck 2a	94036	Passau
384133	Denk	Sophie	Grafinger Str. 122	94469	Deggendorf
210056	Martini	Peter	Passauer Str. 12	94065	Waldkirchen
224021	Ruhland	Christian	Steinweg 3	94032	Passau
275785	Schmidt	Sabine	Hochsteinstr. 1	94034	Passau

LAND		ZIEL		
lid	name	zid	lid	ort
CHAR(3)	VARCHAR(30)	INT	CHAR(3)	VARCHAR(30)
D	Deutschland	1344	D	München
A	Österreich	1901	A	Salzburg
CH	Schweiz	1903	A	Linz
F	Frankreich	1742	D	Nürnberg
NL	Niederlande	1583	CH	Genf
I	Italien	3768	F	Bordeaux
E	Spanien	3769	I	Bozen
		2846	A	Wien
		2007	CH	Zürich
		3355	F	Nizza
		4119	NL	Amsterdam
		2390	E	Pamplona

Abb. 128: Die Tabellen der Musterdatenbank HOTEL

- HOTEL** Die Liste der Hotels, die das Unternehmen als Unterkunftsmöglichkeit anbietet. Für jedes Hotel ist erfasst: Die Nummer des Ortes (**zid**), die innerhalb des gleichen Ortes eindeutige Hotelnummer (**hnr**), der Name des Hotels (**hname**), die Kategorieeinstufung als Zahl (**kat**) und als Text (**katname**) (hierbei ist jeder zahlenmäßigen Kategorieeinstufung ein fester Text zugeordnet), die Zahl der Einzel- (**ezz**) und Doppelzimmer (**dzz**), der Preis pro Nacht für ein Einzel- (**pez**) und ein Doppelzimmer (**pdz**), jeweils in Euro.
- KUNDE** Die Liste der Kunden des Unternehmens. Erfasst ist eine eindeutige Kundennummer (**kid**), Nachname (**name**), Vorname (**vorname**), Adresse (**adresse**), Postleitzahl (**plz**) und Wohnort (**ort**) jedes Kunden.
- BUCHUNG** Aufstellung der aktuellen Hotelbuchungen. Jede Buchung umfasst die Nummer des Kunden, der gebucht hat (**kid**), die Nummer des Ortes, in dem ein Hotel gebucht wurde (**zid**), die Hotelnummer (**hnr**), die Anzahl der gebuchten Einzel- (**aez**) bzw. Doppelzimmer (**adz**), das Anreise- (**anreise**) und das Abreisedatum (**abreise**). Es wird hierbei angenommen, dass ein Kunde nicht mehrere Buchungen zum gleichen Anreisetermin vornimmt. Die Anzahl der jeweiligen Übernachtungen ergibt sich dann aus dem Wert des Ausdrucks **abreise-anreise**.
- UMSATZ** Enthält die Kunden-Gesamtumsätze pro Jahr. Für jeden Kunden und jedes Jahr wird ein Datensatz angelegt, soweit der betreffende Kunde in diesem Jahr Hotelübernachtungen in Anspruch genommen hat. **kid** enthält die Kundennummer, **jahr** das Jahr und **betrag** den Umsatz des betreffenden Kunden im entsprechenden Jahr. Dieser errechnet sich aus der Summe der Preise aller Übernachtungen, die der betreffende Kunde im betreffenden Jahr in Anspruch genommen hat. Für das laufende Jahr (2004) enthält **betrag** den bis jetzt erreichten Jahresumsatz, dabei sind die in der Tabelle BUCHUNG enthaltenen Umsätze noch nicht berücksichtigt.

E.2.1. Aufgaben

zu Abschnitt 2.1/4.4.3

1. Identifizieren Sie für jede Tabelle der gegebenen Datenbank einen (sinnvollen) Primärschlüssel und geben Sie die SQL-Statements zur Definition dieser Primärschlüssel an.

zu Abschnitt 3.4

2. Formulieren Sie einen SQL-Befehl, der eine Liste der Hotels in Nizza/Frankreich liefert, die für ein Einzelzimmer höchstens EUR 50 pro Nacht verlangen.
3. Geben Sie einen SQL-Befehl an, der eine Liste aller Hotels mit den Angaben Land, Ort, Hotelname liefert. Die Liste soll absteigend nach Kategorienummern (1. Sortierkriterium) und aufsteigend nach dem Preis für ein Doppelzimmer (2. Sortierkriterium) sortiert sein.
4. Geben Sie einen SQL-Befehl an, der eine Liste aller Länder liefert, in denen zur Zeit keine Hotels angeboten werden.
5. Geben Sie einen SQL-Befehl an, der eine Liste aller Länder mit der Anzahl der dort angebotenen Hotels erstellt. Länder, in denen zur Zeit keine Hotels angeboten werden, sollen nicht in der Liste stehen.
6. Formulieren Sie einen SQL-Befehl, der Namen und Wohnort aller Kunden ausgibt, die mindestens zwei Hotelaufenthalte gebucht haben.
7. Gesucht sind die Namen aller Kunden, die einen Hotelaufenthalt gebucht haben, der eine Hotelübernachtung vom 8. auf den 9. September 2004 einschließt. Formulieren Sie einen SQL-Befehl, der diese Namen liefert.
8. Es soll eine Liste erstellt werden, die für jeden Kunden den Namen und die Umsatzsumme aus den Jahren 2003 und 2004 (laut Umsatztabelle) enthält. Kunden ohne Umsatz sollen dabei mit Umsatz 0 in der Tabelle stehen. Formulieren Sie einen entsprechenden SQL-Befehl.
9. Es soll eine Liste erstellt werden, die für jede Buchung den Kundennamen, den Zielort und das Land, den Hotelnamen, das Anrei-

sedatum und den Preis des gebuchten Aufenthalts enthält. Geben Sie einen entsprechenden SQL-Befehl an.

zu Abschnitt 3.5

10. Definieren Sie in SQL eine View PRLISTE, die eine Liste aller Hotels mit den Angaben Land, Ort, Hotelname, Kategorienummer, Einzelzimmerpreis, Doppelzimmerpreis enthalten soll.
11. Geben Sie einen SQL-Befehl an, der die Anzahl der freien Doppelzimmer im Hotel “Am Pöstlingberg” in Linz/Österreich in der Nacht vom 5. auf den 6. September 2004 liefert. (Hinweis: Sie können sich bei Bedarf zusätzlich eine Hilfsview definieren.)

zu Abschnitt 3.6

12. Das Reiseunternehmen nimmt ein neues Hotel unter Vertrag: Das “Kardinal” in Salzburg/Österreich. Es gehört der Kategorie 5 (“Luxus”) an und verfügt über 15 Einzel- und 25 Doppelzimmer. Der Preis für ein Einzelzimmer liegt bei EUR 110, der Preis für ein Doppelzimmer bei EUR 190. Das Hotel soll die Hotelnummer 24 erhalten. Formulieren Sie einen SQL-Befehl zur Aufnahme des neuen Hotels in die Tabelle HOTEL.
13. Sophie Denk aus Deggendorf storniert Ihre Buchung für das Anreisedatum 29.08.2004. Formulieren Sie einen SQL-Befehl, der die Tabelle BUCHUNG aktualisiert.
14. Durch eine Steuererhöhung müssen alle Hotels in Frankreich ihre Preise um 10% erhöhen. Formulieren Sie einen SQL-Befehl, mit dem die notwendige Aktualisierung der Tabelle HOTEL durchgeführt werden kann.

zu Abschnitt 4.2/4.3

15. Überführen Sie die Tabelle HOTEL in die dritte Normalform.

zu Abschnitt 4.4

16. Definieren Sie sämtliche Abhängigkeits-(FOREIGN KEY) Constraints für die Tabellen der gegebenen Datenbank und geben Sie die SQL-Statements zur Definition dieser Constraints an.

zu Abschnitt 6

17. Schreiben Sie eine PL/pgSQL-Funktion `clean ()`, die alle Kunden, die 2003 und 2004 keinen Umsatz gemacht haben sowie keinen Eintrag in der Buchungstabelle haben, aus den Tabellen KUNDE und UMSATZ entfernt.
18. Schreiben Sie eine PL/pgSQL-Funktion `cbuchung (stichtag)` mit einem Parameter `stichtag` vom Typ DATE, die folgende Aufgabe ausführt:

Die Kosten eines Hotelaufenthalts der “alten” Einträge der BUCHUNG-Tabelle (das sind diejenigen, bei denen das Abreisedatum vor dem `stichtag` liegt), sollen den betreffenden Kundenumsätzen in der UMSATZ-Tabelle zugeschlagen werden. Anschließend sollen die alten Einträge der BUCHUNG-Tabelle gelöscht werden.

Beachten Sie hierbei folgende Hinweise:

- Sie können aus einer Datumsangabe d (kann auch ein Attributwert sein) das Jahr in vierstelligem Format mit dem Ausdruck `TO_CHAR(d, 'YYYY')` ermitteln.
- Für das Jahr, auf das die Kosten des Hotelaufenthaltes gebucht werden, ist das Abreisedatum maßgeblich.
- Sie müssen evtl. in der UMSATZ-Tabelle einen Datensatz anlegen, wenn für den Kunden und das Jahr noch kein Datensatz existiert.

E.2.2. Lösungen

Man beachte die allgemeinen Lösungshinweise am Anfang von Abschnitt E.1.2. Die Lösungsskizzen in diesem Abschnitt sind meist etwas kürzer gehalten als bei der ersten Musterdatenbank, da davon ausgegangen wird, dass der Leser nun schon etwas Praxis bei der Bearbeitung von Datenbankaufgaben besitzt.

1. Man erkennt ohne Probleme, dass `{lid}` Schlüssel von LAND, `{zid}` Schlüssel von ZIEL und `{kid}` Schlüssel von KUNDE ist. Wir zeichnen diese Schlüssel jeweils als Primärschlüssel aus.

Bei der Beschreibung der Tabelle HOTEL ist vermerkt, dass `hnr` nur innerhalb des gleichen Ortes (gleicher Wert von `zid`) eindeutig ist. Damit bildet `{zid,hnr}` einen Schlüssel von HOTEL.

Die UMSATZ-Tabelle enthält pro Kunde und Jahr einen Datensatz. Damit bildet {kid,jahr} einen Schlüssel von UMSATZ. Laut den Erläuterungen zur Tabelle BUCHUNG nimmt ein Kunde nicht mehrere Buchungen zum gleichen Anreisetermin vor, also ist {kid,anreise} Schlüssel von BUCHUNG. Wir zeichnen die gefundenen Schlüssel jeweils als Primärschlüssel aus.

Die entsprechenden SQL-Statements lauten:

```
ALTER TABLE land ADD PRIMARY KEY (lid);
ALTER TABLE ziel ADD PRIMARY KEY (zid);
ALTER TABLE kunde ADD PRIMARY KEY (kid);
ALTER TABLE hotel ADD PRIMARY KEY (zid, hnr);
ALTER TABLE umsatz ADD PRIMARY KEY (kid, jahr);
ALTER TABLE buchung ADD PRIMARY KEY (kid, anreise);
```

2. Um die geforderte Information liefern zu können, benötigen wir die Tabellen HOTEL, LAND und ZIEL. Diese verknüpfen wir per Natural-Join und geben zusätzlich noch die in der Aufgabe gegebenen Bedingungen an:

```
SELECT hname
FROM land NATURAL JOIN ziel NATURAL JOIN hotel
WHERE name = 'Frankreich'
      AND ort = 'Nizza'
      AND pez <= 50;
```

3. Wir benötigen zur Lösung dieser Aufgabe wieder die Tabellen HOTEL, LAND und ZIEL, verknüpft mit Natural-Joins. Die ORDER BY-Klausel ergibt sich direkt aus der Aufgabe.

```
SELECT name, ort, hname
FROM land NATURAL JOIN ziel NATURAL JOIN hotel
ORDER BY kat DESC, pdz;
```

4. Dies ist ein typisches Beispiel für eine Negativabfrage. Wir lösen diese mit einem NOT IN-Konstrukt.

```
SELECT name FROM land
WHERE lid NOT IN (
    SELECT lid FROM ziel NATURAL JOIN hotel
);
```

5. Für die Beschaffung der geforderten Informationen benötigen wir offensichtlich die Tabellen HOTEL, LAND und ZIEL, die wie gehabt durch Natural-Joins zu verknüpfen sind. Der Hinweis, dass Länder, in denen keine Hotels angeboten werden, nicht in der Er-

gebnisliste stehen sollen, erspart uns die Mühe, einen Outer-Join zu formulieren. Statt dessen lautet die Abfrage einfach:

```
SELECT name, COUNT(hnr)
FROM land NATURAL JOIN ziel NATURAL JOIN hotel
GROUP BY name;
```

6. Die Kundendaten entnehmen wir der Tabelle KUNDE, während die Buchungen in der Tabelle BUCHUNG aufgeführt sind. Die Tabellen werden per Natural-Join über das gemeinsame Attribut `kid` verknüpft. Hierbei sollen jeweils alle zu einem Kunden gehörenden Datensätze zu einer Gruppe zusammengefasst werden. Ein- und derselbe Kunde wird über die gleiche Kundennummer (`kid`) identifiziert. Schließlich sollen nur die Gruppen von Datensätzen zur Ausgabe gewählt werden, die aus mindestens zwei Datensätzen bestehen. Also:

```
SELECT name, ort
FROM kunde NATURAL JOIN buchung
GROUP BY name, ort, kid
HAVING COUNT(anreise) >= 2;
```

Bei GROUP BY müssen auch die Attribute `name` und `ort` aufgeführt werden, da diese in der SELECT-Liste stehen, ohne dass eine Gruppenfunktion darauf angewendet wird.

7. Um die Aufgabe lösen zu können, benötigen wir die Tabellen KUNDE und BUCHUNG. Das Interessante an dieser Aufgabe ist die Zusatzbedingung “Übernachtung vom 8. auf den 9. September”. Diese liegt offensichtlich dann vor, wenn das Anreisedatum vor dem oder am 8. September und das Abreisedatum nach dem oder am 9. September liegt.

```
SELECT name
FROM kunde NATURAL JOIN buchung
WHERE anreise <= '20040908'
AND abreise >= '20040908';
```

8. Wir erzeugen zunächst eine Hilfsview, die für jeden Kunden die Summe der Umsätze in den Jahren 2003 und 2004 angibt, wobei hier nur Kunden aufgeführt werden sollen, die in diesen Jahren auch tatsächlich Umsatz gemacht haben:

```
CREATE VIEW humsatz (kid, betrag) AS
SELECT kid, SUM(betrag)
FROM umsatz
WHERE jahr IN (2003, 2004)
```

```
GROUP BY kid;
```

Die Erzeugung der Ergebnistabelle geschieht nun unter Verwendung dieser Hilfsview, wobei wir durch einen Outer Join mit der Kundentabelle erreichen, dass auch Kunden ohne Umsatz ausgegeben werden. Damit letztere mit Umsatz 0 erscheinen, verwenden wir die COALESCE-Funktion:

```
SELECT name, COALESCE(betrag,0)
FROM kunde NATURAL LEFT JOIN humsatz;
```

9. Für die Lösung dieser Aufgabe brauchen wir mit Ausnahme der UMSATZ-Tabelle alle Tabellen der Datenbank, also 5 Tabellen. Die Tabellen LAND, ZIEL, HOTEL und Buchung können wie gehabt per Natural-Join verknüpft werden. Mit der Tabelle KUNDE ist dies nicht möglich, da hier ein zweites Mal das Attribut ORT in anderer Bedeutung vorkommt. Wir führen daher einen Equi-Join mit expliziter Angabe des "gleichzusetzenden" Attributnamens (KID) durch.

Ansonsten stellt sich noch die Frage, wie man für eine Buchung jeweils den Preis des gebuchten Aufenthalts berechnet. Der Preis für eine Übernachtung setzt sich zusammen aus dem Preis für ein Einzelzimmer, multipliziert mit der Zahl der gebuchten Einzelzimmer, und dem Preis für ein Doppelzimmer, multipliziert mit der Zahl der gebuchten Doppelzimmer. Der Übernachtungspreis muss nun noch mit der Aufenthaltsdauer multipliziert werden.

Somit ergibt sich:

```
SELECT kunde.name, ziel.ort, land.name,
       hname, anreise,
       (abreise - anreise) * (aez*pez + adz*pdz)
FROM (land NATURAL JOIN ziel NATURAL JOIN hotel
      NATURAL JOIN buchung)
JOIN kunde USING (kid);
```

10. Die benötigten Angaben sind in den Tabellen HOTEL, LAND und ZIEL enthalten.

```
CREATE VIEW prliste AS
SELECT name, ort, hname, kat, pez, pdz
FROM land NATURAL JOIN ziel NATURAL JOIN hotel;
```

11. Dem Hinweis der Aufgabe folgend, erzeugen wir zunächst eine Hilfsview DZBELEGT, die für jedes Hotel die Anzahl der in der Nacht vom 5. auf den 6. September belegten Doppelzimmer lie-

fert (Hotels, in denen in der betreffenden Nacht nichts belegt ist, werden nicht aufgeführt):

```
CREATE VIEW dzbelegt (zid, hnr, anzahl) AS
  SELECT zid, hnr, SUM(adz)
  FROM hotel NATURAL JOIN buchung
  WHERE anreise <= '20040905'
        AND abreise >= '20040905'
  GROUP BY zid, hnr;
```

Nun müssen wir für das gewünschte Hotel nur noch die von der View gelieferte Anzahl von der vorhandenen Anzahl an Doppelzimmern abziehen. Ein Outer-Join ist jedoch erforderlich, da die View ja eventuell für das gewünschte Hotel nichts liefert (wenn keine Zimmer reserviert sind).

```
SELECT dzz - COALESCE(anzahl,0)
FROM (land NATURAL JOIN ziel NATURAL JOIN hotel)
     NATURAL LEFT JOIN dzbelegt
WHERE name = 'Österreich'
      AND ort = 'Linz'
      AND hname = 'Am Pöstlingberg';
```

12. Der notwendige INSERT-Befehl muss mit einer SELECT-Abfrage kombiniert werden, um die Zielnummer zu ermitteln:

```
INSERT INTO hotel
  SELECT zid, 24, 'Kardinal', 5, 'Luxus', 15, 25,
         110, 190
  FROM land NATURAL JOIN ziel
  WHERE name = 'Österreich'
        AND ort = 'Salzburg';
```

13. Um die für die Löschoperation in BUCHUNG erforderliche Kundennummer herauszufinden, benötigen wir eine Subquery:

```
DELETE FROM buchung
WHERE kid = (
  SELECT kid FROM kunde
  WHERE name = 'Denk'
        AND vorname = 'Sophie'
        AND ort = 'Deggendorf'
)
AND anreise = '20040829';
```

14. Wir müssen uns hier per Subquery eine Liste aller Zielnummern, die in Frankreich liegende Orte kennzeichnen, beschaffen:

```
UPDATE hotel
SET pez = 1.1 * pez, pdz = 1.1 * pdz
WHERE zid IN (
    SELECT zid
    FROM land NATURAL JOIN ziel
    WHERE name = 'Frankreich'
);
```

15. Offensichtlich “stört” die Abhängigkeit

$\{\text{kat}\} \rightarrow \{\text{katname}\}$

Dekomposition liefert

HOTEL1 (zid, hnr, hname, kat, ezz, dzz, pez, pdz)
HOTEL2 (kat, katname)

16. Wir geben für jede (Kind-)Tabelle an, welche Attribute von welchen Attributen in der Eltern-Tabelle abhängen. Die Angaben zu Name und Attribut der Eltern-Tabelle stehen dabei rechts vom Pfeil.

```
BUCHUNG: kid ← KUNDE.kid
          {zid,hnr} ← HOTEL.{zid,hnr}
UMSATZ: kid ← KUNDE.kid
HOTEL: zid ← ZIEL.zid
ZIEL: lid ← LAND.lid
```

Hier die Constraintdefinitionen in SQL:

```
ALTER TABLE buchung
  ADD FOREIGN KEY (kid) REFERENCES kunde(kid);
ALTER TABLE buchung
  ADD FOREIGN KEY (zid, hnr)
  REFERENCES hotel(zid, hnr);
ALTER TABLE umsatz
  ADD FOREIGN KEY (kid) REFERENCES kunde(kid);
ALTER TABLE hotel
  ADD FOREIGN KEY (zid) REFERENCES ziel(zid);
ALTER TABLE ziel
  ADD FOREIGN KEY (lid) REFERENCES land(lid);
```

17. Im Prinzip müssen wir nur zwei normale SQL-Befehle hintereinander ausführen. Der erste Befehl löscht die gewünschten Datensätze aus der KUNDE-Tabelle, der zweite die Datensätze aus der UMSATZ-Tabelle. Das entsprechende PL/pgSQL-Programm findet sich in Abb. 129.

```
1  CREATE OR REPLACE FUNCTION clean ()
2  RETURNS void AS '
3  BEGIN
4      DELETE FROM kunde
5      WHERE kid NOT IN (
6          SELECT kid FROM umsatz
7          WHERE jahr in (2003,2004)
8              AND betrag > 0
9          UNION
10         SELECT kid FROM buchung
11     );
12     DELETE FROM umsatz
13     WHERE kid NOT IN (
14         SELECT kid FROM umsatz
15         WHERE jahr in (2003,2004)
16             AND betrag > 0
17         UNION
18         SELECT kid FROM buchung
19     );
20     RETURN;
21 END;
22 ' LANGUAGE plpgsql;
```

Abb. 129: Löschen von “Karteileichen”

Es ist allerdings überhaupt nicht klar, warum diese Lösung eigentlich funktioniert: Sind Constraints auf der Datenbank definiert (siehe vorherige Aufgabe), können diese verletzt werden, wenn Kunden zuerst aus der KUNDEN-Tabelle gelöscht werden, aber noch einen Eintrag in der UMSATZ-Tabelle haben. Damit würde man erwarten, dass die Ausführung des ersten DELETE-Statements in der geschilderten Situation vom Datenbanksystem abgelehnt und die Aktion abgebrochen wird.

In PostgreSQL funktioniert dies jedoch, weil die Gültigkeit von Constraints innerhalb einer Funktion nicht geprüft wird, die Prüfung findet frühestens nach Ausführung des Statements statt, das die Funktion aufruft. Zu diesem Zeitpunkt ist in unserem Fall aber alles wieder in Ordnung, da ja in der Funktion im zweiten Schritt auch die Einträge der entfernten Kunden in der UMSATZ-Tabelle gelöscht werden.

Um die Funktion prinzipiell auch in Datenbanksystemen anwenden zu können, die Constraints innerhalb von Funktionen prüfen, kann

man wie folgt “auf Nummer Sicher” gehen: Man beschafft sich zunächst die *nicht* zu löschenden Kundennummern und speichert sie in einer temporären Tabelle. Dafür sei in der Datenbank die Tabelle KTMP vorhanden, die etwa mit

```
CREATE TABLE ktmp (kid INTEGER);
```

erzeugt worden sein könnte. Nach dem Aufbau der Tabelle KTMP kann man nämlich zuerst die Datensätze in der Tabelle UMSATZ, dann diejenigen in der Tabelle KUNDE löschen, so dass sich auch innerhalb der Funktion keine Integritätsverletzungen mehr ergeben. Das verbesserte PL/pgSQL-Programm ist in Abb. 130 abgebildet.

```
1  CREATE OR REPLACE FUNCTION clean ()
2  RETURNS void AS '
3
4  BEGIN
5
6      DELETE FROM ktmp;
7
8      INSERT INTO ktmp
9      SELECT kid FROM umsatz
10     WHERE jahr in (2003,2004)
11           AND betrag > 0
12     UNION
13     SELECT kid FROM buchung;
14
15     DELETE FROM umsatz
16     WHERE kid NOT IN (
17         SELECT kid FROM ktmp
18     );
19
20     DELETE FROM kunde
21     WHERE kid NOT IN (
22         SELECT kid FROM ktmp
23     );
24
25     DELETE FROM ktmp;
26
27     RETURN;
28
29 END;
30 ' LANGUAGE plpgsql;
```

Abb. 130: Löschen von “Karteileichen”, verbesserte Version

18. Wir vollziehen die Lösung der Aufgabe in folgenden Schritten:

- Wir ermitteln die betroffenen Datensätze aus der BUCHUNG-Tabelle (das sind alle, bei denen das Abreisedatum vor dem **stichtag** liegt).

- Für jeden dieser Datensätze berechnen wir die Kosten des gebuchten Aufenthalts.
- Falls für das betreffende Jahr (das Jahr des Abreisedatums) in der UMSATZ-Tabelle noch kein Datensatz für den Kunden existiert, erzeugen wir zunächst einen mit Umsatz 0.
- Dann addieren wir den Umsatz des aktuellen Aufenthaltes zum vorhandenen Umsatz.
- Am Schluss werden alle Buchungen, deren Abreisedatum vor dem Stichtag liegt, aus der BUCHUNG-Tabelle gelöscht.

Der jeweils aktuelle Datensatz der BUCHUNG-Tabelle wird in der RECORD-Variablen `b` gespeichert. Den Umsatz des aktuellen Aufenthalts legen wir in der Variablen `umsatz_buchung` ab. In `j` speichern wir das Jahr des Abreisedatums.

Das Programmlisting ist in Abb. 131 aufgeführt.

Die Schleife in den Zeilen 9–32 wird für jeden Datensatz in der BUCHUNG-Tabelle, bei dem das Abreisedatum vor oder an dem heutigen Tag liegt, ausgeführt. Der gerade behandelte Datensatz wird in die Variable `b` geladen (Zeile 9). In den Zeilen 13–18 wird der mit der aktuellen Buchung verbundene Umsatz berechnet und in die Variable `umsatz_buchung` geschrieben.

Die Zeilen 20–22 prüfen, ob schon ein geeigneter Datensatz in der UMSATZ-Tabelle existiert, dem der berechnete Umsatz zugeschlagen werden kann. Da uns bei dieser Auswahlabfrage nur interessiert, ob es Datensätze gibt, die den spezifizierten Bedingungen entsprechen, nicht jedoch die Ergebnistabelle, schreiben wir `PERFORM` statt `SELECT...INTO`. Falls kein Datensatz in der UMSATZ-Tabelle existiert (Zeile 23–27), wird per `INSERT` ein neuer Datensatz in der UMSATZ-Tabelle erzeugt.

In den Zeilen 28–31 wird der Einzelumsatz dem Jahresumsatz des Kunden zugeschlagen. Dies geschieht per `UPDATE`-Statement. Es wird immer ein Datensatz modifiziert, da der Fall, dass noch kein passender Datensatz in der UMSATZ-Tabelle existiert, schon zuvor behandelt wurde (durch Erzeugung eines passenden Datensatzes).

Schließlich löschen die Zeilen 33–34 die behandelten Datensätze aus der BUCHUNG-Tabelle.

```
1  CREATE OR REPLACE FUNCTION cbuchung (DATE)
2  RETURNS void AS '
3
4  DECLARE
5      stichtag ALIAS FOR $1;
6      b RECORD;
7      umsatz_buchung umsatz.betrag%TYPE;
8      j umsatz.jahr%TYPE;
9
10 BEGIN
11     FOR b IN
12         SELECT * FROM buchung
13         WHERE abreise < stichtag
14     LOOP
15         SELECT (b.abreise - b.anreise)
16             * (b.aez*pez + b.adz*pdz)
17         INTO umsatz_buchung
18         FROM hotel
19         WHERE hotel.zid = b.zid
20             AND hotel.hnr = b.hnr;
21         j := TO_CHAR(b.abreise, 'YYYY');
22         PERFORM kid FROM umsatz
23             WHERE umsatz.kid = b.kid
24             AND jahr = j;
25         IF NOT FOUND THEN
26             INSERT INTO umsatz VALUES (
27                 b.kid, j, 0
28             );
29         END IF;
30         UPDATE umsatz
31         SET betrag = betrag + umsatz_buchung
32         WHERE kid = b.kid
33             AND jahr = j;
34     END LOOP;
35     DELETE FROM buchung
36     WHERE abreise < stichtag;
37     RETURN;
38 END;
39 ' LANGUAGE plpgsql;
```

Abb. 131: Abrechnung erfolgter Aufenthalte

E.3. Musterdatenbank VHS

Eine Volkshochschule verwaltet ihren Datenbestand über angebotene Kurse und Anmeldungen mit einer Datenbank, deren Tabellen in Abb. 132 auszugsweise wiedergegeben sind. NULL-Werte in Spalten sind durch ein leeres Feld dargestellt.

Die Funktion der einzelnen Tabellen ist wie folgt:

TEILNEHMER	Gibt an, welche Personen als (jetzige oder ehemalige) Kursteilnehmer registriert sind. Erfasst ist die eindeutige Teilnehmernummer (tnr), Nachname (name), Vorname (vorname), Adresse (adresse), Postleitzahl (plz) und Wohnort (wohntort).
ORTE	Enthält die Orte, an denen Kurse stattfinden können. Angegeben ist jeweils ein eindeutiges Kürzel (id), der Name des Ortes (name) sowie seine Adresse (adresse).
GEBUEHREN	Hier sind die Teilnahmegebühren für die Kurse abgelegt. Jeder Kurs wird in eine bestimmte Preisgruppe eingestuft. Die Kursgebühr pro Teilnehmer hängt unter Umständen von der Zahl der angemeldeten Teilnehmer ab. Ein Datensatz der Tabelle enthält jeweils für eine Preisgruppe (pg) und eine Teilnehmerzahl von tnmin bis einschließlich tnmax Teilnehmer die Kursgebühr (preis) in EUR, die jeder Teilnehmer bezahlen muss. Für einen bestimmten Kurs gibt der Datensatz in der entsprechenden Preisgruppe mit dem kleinsten Attributwert von tnmin die Mindestteilnehmerzahl an, die erreicht werden muss, damit der Kurs zustande kommt. In der gleichen Weise gibt der Datensatz in der entsprechenden Preisgruppe mit dem größten Attributwert von tnmax die maximal zulässige Teilnehmerzahl für den Kurs an.
KURSE	Verzeichnis der angebotenen Kurse. Für jeden Kurs ist angegeben: Die eindeutige Kursnummer (knr), der Name des Kurses (titel), der Nachname des Leiters (leiter), soweit bekannt. Falls der Name nicht bekannt ist, erhält das Attribut den NULL-Wert. Weiterhin ist für jeden Kurs an-

ANMELDUNGEN				ORTE		
tnr	knr			id	name	adresse
INT	INT			CHAR(3)	VARCHAR(30)	VARCHAR(30)
11090	6601			VHS	Volkshochschule	Nikolastr. 18
12009	7205			KBS	Kaufmännische Berufsschule	Neuburger Str. 96e
11224	13133			JH	Josefsheim	Neuburger Str. 60
11090	3112			HB	Städtisches Hallenbad	Neuburger Str. 96c
11224	6601			TS	Tauchschnule Neptun	Kapuzinerstr. 9
11407	7101			OB	Oberhausmuseum	Rennweg
11090	13096			TH	Stadttheater	Gottfried-Schäffer-Str. 2
11074	13133					

GEBUEHREN				TERMINE		
pg	tnmin	tnmax	preis	knr	tag	zeit
CHAR(2)	INT	INT	NUMERIC(6,2)	INT	CHAR(2)	CHAR(5)
E1	13	25	140.00	6601	Mo	18:30
S	5	6	190.00	6601	Mi	18:30
S	7	8	135.00	6626	Di	18:30
S	9	10	105.00	7101	Mo	18:30
S	11	22	85.00	7101	Do	18:30
E4	4	25	420.00	7205	Mo	19:30
E1	4	12	200.00	7205	Do	18:00
S2	5	22	210.00	13013	Di	19:30
G1	2	14	110.00	13013	Do	19:30
G2	3	10	180.00	13133	Mo	18:00
T	10	50	5.00	13096	Do	19:30
				3112	Mo	19:30
				6602	Di	18:30
				6602	Do	18:30

knr	titel	leiter	id	abende	dauer	pg	beginn
INT	VARCHAR(40)	VARCHAR(30)	CHAR(3)	INT	INT	CHAR(2)	DATE
6601	EDV-Grundlagen		VHS	4	180	E1	04.10.2004
6626	Programmieren in C		JH	10	180	E4	12.10.2004
7101	Englisch Express I	Oppermann	KBS	20	135	S2	04.10.2004
7209	Französisch M1	Merkle	KBS	12	90	S	04.10.2004
7205	Französisch G2	Wittmann	KBS	12	90	S	07.10.2004
13013	Wassergymnastik	Straub	HB	8	60	G1	05.10.2004
13133	Tauchschnule	Straub	TS	3	180	G2	22.11.2004
3112	Einführung "Medea"	Celeste	TH	1	90	T	06.12.2004
13096	Fitness-Training	Jungwirth	JH	9	60	G1	07.10.2004
6602	EDV-Grundlagen		VHS	4	180	E1	05.10.2004

TEILNEHMER					
tnr	name	vorname	adresse	plz	wohnort
INT	VARCHAR(30)	VARCHAR(20)	VARCHAR(30)	CHAR(5)	VARCHAR(30)
11074	Öllinger	Matthias	Holzheimerstr. 2	94032	Passau
11090	Hg	Sabine	Bayerwaldring 12	94104	Tittling
11221	Kaiser	Willi	Am Kirchberg 17	94164	Sonnen
11224	Haas	Bernhard	Königsberger Str. 22	94036	Passau
11225	Buchbauer	Maria	Abteistr. 18	94034	Passau
11407	Rieger	Bettina	Warbachweg 7	94474	Vilshofen
12009	Kraus	Heinz	Stadtplatz 12	94474	Vilshofen

Abb. 132: Die Tabellen der Musterdatenbank VHS

gegeben: Das Kürzel für den Veranstaltungsort (**id**), die Anzahl der Abende, an denen der Kurs stattfindet (**abende**), die Dauer eines Kursabends in Minuten (**dauer**), die Preisgruppe des Kurses (**pg**), sowie das Datum des Kursbeginns (**beginn**).

TERMINE Enthält die wöchentlichen Kurstermine. Für jeden Termin ist angegeben die Kursnummer (**knr**), die ersten zwei Buchstaben des Wochentags (**tag**) und die Uhrzeit des Beginns (**zeit**). Es wird angenommen, dass es für den gleichen Kurs nicht zwei Termine am gleichen Wochentag gibt.

ANMELDUNGEN Liste der vorliegenden Anmeldungen. Für jede Anmeldung ist die Nummer des Teilnehmers (**tnr**) und die Nummer des Kurses (**knr**), für den sich der Teilnehmer angemeldet hat, angegeben.

Es wird davon ausgegangen, dass die Tabellen keine widersprüchlichen Informationen enthalten.

E.3.1. Aufgaben

zu Abschnitt 2.1/4.4.3

1. Identifizieren Sie für jede Tabelle der Datenbank einen (sinnvollen) Primärschlüssel und geben Sie die SQL-Statements zur Definition der Primärschlüssel an.

zu Abschnitt 3.4

2. Eine Liste, die für jeden angebotenen Kurs den Titel und den Veranstaltungsort enthält, soll erstellt werden. Geben Sie einen SQL-Befehl an, der dies leistet.
3. Geben Sie einen SQL-Befehl an, der die Namen aller nicht von Kursen genutzten Veranstaltungsorte liefert.
4. Die Titel der Kurse mit der Höchstzahl von Kursabenden sollen ermittelt werden. Formulieren Sie einen SQL-Befehl, der eine Liste mit diesen Titeln produziert.
5. Formulieren Sie einen SQL-Befehl, der eine Liste der Namen aller Kurse erstellt, für die noch kein Leiter bestimmt ist.

6. Es soll eine Liste erstellt werden, die für jeden Kurs folgende Angaben enthält: Den Kurstitel und die Anzahl der Abende pro Woche, an denen der betreffende Kurs stattfindet. Geben Sie einen SQL-Befehl an, der dies leistet.
7. Es soll die Anzahl der Kurse ermittelt werden, die bei einer angenommenen Teilnehmerzahl von 10 Personen höchstens 150 EUR Kursgebühr pro Teilnehmer kosten. Formulieren Sie einen SQL-Befehl, der diese Anzahl berechnet.
8. Für die eventuelle Bildung von Fahrgemeinschaften unter Kursteilnehmern soll eine SQL-Abfrage formuliert werden, die aus den Paaren von Teilnehmernamen (Nachname und Vorname) besteht, die sich für den gleichen Kurs angemeldet haben. Es soll kein Paar in der Liste doppelt vorkommen. Mit den Inhalten der Beispieltabellen sollte die Ausgabe des SQL-Befehls etwa lauten:

Öllinger	Matthias	Haas	Bernhard
Ilg	Sabine	Haas	Bernhard

zu Abschnitt 3.5

9. Es soll eine View mit Namen KONSTANT erstellt werden, die die Namen aller Kurse enthält, bei denen die Gebühren *nicht* von der Teilnehmerzahl abhängen (d. h. für die Preisgruppe des Kurses gibt es nur einen passenden Datensatz in der Tabelle GEBUEHREN). Geben Sie den SQL-Befehl zur Erstellung dieser View an.
10. Eine Liste der Kursnamen, bei denen die Mindestteilnehmerzahl (laut den vorliegenden Anmeldungen) nicht erreicht wurde, soll erstellt werden. Geben Sie einen SQL-Befehl an, der dies leistet. (Hinweis: Es empfiehlt sich, zunächst geeignete Hilfsviews zu definieren und diese dann in der eigentlichen Abfrage zu verwenden.)
11. Es soll eine Liste erstellt werden, die für jeden Teilnehmer, der mindestens einen Kurs belegt, Nachname, Vorname, sowie die Summe seiner Teilnahmegebühren enthält. Geben Sie einen SQL-Befehl an, der diese Liste liefert. (Hinweis: Es empfiehlt sich, zunächst eine Hilfsview zu definieren, die jeweils die Kursnummer und die Zahl der für den betreffenden Kurs vorliegenden Anmeldungen enthält.)
12. Die Gesamteinnahmen für alle in der Kaufmännischen Berufsschule stattfindenden Kurse sollen berechnet werden, wobei angenommen wird, dass diese Kurse alle voll belegt sind (d. h. es sind so viele Teilnehmer angemeldet, wie maximal zulässig). Geben Sie einen

entsprechenden SQL-Befehl an. (Hinweis: Legen Sie zunächst eine Hilfsview an, die für jeden Kurs jeweils Kursnummer und maximale Teilnehmerzahl enthält.)

zu Abschnitt 3.6

13. Herr Kraus aus Vilshofen, der bereits als Teilnehmer registriert ist, meldet sich für den Kurs “Englisch Express I” an. Mit welchem SQL-Befehl kann die Datenbank entsprechend aktualisiert werden?
14. Für den Kurs “EDV-Grundlagen”, der jeweils dienstags und donnerstags stattfindet, wurde als Leiter Herr Schuster bestimmt. Formulieren Sie einen SQL-Befehl, der die KURSE-Tabelle entsprechend diesem Sachverhalt aktualisiert.

zu Abschnitt 4.1/4.2

15. Erläutern Sie kurz, warum eine Ersetzung der Tabellen KURSE und TERMINE in eine zusammengefasste Tabelle KURSDATEN (knr, titel, leiter, id, abende, dauer, pg, beginn, tag, zeit) zu Anomalien in der Datenbank führen kann.

zu Abschnitt 4.4

16. Definieren Sie sämtliche Abhängigkeits-(FOREIGN KEY) Constraints für die Tabellen der gegebenen Datenbank.

zu Abschnitt 6

17. Es soll für einen Teilnehmer nicht möglich sein, sich für einen Kurs anzumelden, wenn er bereits einen anderen Kurs belegt hat, der am gleichen Wochentag wie der neu zu belegende Kurs stattfindet. Definieren Sie einen Trigger CHECK_ANM auf der Tabelle ANMELDUNGEN, der sicherstellt, dass Neuanmeldungen nicht möglich sind, wenn oben beschriebene Situation eintritt. (Hinweise: Verwenden Sie einen BEFORE INSERT-Zeilentrigger. Sie dürfen annehmen, dass eine Tabelle TAGE(tag CHAR(2)) in der Datenbank existiert, die Sie beliebig verändern und für die Ablage von Zwischenresultaten nutzen können.)

E.3.2. Lösungen

Man beachte die allgemeinen Lösungshinweise am Anfang von Abschnitt E.1.2. Auch in diesem Abschnitt halten wir die Lösungsskizzen etwas kürzer als bei der ersten Musterdatenbank, da wir davon ausgehen, dass der Leser nun schon etwas Praxis bei der Bearbeitung von Datenbankaufgaben besitzt.

1. Aus den Beschreibungen der Tabellen einfach zu entnehmen sind Schlüssel für die Tabellen ORTE ($\{id\}$), KURSE ($\{knr\}$) und TEILNEHMER ($\{tnr\}$), die wir auch als Primärschlüssel auszeichnen.

Bei der Tabelle ANMELDUNGEN müssen offensichtlich beide Attribute als Schlüssel $\{tnr, knr\}$ verwendet werden, da sich ein Teilnehmer ja für mehrere Kurse anmelden kann. Bei der Tabelle TERMINE überlegen wir uns, dass Kurse an mehreren Wochentagen, aber laut Erläuterung nicht mit zwei Terminen am gleichen Wochentag stattfinden können. Aus diesem Grunde ist $\{knr, tag\}$ Schlüssel von TERMINE. Die eben identifizierten Schlüssel verwenden wir auch als Primärschlüssel.

Bei der Tabelle GEBUEHREN gilt es zu beachten, dass aufgrund der geforderten Widerspruchsfreiheit der Datenbank keine zwei Datensätze mit den gleichen Attributwerten für pg und tnmin existieren können, d. h. $\{pg, tnmin\}$ bildet einen Schlüssel von GEBUEHREN. Mit der gleichen Argumentation erkennt man, dass auch $\{pg, tnmax\}$ einen Schlüssel bildet. Einen dieser beiden Schlüssel können wir als Primärschlüssel auszeichnen.

Die SQL-Statements für die Definition der Primärschlüssel lauten:

```
ALTER TABLE orte ADD PRIMARY KEY (id);
ALTER TABLE kurse ADD PRIMARY KEY (knr);
ALTER TABLE teilnehmer ADD PRIMARY KEY (tnr);
ALTER TABLE anmeldungen ADD PRIMARY KEY (tnr, knr);
ALTER TABLE termine ADD PRIMARY KEY (knr, tag);
ALTER TABLE gebuehren ADD PRIMARY KEY (pg, tnmin);
```

2. Wir benötigen die Tabellen KURSE und ORTE, die wir per Natural-Join über das gemeinsame Attribut ID verknüpfen.

```
SELECT titel, name
FROM kurse NATURAL JOIN orte;
```


3. Es handelt sich hier um eine typische Negativabfrage, die wir folgendermaßen formulieren können:

```
SELECT name FROM orte
WHERE id NOT IN
  (SELECT id FROM kurse);
```

4. Offensichtlich müssen wir hier erst per Subquery die Höchstzahl von Kursabenden ermitteln, um das gewünschte Ergebnis zu produzieren:

```
SELECT titel FROM kurse
WHERE abende =
  (SELECT MAX(abende) FROM kurse);
```

5. Dass ein Kurs noch keinen Leiter hat, lässt sich daran erkennen, dass das Attribut LEITER der Tabelle KURSE den Wert NULL hat. Also:

```
SELECT titel FROM kurse
WHERE leiter IS NULL;
```

6. Die Kurstitel entnehmen wir der Tabelle KURSE, die Kursfrequenz berechnen wir aus der Tabelle TERMINE. Gemeinsames Attribut der beiden Tabellen ist knr.

```
SELECT titel, COUNT(tag)
FROM kurse NATURAL JOIN termine
GROUP BY knr, titel;
```

Für die Attribute der GROUP BY-Klausel gilt: TITEL muss aufgeführt sein, da dieses Attribut in der SELECT-Liste auftritt, ohne Argument einer Gruppenfunktion zu sein. Das eigentliche Gruppierungskriterium ist jedoch die Kursnummer, die deshalb aufgeführt ist.

7. Das gewünschte Ergebnis können wir mit den Tabellen KURSE und GEBUEHREN ermitteln. Das gemeinsame Attribut ist PG. Aus der GEBUEHREN-Tabelle muss der Datensatz verwendet werden, der für eine Teilnehmerzahl von 10 Personen gilt; das ist genau derjenige, für den gilt: Mindestteilnehmerzahl kleiner oder gleich 10, Höchstteilnehmerzahl größer oder gleich 10.

```
SELECT COUNT(knr)
FROM kurse NATURAL JOIN gebuehren
WHERE tnmin <= 10 AND tnmax >= 10
AND preis <= 150;
```

8. Hier handelt es sich um einen Self-Join mit der Tabelle TEILNEHMER. Da zu einem Teilnehmer auch die Kurse ermittelt werden müssen, zu denen er sich angemeldet hat, muss auch die ANMELDUNGEN-Tabelle in den Self-Join einbezogen werden. Wir können das so lösen, dass wir *zwei Mal* einen Natural-Join von TEILNEHMER und ANMELDUNGEN bilden und dem jeweiligen Resultat einen Tabellen-Alias (t1 bzw. t2) zuordnen. Die Resultate der beiden Natural-Joins werden dann per Equi-Join über das Attribut knr verknüpft – es interessieren ja nur Paare von Teilnehmern im gleichen Kurs. Doppelt vorkommende Paare eliminieren wir dadurch, indem wir nur Paare in der Ausgabe zulassen, wo die Teilnehmernummer des ersten Teilnehmers kleiner ist als die Teilnehmernummer des zweiten Teilnehmers.

```
SELECT t1.name, t1.vorname, t2.name, t2.vorname
FROM (teilnehmer NATURAL JOIN anmeldungen) t1
     JOIN
     (teilnehmer NATURAL JOIN anmeldungen) t2
     USING (knr)
WHERE t1.tnr < t2.tnr;
```

9. Wir ermitteln die Kurstitel aus der Tabelle KURSE und plazieren nur diejenigen Titel in der Ausgabe, für die in GEBUEHREN lediglich ein zugehöriger Datensatz existiert, was wir durch Kombination von GROUP BY- und HAVING-Klauseln erreichen:

```
CREATE VIEW konstant AS
SELECT titel
FROM kurse NATURAL JOIN gebuehren
GROUP BY knr, titel
HAVING COUNT(tnmin) = 1;
```

10. Wir erzeugen zunächst eine View, die für jeden Kurs die Mindestteilnehmerzahl liefert:

```
CREATE VIEW mint (knr, minteil) AS
SELECT knr, MIN(tnmin)
FROM kurse NATURAL JOIN gebuehren
GROUP BY knr;
```

Mit einer zweiten View bestimmen wir die Zahl der tatsächlichen Teilnehmer in jedem Kurs. Hierbei müssen wir durch einen Outer-Join dafür sorgen, dass auch Kurse geliefert werden, für die keine Anmeldungen vorliegen, also die Teilnehmerzahl gleich Null ist.

```
CREATE VIEW anm0 (knr, anzahl) AS
```

```

SELECT knr, COUNT(tnr)
FROM kurse NATURAL LEFT JOIN anmeldungen
GROUP BY knr;

```

Man beachte, dass bei COUNT in diesem Fall kein COALESCE-Konstrukt erforderlich ist, da COUNT mit einem Argument, das kein '*' ist, nur Datensätze zählt, bei denen der Wert des Arguments nicht NULL ist. Bei einem durch den Outer-Join erzeugten NULL-Datensatz ist jedoch TNR gleich NULL, wird also von COUNT nicht gezählt, so dass der richtige Wert 0 geliefert wird.

Die eigentliche Liste wird nun einfach unter Verwendung von KURSE (für die Titel) und der beiden Views erzeugt:

```

SELECT titel
FROM kurse NATURAL JOIN mint
      NATURAL JOIN anm0
WHERE anzahl < minteil;

```

11. Wir können bei dieser Aufgabe die View ANM0 aus der letzten Aufgabe verwenden. Zusätzlich benötigen wir die Tabellen ANMELDUNGEN, KURSE, GEBUEHREN und TEILNEHMER. Den jeweils für die Gebührenberechnung maßgeblichen Datensatz aus der GEBUEHREN-Tabelle ermitteln wir über die Zahl der tatsächlich vorliegenden Anmeldungen. Für den Datensatz muss gelten: Tatsächliche Anzahl größer oder gleich minimaler Teilnehmerzahl und kleiner oder gleich maximaler Teilnehmerzahl.

```

SELECT name, vorname, SUM(preis)
FROM anmeldungen NATURAL JOIN kurse
      NATURAL JOIN gebuehren
      NATURAL JOIN teilnehmer
      NATURAL JOIN anm0
WHERE anzahl >= tnmin AND anzahl <= tnmax
GROUP BY tnr, name, vorname;

```

12. Gemäß dem Hinweis in der Aufgabe legen wir zunächst eine View an, die für jeden Kurs die maximale Teilnehmerzahl angibt:

```

CREATE VIEW maxt (knr, maxteil) AS
SELECT knr, MAX(tnmax)
FROM kurse NATURAL JOIN gebuehren
GROUP BY knr;

```

Nun verwenden wir diese View und die drei Tabellen KURSE, ORTE, GEBUEHREN, um die gewünschte Auskunft zu erhalten:

```

SELECT SUM(preis * maxteil)

```

```
FROM kurse NATURAL JOIN orte
      NATURAL JOIN gebuehren
      NATURAL JOIN maxt
WHERE tnmax = maxteil
      AND name = 'Kaufmännische Berufsschule';
```

13. Um Teilnehmer- und Kursnummer zu ermitteln, müssen wir den INSERT-Befehl mit einer SELECT-Abfrage koppeln:

```
INSERT INTO anmeldungen
SELECT tnr, knr
FROM teilnehmer, kurse
WHERE titel = 'Englisch Express I'
      AND name = 'Kraus'
      AND wohnort = 'Vilshofen';
```

Man beachte, dass hier TEILNEHMER und KURSE nicht durch einen Join verbunden sind, da die WHERE-Klauseln so konstruiert sind, dass aus beiden Tabellen ohnehin jeweils nur ein Datensatz selektiert wird.

14. Wir müssen hier per Subquery herausfinden, welche Kurse jeweils dienstags und donnerstags stattfinden, um für den richtigen Kurs die Änderungen durchzuführen.

```
UPDATE kurse
SET leiter = 'Schuster'
WHERE titel = 'EDV-Grundlagen'
      AND knr IN (
      SELECT knr FROM termine
      WHERE tag = 'Di'
      INTERSECT
      SELECT knr FROM termine
      WHERE tag = 'Do'
      );
```

15. Die in KURS DATEN vorliegende (durch die Schlüsseleigenschaft von knr in KURSE bedingte) funktionale Abhängigkeit

$\{knr\} \rightarrow \{titel, leiter, id, abende, dauer, pg, beginn\}$

verletzt die 3. Normalform von KURS DATEN, da $\{knr\}$ nicht Superschlüssel von KURS DATEN und die Attribute auf der rechten Seite sämtlich nicht prim sind. Folglich ist KURS DATEN nicht in der 3. Normalform.

Daraus ergibt sich beispielsweise eine Änderungsanomalie: Wird für einen Kurs, der mehrmals pro Woche stattfindet, ein ande-

rer Leiter bestimmt, müssen mehrere Datensätze in KURSDATEN geändert werden.

16. Wir geben für jede (Kind-)Tabelle an, welche Attribute von welchen Attributen in der Eltern-Tabelle abhängen. Die Angaben zu Name und Attribut der Eltern-Tabelle stehen dabei rechts vom Pfeil.

```
ANMELDUNGEN: tnr ← TEILNEHMER.tnr
              knr ← KURSE.knr
TERMINE: knr ← KURSE.knr
KURSE: id ← ORTE.id
        pg ← GEBUEHREN.pg
```

Man beachte, dass der letzte Constraint in PostgreSQL nicht als FOREIGN KEY-Constraint definiert werden kann, denn dafür müsste pg einen PRIMARY KEY- oder zumindest einen UNIQUE-Constraint in GEBUEHREN besitzen, was jedoch aufgrund der Struktur der Tabelle nicht gegeben ist. (Man würde sich hier durch Definition eines geeigneten Triggers behelfen.)

Die SQL-Statements für die Constraintdefinitionen lauten:

```
ALTER TABLE gebuehren
  ADD PRIMARY KEY (pg, tnmin);
ALTER TABLE anmeldungen
  ADD FOREIGN KEY (tnr) REFERENCES teilnehmer(tnr);
ALTER TABLE anmeldungen
  ADD FOREIGN KEY (knr) REFERENCES kurse(knr);
ALTER TABLE termine
  ADD FOREIGN KEY (knr) REFERENCES kurse(knr);
ALTER TABLE kurse
  ADD FOREIGN KEY (id) REFERENCES orte(id);
```

17. Wir strukturieren das Problem zunächst, bevor wir an die Programmierung des Triggers gehen:

- Wir stellen zunächst fest, an welchen Wochentagen der Teilnehmer, der sich für einen neuen Kurs anmelden will, bereits einen Kurs besucht. (Diese Tage speichern wir in der Tabelle TAGE.)
- Wir ermitteln, wie viele Wochentage der neue Kurs mit bereits gebuchten Kursen gemeinsam hat.
- Ist das Resultat des letzten Schrittes größer als 0, wird die Neuanmeldung abgelehnt.

```
1  CREATE OR REPLACE FUNCTION f_check_anm ()
2  RETURNS trigger AS '
3
4  DECLARE
5      n INTEGER;
6
7  BEGIN
8
9      DELETE FROM tage;
10     INSERT INTO tage
11         SELECT tag
12         FROM termine NATURAL JOIN anmeldungen
13         WHERE tnr = new.tnr;
14
15     SELECT COUNT(tag) INTO n
16     FROM tage NATURAL JOIN termine
17     WHERE knr = new.knr;
18
19     DELETE FROM tage;
20
21     IF n > 0 THEN
22         RAISE EXCEPTION
23         'Bereits Anmeldung für gleichen Tag vorhanden!';
24     END IF;
25
26     RETURN new;
27
28 END;
29 ' LANGUAGE plpgsql;
30
31 CREATE TRIGGER check_anm
32 BEFORE INSERT ON anmeldungen
33 FOR EACH ROW
34 EXECUTE PROCEDURE f_check_anm ();
```

Abb. 133: Überprüfung von Kursanmeldungen

Betrachten wir nun das Listing in Abb. 133. Wir definieren zunächst in den Zeilen 1–21 eine Funktion `f_check_anm`, die den Trigger realisiert, und dann den eigentlichen Trigger in Zeile 22–25.

In der Funktion werden zunächst die Wochentage ermittelt, an denen der Teilnehmer bereits einen Kurs besucht (Zeile 7–10); diese werden in der Tabelle `TAGE` abgelegt. Die Anzahl der gemeinsamen Wochentage mit dem neuen Kurs berechnen wir in Zeile 11–13.

Ist diese Zahl nicht 0 (Zeile 15), gibt es gemeinsame Wochentage, d. h. der neue Kurs ist wochentagsmäßig nicht überschneidungsfrei mit den bereits belegten Kursen. Wir generieren daher eine Ausnahmebedingung (Zeile 16–17).

F. SQL-Kommandoreferenz

In der folgenden Übersicht finden Sie alle im Buch behandelten SQL-Kommandos zusammen mit der Nummer des Abschnitts, in dem das betreffende Kommando eingeführt wird.

SQL-Kommando	Abschnitt
ALTER TABLE	4.4.2
COMMIT	5.1
CREATE FUNCTION	6.1
CREATE INDEX	4.4.4
CREATE TABLE	4.4.1
CREATE TRIGGER	6.4
CREATE VIEW	3.5
DELETE FROM	3.6.3
DROP FUNCTION	6.2
DROP INDEX	4.4.4
DROP TABLE	4.4.1
DROP TRIGGER	6.4
DROP VIEW	3.5
GRANT	5.4
INSERT INTO	3.6.1
LOCK TABLE	5.2
PERFORM	6.3.5
REVOKE	5.4
ROLLBACK	5.1
SELECT	3.4
... CONNECT BY ...	6.3.5
... EXISTS ...	3.4.7
... FOR UPDATE ...	5.2
... GROUP BY ...	3.4.3
... ORDER BY ...	3.4.4
SELECT ... INTO	6.3.1
UPDATE	3.6.2

Abbildungsverzeichnis

Abb. 1:	Zugriff auf Daten im klassischen Dateisystem	2
Abb. 2:	Zugriff auf Daten durch ein DBMS	3
Abb. 3:	Vereinfachtes Modell einer Datenbank-Umgebung	4
Abb. 4:	Drei-Ebenen-Modell einer DBMS-Architektur	6
Abb. 5:	Tabellendarstellung einer Relation	8
Abb. 6:	Darstellung von Relationship-Typen in Entity-Relationship-Diagrammen	10
Abb. 7:	Beispiel für eine durch ein Entity-Relationship-Diagramm dargestellte Datenbank	11
Abb. 8:	Konkretes Beispiel einer kleinen Datenbank (Musterdatenbank FIRMA)	13
Abb. 9:	Konstruktion eines Equi-Join	16
Abb. 10:	Ergebnis eines Equi-Join	16
Abb. 11:	Konstruktion eines Self-Join	17
Abb. 12:	Anwendung von Aggregationsoperationen	18
Abb. 13:	Schematischer Aufbau eines Datenbank-Servers	21
Abb. 14:	Bestandteile eines einfachen SELECT-Statements	25
Abb. 15:	Syntax für die Bezeichnung von Objekten und Objektteilen	28
Abb. 16:	Zeichenkettenvergleiche mit verschiedenen Datentypen	32
Abb. 17:	Ergebnisse von Vergleichen mit Patterns	32
Abb. 18:	Arithmetik mit Datums- und Zeitangaben	33
Abb. 19:	Syntax für Ausdrücke	34
Abb. 20:	Syntax des SELECT-Statements	39
Abb. 21:	Auswertung einer SELECT-Abfrage mit GROUP BY- und HAVING-Klausel	42
Abb. 22:	Syntax einer JOIN-Klausel	44
Abb. 23:	Konstruktion eines Equi-Join der Tabellen PERSONAL und ZUORDNUNG	46
Abb. 24:	Ergebnis des Equi-Join mit expliziter Join-Bedingung	47
Abb. 25:	Ergebnis des Equi-Join mit USING bzw. NATURAL JOIN	47
Abb. 26:	Konstruktion eines Equi-Join aus drei Tabellen	49
Abb. 27:	Ergebnistabelle eines Inner Self-Join	50

Abb. 28:	Ergebnistabellen von Outer-Joins	51
Abb. 29:	Inner- und Outer Join mit mehreren Tabellen	52
Abb. 30:	Konstruktion eines Outer-Join mit Aggregationsoperationen	52
Abb. 31:	Join mit Positivabfrage	56
Abb. 32:	Syntax des CREATE VIEW-Statements	59
Abb. 33:	Syntax des INSERT INTO-Statements	60
Abb. 34:	Syntax des UPDATE-Statements	62
Abb. 35:	Syntax des DELETE FROM-Statements	64
Abb. 36:	Syntax des CREATE RULE-Statements	65
Abb. 37:	Eine Relation mit Anomalien	74
Abb. 38:	Eine Relation mit nichtatomaren Attributen	77
Abb. 39:	Relation nach Überführung in erste Normalform	78
Abb. 40:	Abhängigkeiten im Relationenschema KINO	79
Abb. 41:	KINO nach Transformation in zweite Normalform	80
Abb. 42:	KINO1 und KINO2 nach Transformation in dritte Normalform	81
Abb. 43:	Relation mit Anomalien und Abhängigkeiten	83
Abb. 44:	Relationen mit Anomalien in dritter Normalform	83
Abb. 45:	Relationen ohne Änderungs- und Löschanomalie	83
Abb. 46:	Syntax des CREATE TABLE-Statements	84
Abb. 47:	Syntax des ALTER TABLE-Statements	86
Abb. 48:	Syntax einer Tabellenconstraintdefinition	89
Abb. 49:	Syntax einer Spaltenconstraintdefinition	90
Abb. 50:	Struktur eines Index	94
Abb. 51:	Syntax des CREATE INDEX-Statements	95
Abb. 52:	Beispiel für eine Transaktion	97
Abb. 53:	Eine typische Deadlock-Situation	99
Abb. 54:	Transaktionsmodi im SQL-Standard	99
Abb. 55:	Syntax des START TRANSACTION-Statements	100
Abb. 56:	Syntax der FOR UPDATE-Klausel	100
Abb. 57:	Ablauf von zwei konkurrierenden Transaktionen	103
Abb. 58:	Syntax des SET CONSTRAINTS-Statements	104
Abb. 59:	Syntax des GRANT-Statements	106
Abb. 60:	Syntax des REVOKE-Statements	107

Abb. 61:	Erzeugen von Views zum “Verstecken” von Spalten	108
Abb. 62:	Datenschutz durch selektiven Tabellenzugriff über Views	110
Abb. 63:	Syntax des CREATE FUNCTION-Statements	113
Abb. 64:	Syntax des DROP FUNCTION-Statements	115
Abb. 65:	PL/pgSQL-Programm zur Gehaltserhöhung für einen Mitarbeiter	117
Abb. 66:	Arbeitsweise eines PL/pgSQL-Cursors	120
Abb. 67:	PL/pgSQL-Programm zur Gehaltserhöhung für alle Mitarbeiter	121
Abb. 68:	Benutzung eines Cursors mit OPEN ... FETCH ... CLOSE	123
Abb. 69:	Implizite Benutzung eines Cursors mit einer FOR-Schleife	124
Abb. 70:	Programm mit Ausnahmebedingungen	125
Abb. 71:	Hierarchische Struktur in der PERSONAL-Tabelle	126
Abb. 72:	Bestimmung der Hierarchiestufe eines Mitarbeiters	127
Abb. 73:	Rekursive Aufrufstruktur bei der Bestimmung der Hierarchiestufe	128
Abb. 74:	Depth first-Ordnung in der PERSONAL-Tabelle	129
Abb. 75:	Bestimmung der Hierarchiestufe für alle Mitarbeiter	130
Abb. 76:	Bestimmung der Zahl der Untergebenen eines Mitarbeiters	132
Abb. 77:	Eine Stückliste als Relation und als hierarchische Struktur	132
Abb. 78:	Rekursive Funktion zur Teilebedarfsbestimmung	133
Abb. 79:	Abarbeitung der hierarchischen Struktur mit numcomp	134
Abb. 80:	Syntax des CREATE TRIGGER-Statements	136
Abb. 81:	Definition eines Triggers zur Zuordnungsbeschränkung	138
Abb. 82:	Trigger zur Erstellung eines Änderungsprotokolls	140
Abb. 83:	Vom Trigger geschriebene Protokolldatensätze	140
Abb. 84:	Konsistenzsicherung mit einem Trigger	141
Abb. 85:	Syntax von Embedded SQL-Statements	146
Abb. 86:	Ein C-Programm mit ECPG-Konstrukten	148
Abb. 87:	Ablauf eines Datenbankzugriffs mit DBI	151
Abb. 88:	Ein einfacher Datenbankzugriff mit DBI	152

Abb. 89:	Übergabe von Daten an eine SQL-Anweisung mit DBI	156
Abb. 90:	Ein komplettes Programm in Perl/DBI	158
Abb. 91:	Programmgesteuerter Ablauf einer Oberfläche	163
Abb. 92:	Ereignisgesteuerter Ablauf einer Oberfläche	163
Abb. 93:	Fat Client und Datenbank-Server	164
Abb. 94:	Thin Client, Applikations- und Datenbank-Server	164
Abb. 95:	X-Terminal, Applikations- und Datenbank-Server	165
Abb. 96:	Ablauf der Ausführung eines CGI-Skripts	167
Abb. 97:	Datenbankzugriff aus dem WWW mit CGI und DBI	168
Abb. 98:	Web-Formular zum Eintragen eines neuen Mitarbeiters	170
Abb. 99:	HTML-Code zum Erzeugen des Web-Formulars	170
Abb. 100:	CGI-Skript zum Eintragen eines neuen Mitarbeiters	172
Abb. 101:	Ergebnisseite des Skripts bei nicht existenter Abteilung	174
Abb. 102:	CGI-Skript zum Erzeugen des Web-Formulars	175
Abb. 103:	Fehlerprüfung bei Web-Formularen	176
Abb. 104:	Ablauf der Korrektur einer fehlerbehafteten Eingabe	179
Abb. 105:	Web-Formular mit Fehlerbehandlung (Anfang)	180
Abb. 106:	Web-Formular mit Fehlerbehandlung (Fortsetzung)	181
Abb. 107:	Bearbeitung von Requests durch vier WWW-Serverprozesse	186
Abb. 108:	Über mehrere Formulare verteilte Transaktion	187
Abb. 109:	Nicht komplettierte Transaktion	188
Abb. 110:	Fast gleichzeitiger Zugriff auf ein CGI-Skript durch zwei Benutzer	188
Abb. 111:	Konsequenzen einer "race condition"	190
Abb. 112:	"Mehrbenutzerfähiges" Web-Formular mit Fehlerbehandlung	191
Abb. 113:	Konkurrierende Transaktionen bei Web-Formularen	192
Abb. 114:	Eine <code>psql</code> -Beispielsitzung	195
Abb. 115:	Datenbankadministration mit <code>phpPgAdmin</code>	196
Abb. 116:	Freie SQL-Statements in <code>phpPgAdmin</code>	197
Abb. 117:	Datenbankabfrage nach dem "Query by example"-Prinzip	198
Abb. 118:	Musterdatenbank FIRMA in <code>pgAdmin III</code>	199

Abb. 119:	Definition einer Triggerfunktion in pgAdmin III	200
Abb. 120:	Definition eines Triggers in pgAdmin III	201
Abb. 121:	Syntaxdiagramm ‘Ziffer’	203
Abb. 122:	Syntaxdiagramm ‘Gleitpunktzahl’	203
Abb. 123:	Zulässige Wege im Syntaxdiagramm	204
Abb. 124:	Die Tabellen der Musterdatenbank THEATER	213
Abb. 125:	Platzplan von ‘Mozarthaus, Saal K’	218
Abb. 126:	Reservierung eines Platzes	225
Abb. 127:	Ermittlung eines freien Platzes	226
Abb. 128:	Die Tabellen der Musterdatenbank HOTEL	228
Abb. 129:	Löschen von “Karteileichen”	238
Abb. 130:	Löschen von “Karteileichen”, verbesserte Version	239
Abb. 131:	Abrechnung erfolgter Aufenthalte	241
Abb. 132:	Die Tabellen der Musterdatenbank VHS	243
Abb. 133:	Überprüfung von Kursanmeldungen	253

Stichwortverzeichnis

SQL-Kommandos wurden aus Gründen der Übersichtlichkeit nicht in das Stichwortverzeichnis aufgenommen.

Sie finden eine Übersicht aller besprochenen SQL-Kommandos mit Verweisen auf die betreffenden Buchabschnitte in Anhang F.

A

Abfrage 5, 20, 25, 38, 40–45, 47, 48, 51, 54–57, 59, 61, 63–67, 69, 72, 73, 85, 93, 98, 101, 114, 115, 118, 120, 122, 123, 129, 153, 157, 159, 194, 197, 214, 215, 217, 221, 227, 234, 236, 245, 251
 -resultat 119, 120
 -sprache 6, 22, 23
Abhängigkeit 8, 9, 75, 76, 78–80, 82, 83, 216, 223, 231, 237, 246, 251
Abhängigkeitssystem 76, 78
Abstraktionsniveau 20, 151
Administrationsoperation 198
Adminitrationstools 198
Änderungsabfrage 65, 100
Änderungsoperation 139
Aggregation 26, 41
Aggregationsfunktion 18, 34, 36, 41, 42, 62
Aggregationsoperation 25, 51

Aktion 66, 105, 135, 146, 147, 152, 169, 176, 177, 183, 185, 238
Aktionsskript 176, 177, 183
Aktualisierung 156, 208, 231
Alias 50, 118, 249
Anomalie 12, 74, 79, 81, 246
 Änderungs- 74, 79, 80, 83, 251
Antwortzeit 81
Anweisungskategorie 27
Anweisungsteil 116, 118
Anwendungsentwickler 24
Anwendungsentwicklung 5
Anwendungsgebiet 139
Anwendungsprogramm 1, 2
Apache-Webserver 186, 193, 196, 206
Applikationsserver 165
Attribut 7–9, 11, 13, 16, 17, 25, 29, 34, 41, 46, 48, 55, 73, 75–80, 126, 127, 139, 170, 171, 188, 217–221, 223, 234, 235, 237, 242, 247–249, 251, 252
 -menge 7, 8, 75, 82, 91
 -name 11, 12, 15, 235
 -wert 7, 8, 13–15, 17–19, 74, 75, 77, 128, 131, 139, 151, 232, 242, 247
Aufruf 23, 113, 125, 129, 134, 143, 147, 154, 156, 160, 167, 173, 177, 183–187, 190
Aufrufstruktur 128
Augmentierung 75

Ausdruck 25, 30–33, 36–38, 40,
41, 43, 45, 58, 112, 118,
229, 232

Ausführungsgeschwindigkeit 93

Ausführungsreihenfolge 48

Ausgabe-Hostvariable 144

Ausgangs-Relationenschema 77,
82

Ausgangsrelation 18, 77

Ausnahmebedingung 123, 125,
137, 149, 253

Autocommit 97, 102, 155, 190

B

B-Baum 93, 94

Backup 5

Basistabelle 58, 62, 65–67, 107

Bedingung 13, 15–17, 25, 38, 43,
45, 47, 52, 56, 64, 66, 67,
79, 88, 98, 129, 153, 217,
219, 233, 240

Befehl 19, 25, 86, 109, 212, 214–
217, 219, 222, 223, 230,
231, 236, 237, 244–246,
251

Benchmarktests 5

Benutzer

-aktion 162

-daten 19

-eingabe 157, 167, 169, 177,
183, 187

-kennung 139, 210

-name 28, 105, 149, 152

-oberfläche 6, 162, 164–166,
194

-prozess 21

-schnittstelle 24

-verwaltung 104

-zugriff 21

Berechtigung 71

Betriebssystem 28, 150, 162, 165,
198, 205, 206

Betriebssystemplattform 1, 150,
161

Bibliotheksfunktion 143

Binderegeln 31, 33

Block 116

Boyce-Codd-Normalform
(BCNF) 81

Browser 166–168, 171, 173, 174,
183, 185, 195, 196

C

C

-Datentyp 144

-Funktion 147

-Programm 143–145, 147

-Variable 145

CGI 166–178, 182, 184–190, 193,
196, 206

-Skript 169, 188

Client 21, 163–166, 195, 196

-bibliothek 151, 160

-software 151, 164

Codierung 193

Command Line Interface (CLI)
162, 164, 165

Compiler 143

Compilierungsaufwand 160

Compilierungsphase 186

Compilierungsschritt 150

Constraint 84, 87–93, 95, 102,
104, 106, 135, 139, 141,
216, 223, 231, 238, 246,
252

-definition 88, 104, 237, 252

Core-SQL 24, 35

Cursor 117, 120–122, 176, 227

-prinzip 120, 153

D

Datei 1–3, 5, 167, 171, 177, 210,
211

-namen 171

-system 3

-verwaltung 1

Daten

-abstraktion 3

-änderung 1

-austausch 166

-basis 1

-bestand 2, 92, 96, 242

-ebene 3

-eingabe 5

-haltung 3

-inkonsistenz 96

-integrität 5

-konsistenz 99

-manipulation 5

-modell 6

-modifikationsbefehl 93

-sicherheit 3, 5

-speicherung 1

-struktur 58, 93

-typ 23, 29–33, 36, 37, 40,
68, 71, 84, 85, 113, 114,
116, 118, 137, 144, 212

-wert 85

-übertragung 175

Datenbank 1–3, 5, 6, 9, 10, 12,
19–25, 27, 28, 31, 37, 46,
59, 61, 68, 73, 81, 84,
87, 96, 97, 100, 104, 105,
107, 109, 111, 112, 116,
129, 135, 141, 143, 145,
147, 149–153, 155–157,
159–164, 167–169, 171,
177, 178, 184–187, 190,
192–198, 201, 205–207,
210–212, 214, 216, 217,
227, 230, 231, 235, 238,
239, 242, 244, 246, 247

-Aufgaben 232, 247

-Clientsoftware 164

-Inhalt 68, 71, 217

-Installation 116, 196

-Sprache 19, 143

-Struktur 194

-abfrage 34, 58, 81, 93, 184,
196, 217

-administration 194

-administrator 5, 6, 116, 194

-anbindung 161, 164

-anwendung 143

-applikation 5, 6, 177

-architektur 5

-betrieb 96

-definition 196

-designer 24

-ebene 161

- implementierung 143
- integrität 20
- katalog 19
- objekt 19, 26, 59, 194
- operation 24, 25, 96
- programmierer 162
- programmierung 58
- sicherheit 142
- system 1, 19–23, 37, 47, 62, 68, 69, 77, 92, 93, 95, 98, 99, 104, 116, 143, 145, 150–155, 157, 159–161, 163, 194, 205, 207, 209, 238
- tabelle 93
- theorie 9
- verbindung 155, 184–186
- verwalter 105
- verwaltungssystem 1
- zugriff 3, 150, 151, 154, 157
- Datensatz 13–15, 18–20, 25, 38, 40–43, 45, 49, 50, 54–57, 60–64, 66, 67, 74, 77, 86–89, 94, 98–101, 110, 118–120, 122, 126, 128, 129, 132, 134, 135, 137–139, 146, 147, 153, 154, 157, 159, 174, 188, 189, 216–220, 222, 225, 227, 229, 232–234, 237, 239, 240, 242, 245, 247–252, 263
- bearbeitung 122
- inhalt 138
- Datensatz 41
- Datenschutz 58, 107
 - aufgaben 5
 - konzept 104
- Datumsformat 72
- DBD 150–152, 154, 160
- DBI 150–152, 154, 155, 157, 159–161, 168, 171, 173, 186, 190, 206
- DBMS
 - Applikation 3
- DCL 19, 27, 96
- DDL 19, 26, 100, 217
- Deadlock 99
 - Situation 99
- Deklaration 116, 122
- Deklarationsteil 116, 118, 120
- Dekomposition 82–84, 223, 237
- Dekompositionsoperation 82, 83
- Dezimalpunkt 29, 35, 70
- Dezimalstellen 29, 30
- Dialog 199
- DML 19, 26, 96
- Domain-Key-Normalform (DKNF) 81
- Download 205, 206
- Drei-Ebenen-Modell 5
- E**
- ECPG 143, 145–147
- Effizienz 48, 185
- Einbettung 166
- Einfüge
 - operation 61, 157, 174
- Einfügen 60, 61, 85, 92, 135, 139, 157, 177, 178, 182, 184, 189

- Eingabe
 - aufforderung 157
 - daten 216
 - feld 170, 171, 176, 178, 182
 - formular 175
 - maske 177
- Einzelabfrage 40
- Eltern-Datensatz 126
- Eltern-Kind-Beziehung 126
- Eltern-Tabelle 88, 89, 106, 223, 237, 252
- Embedded SQL (ESQL) 143
- Entity 9–11
 - typ 10–12
- Entwicklungsprozess 193
- Entwicklungstool 163, 193
- Entwicklungsumgebung 193
- Entwicklungswerkzeug 162
- Entwurf 5, 9, 24, 46, 73, 84
- Ergebnis
 - liste 114, 234
 - relation 14, 15, 17, 18
 - tabelle 15, 16, 25, 26, 38, 40–46, 48–51, 56–58, 61, 85, 114, 115, 120, 122, 153, 159, 235, 240
 - zeile 63
- Ersatzdarstellung 119
- Exception 123, 125

- F**
- Fehler
 - analyse 184
 - ausgabe 173
 - behandlung 146, 155, 157, 174, 176–178, 184
 - code 123
 - freiheit 176
 - kontrolle 159, 177, 182
 - korrektur 208
 - meldung 89, 92, 102, 125, 138, 141, 149, 154, 173, 182–184
 - prüfung 153–155, 176
 - quelle 189
 - situation 123, 153, 154
 - suche 173
- Festkommazahl 81
- Festplatte 2
- Festplattenspeicher 21
- Festpunktzahl 29
- Foreign-Key-Constraint 91
- Format
 - string 69, 70
 - zeichen 69–71
 - zeichenkette 69
- Formular 166–171, 173–178, 182–185, 187, 189
 - bezeichnung 183
 - daten 170
 - eingabe 166, 169, 171, 184
 - element 169, 171
 - erzeugung 175
 - feld 173, 178, 182, 183
 - seite 174, 175
- Funktion 4, 9, 18, 19, 27, 34–36, 59, 70, 112–118, 122–125, 127–129, 131, 133,

136–138, 147, 156, 170,
173, 178, 182, 185, 194,
198, 199, 212, 216, 219,
224, 227, 232, 235, 238,
239, 242, 253
funktional 8, 9, 75, 76, 78–80, 82,
83, 223, 251
Funktionalität 42, 155, 190, 194
Funktionsaufruf 172, 174, 185
Funktionsname 129
Funktionsparameter 114, 118

G

Gateway 166
Genauigkeit 29, 30
Gleitpunktzahl 29, 30, 203
Grantee 105–107
Grantor 105
Graphical User Interface (GUI)
162–165, 176
Gruppenfunktion 36, 41, 42, 234,
248
Gruppierungsattribut 18, 19
Gruppierungskriterium 248

H

Handle 152–155, 178
Hash 173, 178, 182–184
Hauptabfrage 54
Hauptprogramm 183
Hierarchie 126, 128
-stufe 127–129, 131
Hilfsview 58, 215, 221, 231, 234,
235, 245, 246
Host

-Applikation 22
-Programm 143, 161
-Programmiersprache 22,
143
-variable 143–145, 147, 149,
155
HTML 166–169, 173–175, 182,
183, 185, 196
-Tag 169
HTTP 166

I

Indikatorvariable 144, 145
Indizes 7, 27, 93, 95
Inferenzregel 75, 76
Information 1, 3, 4, 6, 7, 19, 20,
24, 37, 51, 56, 58, 68,
69, 73, 74, 76, 81, 84, 93,
111, 118, 155, 182, 186,
205, 206, 212, 217, 219,
224, 233, 244
Informations-
-regel 19
-system 1
Initialisierung 122, 129
Inkonsistenz 1, 3
Instanz 102, 124, 184, 188, 189
Integrität 87
Integritäts-
-bedingung 5, 84, 87, 189
-regel 19
-unabhängigkeit 19
-verletzung 239
Interface 24, 166

Interpreter 151, 171, 184–186,
206

Iterationskriterium 159

J

Java 23

JavaScript 166

Join 15–17, 26, 38, 42–53, 55–57,
62, 81, 82, 217, 219–221,
233–236, 247, 249–251

Equi- 15, 16, 26, 45–47, 56,
235, 249

Natural- 16, 47, 48, 82, 219,
233–235, 247, 249

Outer- 49, 51, 220, 221, 234,
236, 249, 250

Self- 16, 26, 38, 50, 249

K

Kind-Tabelle 88

Kommandozeilen
-tool 194, 196

Kommandozeilen-
-ebene 165
-programm 189

Konformität 23

Konformitäts-
-ebene 23

Konkatenationsoperator 31

Konsistenz 1, 81, 96, 141, 176
-problem 101
-prüfung 3

L

Laufzeitsystem 151

Leseoperation 1, 24, 98

Linux 205, 206

Literal 28, 29, 113, 124

Löschanomalie 74, 83

Löschoperation 236

M

Mehrbenutzer

-system 28

Mehrbenutzer-

-betrieb 1, 5, 98, 185

-umgebung 102, 142, 190

Mengen-

-operator 40, 41, 62

-orientierung 19

Menüpunkt 162

Metadaten 3, 111

N

Nationalsprachenunterstützung
5

Negativabfrage 57, 233, 248

Netzwerk 165

-anbindung 165

-fähigkeit 21

-prozess 21

Nichtlinearität 162

Nichtschlüsselattribut 80

Normalform 77–83, 216, 223, 231,
251

Normalisierung 75, 79, 81, 84

NULL

-Attributwert 17, 128

-Constraint 87, 91

- Datensatz 250
- Operator 33
- Wert 17, 49, 50, 61, 67, 74, 75, 87, 88, 91, 144, 242
- Nullbyte 144
- O**
- Objektprivileg 105, 106
- Open Source 150, 192, 193, 205–207
- Operand 31, 33
- Operator 14, 15, 31–33, 37, 55–57, 62
- Overloading 115, 129
- P**
- Package 23
- Parameter 113–115, 117, 124, 127, 129, 131, 137, 145, 152, 156, 173, 174, 178, 183–185, 199, 216, 223, 224, 232
 - name 173
 - wert 169
- Perl 143, 149–151, 153, 155–157, 159–161, 167, 168, 171, 173, 183–186, 193, 206
- PHP 166, 196
- PL/pgSQL 112, 116–118, 120, 122, 123, 129, 135, 137, 143, 153, 212, 216, 224, 232, 237, 239
- Portierbarkeit 161, 193
- Portierungs-
 - aufwand 150, 161
 - fragen 160
 - schwierigkeit 161
- Positivabfrage 55, 57
- PostgreSQL 1, 23, 24, 27, 29, 30, 33, 62, 65, 66, 69, 71, 72, 88, 90, 96, 97, 99–102, 105, 106, 111, 112, 115, 116, 135, 136, 142, 143, 145, 151, 152, 155, 194, 196, 201, 205–207, 209–211, 238, 252
 - Superuser 211
- Precompiler 143, 147, 160
- Primärschlüssel 9, 11, 87, 90, 91, 188, 214, 217–219, 230, 232, 233, 244, 247
 - eigenschaft 188
 - wert 19
- Prioritätsregel 99
- Privileg 28, 104–107, 109, 111, 113, 142, 210
- Programm 1, 2, 4, 5, 21, 23, 116, 118, 121, 122, 135, 143–147, 149–151, 153–157, 159–162, 166, 167, 171, 224, 225, 237, 239
 - abbruch 147, 154
 - ablauf 150
 - ausführung 149
 - code 159, 166, 190
 - erstellungprozess 160
 - listing 208, 240
 - logik 193
 - system 1
 - teil 162
 - version 189
- Programmier-

- aufwand 1, 3
- ebene 3
- sprache 22–24, 30, 33, 112, 116, 117, 143, 149, 150, 159, 160, 167
- Programmierung 99, 163, 164, 166, 167, 252
- Protokolldatei 173
- Protokollierung 96, 139
- Pseudo-Datentyp 144

- Q**
- Quellcode 147, 150, 160
- Query 22, 122, 134, 197

- R**
- race condition 189
- Reaktionszeit 5
- Redundanz 1, 4, 81
 - freiheit 4
- Reflexivität 75
- rekursiv 127, 128, 131, 133, 134
- Relation 7–9, 12–17, 20, 22, 73–78, 80, 82, 83
- Relationen-
 - form 73
 - modell 74, 75
 - schema 8, 11, 12, 73–84, 131
- Relationship 10–12
- Relationsname 15, 73
- Repräsentation 68, 69
- Rollback 187
- Rückgabetypp 113, 114
- Rückgabewert 113–115, 125, 133, 157, 159
- Rule-System 66

- S**
- Schema 28, 73, 105, 111, 217
- Schleifenbedingung 154
- Schleifenvariable 178
- Schlüssel 9, 27, 74, 75, 77–80, 182, 217–219, 232, 233, 247
 - attribut 12
 - eigenschaft 9, 77, 251
 - wert 75
- Schlüsselwort 45, 50, 66, 116, 145
- Schlüssel 173
- Schnittstelle 20, 150, 151, 160, 172
- Schreib-/Lese-Transaktion 100
- Schreiboperation 3
- Schreibzugriff 224
- Semantik 67, 76, 138
- Server 20–22, 97, 145, 152, 164–167, 169, 171, 173, 185, 186, 193, 196, 206
- Serverprozess 21, 167, 186, 187
- Sicherheit 5, 20, 173
- Sicherheits-
 - aspekt 169
 - kontext 113, 114
 - lücke 166
 - mechanismus 139
- Skript
 - aufruf 186
 - code 189
 - programmierung 187

- Software 20, 150, 165, 192, 194–196, 205, 206, 208
 - entwicklung 178
 - hersteller 193
 - paket 205
 - produkt 5, 193
 - umgebung 4
- Sonderzeichen 203
- Sortierkriterium 220, 230
- Spalten
 - constraint 88
 - definition 85
 - liste 60, 61
 - menge 87, 88, 91
 - name 19, 38, 46–48
 - zugriff 108
- Speicherverwaltung 20
- Sperre 98–102, 189, 190, 224
 - Exklusiv- 98, 102, 189, 224
 - Lese- 98, 100–102
- SQL 1, 6, 20, 22–27, 29–31, 33–35, 37, 41, 43, 45, 52, 58, 62, 65, 66, 69, 71, 72, 84, 86–88, 90, 91, 95–97, 99–102, 105, 106, 111–118, 120, 122–124, 129, 135–137, 139, 142–147, 149, 151–153, 155–157, 159–162, 178, 194, 196, 197, 201, 205–212, 214–217, 219, 223, 224, 230–233, 237–239, 244–247, 252
 - Datentyp 116, 118
 - Dialekt 209
 - Implementierung 143
 - Programm 116
 - Standard 23, 27, 143, 147, 159, 160
 - Statement 66, 112, 114, 197
 - Worksheet 197
- Standardwert 61, 84, 85, 171
- Statement 24, 25, 27, 37, 40–42, 52, 54, 58, 66, 68, 71, 84, 85, 88–90, 96, 97, 100–102, 104, 105, 112, 114, 116, 118, 124, 129, 137, 139, 141, 143–147, 149, 152–155, 178, 194, 197, 201, 209–211, 214, 217, 222, 230, 231, 233, 238, 240, 244, 247, 252
- String 7, 125, 144
- Stringliteral 72, 113, 137
- Subquery 52–56, 60–63, 222, 236, 248, 251
 - korrelierte 54–56
- Suchbegriff 93
- Superschlüssel 9, 80, 217, 251
- Superuser-Privileg 210
- Superuser-Recht 105
- Syntax 24, 25, 28, 32, 33, 37, 42, 43, 48, 65, 88, 93, 100, 104, 203
 - diagramm 37, 38, 58, 60, 84, 112, 145, 203, 204
- System
 - administration 5
 - privileg 104, 105
 - tabelle 111
 - umgebung 185, 186, 205
 - variable 118, 139

T

Tabellen-

- Alias 50, 249
- Liste 25, 54
- beschreibung 217
- constraint 87, 88, 91
- darstellung 9
- definition 60, 61, 85, 87
- eintrag 60, 105, 106
- form 8, 19, 20
- format 183
- inhalt 139, 212
- name 19, 38, 44, 48
- spalte 30, 36, 41, 45, 60, 61, 84, 106, 111, 113, 118, 183
- struktur 48, 170
- zeile 34, 106, 122
- zugriff 93, 216, 224

Telnet-Client 165

Text-

- datei 151
- feld 169, 171
- interface 162
- konstante 28
- literal 28, 29, 31

Thin Client 163, 164, 166

Transaktion 89, 90, 96–102, 104, 124, 137, 142, 145, 155, 161, 185–190, 224

Transaktions-

- ende 90, 101, 102
- kontext 97, 102, 124, 145
- kontrolle 1, 145

-konzept 96, 161

-modus 99, 100

-verarbeitung 20

Transitivität 75

Trigger 27, 106, 123, 135–139, 141, 142, 194, 198, 199, 246, 252, 253

-definition 138, 199

-funktion 137

Typ

-konversion 36, 37, 71, 144

U

Unix-System 151, 165

Unterabfrage 52–54, 63, 85

Unterprogramm 157, 182, 184

Update 19, 63, 64, 199

V

Variable 116, 118, 120, 122, 144, 145, 147, 152, 153, 159, 171, 173, 178, 182, 187, 224, 225, 227, 240

Variablenname 171

Variablenwert 169

Verfügbarkeit 5, 20, 161

Vergleichs-

-operation 31

-operator 30–33, 53

Verkettungsoperation 37

Verknüpfung 2, 31, 43, 219

Vernestung 42

View 27, 42, 58, 59, 62, 64–67, 81, 105–111, 135, 207, 215, 221, 231, 236, 245, 249, 250

- W**
- Web-
 - Formular 166–170, 176
 - Interface 24
 - Webserver 186, 193, 196
 - Website 206
 - Wurzelknoten 128, 129
 - WWW
 - Adresse 167, 170, 185
 - Anbindung 169, 187, 193, 205
 - Browser 166, 167, 173, 185, 195, 196
 - Oberfläche 177, 192
 - Seite 150, 166–169, 171, 174
 - Server 166, 167, 169, 171, 173, 185, 186, 206
 - Serverprozess 167
 - Serverumgebung 171
 - Site 208
- X**
- X-Server-Software 165
 - X-Terminal 165
- Z**
- Zahlendarstellung 68
 - Zahlenkonstante 30
 - Zahlenwert 68
 - Zeichenfolge 68
 - Zeichenkette 7, 30–32, 35, 37, 61, 68–71, 144, 147, 152, 156, 157, 161, 169, 173
 - Zeilentrigger 137, 138, 199, 246
 - Zeitintervall 72
- Zugriffsrecht 26, 28, 142
 - Zuweisung 118
 - Zwischencode 150, 185, 186