3 PostgreSQL – der Einstieg

Nachdem Sie sich durch die Datenbanktheorie und die verschiedenen Installations- und Konfigurationsprozeduren durchgearbeitet haben, wird es jetzt konkret. Die aus dem ersten Kapitel bekannte Relation `autoren` *wird nun in eine PostgreSQL-Tabelle umgesetzt. Dabei lernen Sie das interaktive Datenbankterminal (oft auch Datenbankmonitor genannt) von PostgreSQL,* `psql`*, kennen. Viele SQL-Befehle, die zum Anlegen, Ändern oder Löschen von Tabellen gebraucht werden, sowie einige PostgreSQL-Datentypen werden ebenfalls vorgestellt.*

SQL-erfahrene Leser, die PostgreSQL noch nicht kennen, sollten sich die Abschnitte über das Datenbankterminal anschauen. Außerdem enthält das Kapitel einige PostgreSQL-Erweiterungen, die nicht im SQL-Standard enthalten sind, so dass sich die Durchsicht auch für diesen Leserkreis lohnt.

3.1 Einige Konventionen zur Vergabe von Namen

Wie in allen anderen Programmiersprachen gibt es auch in SQL Konventionen, die bei der Vergabe von Namen eingehalten werden müssen.

- Namen in SQL müssen mit einem Buchstaben oder dem Unterstrich beginnen, danach dürfen Buchstaben, Zahlen und Unterstriche stehen.
- Zeichenketten müssen in Hochkommas eingeschlossen sein. SQL erlaubt auch Hochkommas innerhalb einer Zeichenkette, wenn ein zweites Hochkomma vorangestellt wird. In PostgreSQL kann man außerdem den Backslash \ als Escape-Zeichen verwenden.
- Manche Zeichen müssen durch Voranstellen eines Sonderzeichens, dessen Bedeutung genau festgelegt ist, gekennzeichnet werden, da sie sonst als Steuerzeichen oder Ähnliches interpretiert werden. Man spricht dabei vom Maskieren oder Escapen eines Zeichens.

Escape-Zeichen

Dieses Sonderzeichen wird Escape-Zeichen genannt. Als Escape-Zeichen stehen \b für Backspace, \f für Formularvorschub, \n für einen Zeilenvorschub, \r für einen Wagenrücklauf, \t für einen Tabulator zur Verfügung. Außerdem wird \xxx akzeptiert, wobei xxx die Oktalzahl eines ASCII-Zeichens ist. Alle anderen Zeichen hinter einem Backslash werden als das Zeichen selbst interpretiert. (z. B. \\ wird als Backslash interpretiert.)

- Ein Name darf nicht mehr als 31 signifikante Zeichen lang sein, alle weiteren Zeichen werden abgeschnitten.
- Namen und Schlüsselwörter sind case-insensitiv, sie können in Groß-oder Kleinbuchstaben geschrieben werden oder gemischt. Schlüsselwörter haben eine genau definierte Bedeutung in SQL. Eine Liste der SQL Key Words ist im Anhang B des PostgreSQL User's Guide enthalten

Quoted Identifiers

- Eine Zeichenkette in doppelten Anführungszeichen (quoted identifier) wird niemals als Schlüsselwort interpretiert. "Select" könnte so als Tabellenname oder Spaltenname benutzt werden, aber ohne Anführungszeichen würde es als Schlüsselwort interpretiert werden. Quoted Identifiers dürfen alle Sonderzeichen außer dem doppelten Anführungszeichen enthalten. Damit kann man beispielsweise exotische Spaltennamen vergeben, die Längenbeschränkung bleibt aber gültig. Außerdem sind diese Namen case-sensitiv, d. h. Groß-und Kleinschreibung werden unterschieden.
- Namen, die nicht in doppelten Anführungszeichen stehen, werden von PostgreSQL in Kleinbuchstaben umgewandelt.

3.2 Das Handwerkszeug: psql

Datenbanken erzeugen

Bevor irgendwelche Daten gespeichert werden können, müssen die Speicherstrukturen erzeugt werden, die die Daten aufnehmen. Jeder Benutzer, der eine PostgreSQL-Datenbank erstellen möchte, muss dazu die nötigen Rechte haben. Diese werden beim Erzeugen eines Benutzers definiert (siehe Kapitel 2). Jeder, der mit diesen Privilegien ausgestattet ist, kann eine Datenbank erzeugen, deren Besitzer er dann ist, indem er auf der Kommandozeile die Anweisung createdb mit entsprechenden Parametern absetzt. Die Parameter sind:

```
createdb -h host -p port -U user -W -e
```

Host ist der Name des Zielrechners. Wenn dieser nicht angegeben wird, wird der lokale Rechner adressiert. Mit der Option port geben Sie die Portnummer des Datenbankservers an, die ohne die Option auf den Standardwert 5432 gesetzt wird. Der user ist Ihr Benutzername. Ist die

Option nicht gesetzt, nimmt PostgreSQL den Namen des aktuell angemeldeten Benutzers an. Mit der Option –W können Sie eine Passwortabfrage erzwingen und mit –e können Sie das Kommando, das an den Datenbankserver geschickt wird, anzeigen. Ein Aufruf von

```
createdb
```

ohne Parameter erstellt eine Datenbank mit dem Namen des aktuellen Benutzers auf dem lokalen Rechner mit dem Standardport 5432. Als Beispiel-Anwendung soll hier ein Dokument-Generator entwikkelt werden, weshalb unsere Datenbank den Namen docgen bekommt.

```
createdb docgen
```

Datenbanken öffnen

Wenn die Datenbank angelegt werden konnte, quittiert PostgreSQL das mit der Meldung CREATE DATABASE. Wenn ein Fehler aufgetreten ist, beispielsweise, weil eine Datenbank mit diesem Namen bereits existiert, wird eine Fehlermeldung generiert. Diese neu erstellte Datenbank kann jetzt geöffnet werden. Grundsätzlich gilt, dass Sie psql sowohl am Linux-Systemprompt ausführen können als auch unter KDE oder Gnome in einem Terminal (auch Konsole genannt).

```
psql docgen
```

Mit diesem Kommando wird das Datenbankterminal von PostgreSQL gestartet, das die Datenbank docgen öffnet und Sie mit der folgenden Meldung begrüßt:

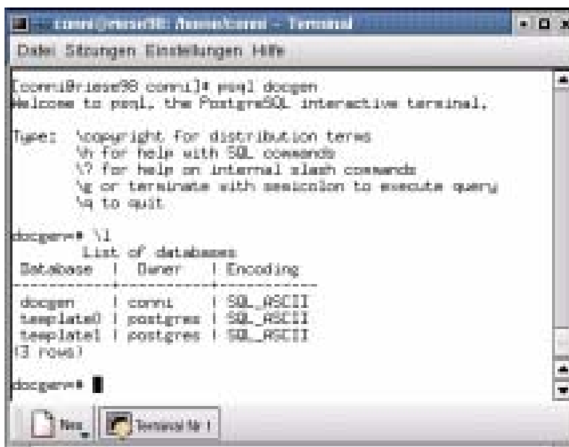

Abb. 3–1
Startmeldung von PostgreSQL

Jetzt sind Sie mit dem Datenbankserver verbunden, der mit der Eingabeaufforderung

```
docgen=#
```

auf Ihre weiteren Anweisungen wartet. Die Eingabeaufforderung oder auch Prompt zeigt standardmäßig immer zuerst den Namen der Datenbank an, gefolgt von einem Gleichheitszeichen und dem #-Zeichen, falls Sie psql als Eigentümer der Datenbank aufgerufen haben. Anderenfalls hat die Eingabeaufforderung die Form nameDerDB=>.

Backslash-Kommandos

Wie Sie in dem Begrüßungsbildschirm sehen, stellt psql Ihnen gleich beim Start einige so genannte Backslash-Kommandos zur Verfügung, mit denen Sie Hilfe aufrufen können:

Kommando	Hilfe
\h	zeigt syntaktische Hilfe zu SQL-Befehlen an
\?	listet alle Kommandos von psql auf
\g	für go schließt eine SQL-Anweisung ab, alternativ zu einem Semikolon
\q	für quit beendet psql

Für alle diese Kommandos gilt: Bis zur Version 7.1.3 von PostgreSQL werden sie nicht mit einem Semikolon abgeschlossen. Wenn Sie doch einmal aus Versehen eines anhängen, gibt PostgreSQL eine Fehlermeldung aus. Ab der Version 7.2 wird ein Semikolon akzeptiert.

Eingebaute Variablen

Bis jetzt sind weder Tabellen erstellt noch irgendwelche Daten eingetragen, es gibt aber Werte, die immer abfragbar sind: die eingebauten Variablen von PostgreSQL: CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, CURRENT_USER oder die aktuelle PostgreSQL-Version. Sie können die Werte der eingebauten Variablen mit dem SELECT-Befehl erfragen (dieser Befehl ist einer der wichtigsten Befehle in SQL und wird im Abschnitt 3.5 genauer erklärt).

```
docgen=# SELECT CURRENT_TIME;
   time
----------
 17:45:20
(1 row)
docgen=# SELECT version();
                           version
-----------------------------------------------------------
PostgreSQL 7.1.3 on i386-pc-linux-gnu, compiled by GCC 2.96
(1 row)
docgen=#
```

PostgreSQL beantwortet alle Anfragen in dem gleichen Format.

Sie geben am Prompt einen SQL-Befehl ein, der mit einem Semikolon oder mit \g (go) abgeschlossen werden muss, damit die Anfrage an den Server geschickt wird. Wenn Sie das Statement nicht abschließen,

zeigt PostgreSQL Ihnen die Eingabeaufforderung mit einem Bindestrich (oder einer geöffneten Klammer) anstelle des Gleichheitszeichens an.

```
docgen=# SELECT CURRENT_USER
docgen-# ;
 user
------
 conni
(1 row)
```

Ein Zeilenende oder das Drücken der Eingabetaste schließt die Anfrage nicht ab, so dass Sie Kommandos auf verschiedene Zeilen verteilt übersichtlich eingeben und am Ende mit einem Semikolon oder \g ausführen können. Die erste Zeile des Ergebnisses zeigt den oder die Spaltennamen an, die Sie ausgewählt haben, und unterhalb einer gestrichelten Zeile gibt PostgreSQL die Ergebnisse aus und wartet danach wieder auf Eingaben.

Abfragepuffer – Querybuffer

Alle Eingaben, die Sie in psql machen, werden zeilenweise in einen Puffer geschrieben. Mit den Pfeiltasten können Sie durch diesen Puffer blättern und so das angezeigte Kommando wiederholt ausführen, indem Sie die Eingabetaste drücken. Sie können diese so genannte History mit dem Kommando \s anzeigen oder mit \s <datei> in eine Datei schreiben.

Es ist üblich, die SQL-Befehle generell als »Abfrage« oder »Query« zu bezeichnen, ob es Einfüge- oder Anfrageoperationen sind, ist dabei unerheblich, gemeinsam ist, dass alle SQL-Befehle eine Anweisung an den Datenbankserver senden, der seinerseits eine Antwort an den Client zurückgibt. Solche Serverantworten sind Abfrageergebnisse, Fehlermeldungen, Systemmeldungen oder, wie im obigen Beispiel, der Inhalt einer Variablen.

Eine Übersicht über alle Backslash-Kommandos und Kommandozeilenparameter von psql finden Sie im Anhang.

3.3 Datenbanken erzeugen – CREATE DATABASE

Im Kapitel 2.2.1 sahen Sie eine Möglichkeit, eine Datenbank zu erstellen. Es gibt noch einen anderen Weg: Immer, wenn in SQL etwas erzeugt wird, wird das Schlüsselwort CREATE benutzt. Sie ahnen es vielleicht, CREATE ist der wichtigste Befehl der Datendefinitionssprache SQL, von der im ersten Kapitel die Rede war. Innerhalb des Datenbankmonitors kann man mit dem Kommando

```
docgen=# CREATE DATABASE dbname;
```

die neue Datenbank dbname erzeugen, der Besitzer ist der aktuelle Benutzer. Die Parameter, die diesem Kommando mitgegeben werden können, sind dieselben wie bei der Anwendung createdb, die vom Linux-Prompt aus aufgerufen wird. Dieses Kommando ist eine PostgreSQL-Erweiterung und ist nicht im SQL92-Standard enthalten.

Mit dem psql-Befehl \l (list) lassen Sie sich eine Liste aller Datenbanken anzeigen und mit \c dbname (connect) wechseln Sie die Datenbank. Die Eingabeaufforderung zeigt danach den Namen der anderen Datenbank an.

3.4 Tabellen definieren – CREATE TABLE

Mit dem SQL-Befehl CREATE TABLE erstellen Sie eine Tabelle. Für jedes Attribut der Relation definieren Sie eine Spalte und dieser Spalte muss ein passender Datentyp zugewiesen werden. (Tabellenspalten werden auch als Felder bezeichnet.) Sie erinnern sich, die Attributwerte stammten jeweils aus einem Wertebereich und dieser Wertebereich muss nun auf die Datentypen, die PostgreSQL unterstützt, abgebildet werden. Das hört sich zunächst schwieriger an, als es ist. PostgreSQL stellt Ihnen dazu eine Reihe vordefinierter Datentypen zur Verfügung, die recht intuitiv zu benutzen sind und im nächsten Kapitel ausführlich behandelt werden.

Erinnern wir uns an die erste Relation autoren, die mit ihrem Vornamen, Namen und einem eindeutigen Identifikator als Schlüssel beschrieben wurde. In einer realen Anwendung wird man noch einige andere Attribute definiert haben und, damit wir etwas zum Üben haben, werden wir noch ein Feld a_ort für den Wohnort und ein Feld a_gebj für das Geburtsjahr des Autors definieren. Die erste Spalte enthält die Autoren_ID, also einen ganzzahligen positiven numerischen Wert, ein Integer. In den drei nächsten Spalten der Tabelle sollen Name, Vorname und Wohnort des Autors gespeichert werden und im letzten Feld sein Geburtsjahr.

```
docgen=# CREATE TABLE autoren (
docgen(#  a_id integer,           --Nummer des Autors
docgen(#  a_vorname varchar(20), --Vorname des Autors
docgen(#  a_name varchar(20),    --Nachname des Autors
docgen(#  a_ort varchar(20),     --Wohnort des Autors
docgen(#  a_gebj integer         --Geburtsjahr
docgen(# );
CREATE
docgen=#
```

Der PostgreSQL-Server meldet CREATE – wir haben die Tabelle autoren erfolgreich erstellt. Schauen wir das genauer an:

- Mit den Schlüsselwörtern CREATE TABLE teilen wir dem Server mit, dass wir eine Tabelle erstellen wollen, der Name der Tabelle soll autoren sein. Die Tabelle gehört dem Benutzer, der sie erzeugt, anderen Benutzern ist der Zugriff auf die Tabelle verwehrt.
- Danach werden in runden Klammern die Spalten der Tabelle aufgelistet, indem man einfach einen Spaltennamen und einen Spaltentyp vergibt.
- Die SQL-Anweisung wird mit einem Semikolon abgeschlossen.
- Kommentare werden mit -- eingeleitet, danach wird alles bis zum Ende der Zeile als Kommentar interpretiert.

Zuordnung von Datentypen für Spalten

Durch die Zuordnung eines Datentyps für jede Spalte wird der Wertebereich definiert, aus dem die Inhalte der Spalte kommen dürfen. Dem Feld a_id wurde der Typ integer zugewiesen. Das sind ganzzahlige Werte mit einem Wertebereich von –2147483648 bis +2147483647. In diese Spalte können Sie keine anderen Werte eintragen, und wenn Sie es trotzdem versuchen, wird Ihre Eingabe vom Datenbankserver abgelehnt. Den Feldern für die Namen wurde der Datentyp varchar(n) zugewiesen. Alle so definierten Felder können maximal n Zeichen lang sein und erwarten Zeichenketten als Werte. Wenn der eingegebene String weniger als n Zeichen hat, wird nur die tatsächliche Anzahl Zeichen gespeichert, und wenn der String länger ist, wird er bis zur PostgreSQL-Version 7.1.3 auf n Zeichen abgeschnitten, ab Version 7.2 wird die Zeichenkette abgelehnt und eine Fehlermeldung generiert. Zeichenketten müssen in Hochkommas eingeschlossen sein.

Modifizierer kontrollieren die Eingabewerte

Bei der Definition der Spalten können Sie zusätzlich so genannte Modifizierer angeben, mit denen Sie den Datenbankserver anweisen, die Eingabewerte zu prüfen und gegebenenfalls abzulehnen. Damit haben Sie schon bei der Definition der Tabelle ein wichtiges Kontrollinstrument, das bereits auf der Systemebene die Eingabedaten sondiert. Diese Modifizierer nennt man Constraints. Sie werden im Kapitel 7 ausführlich beschrieben.

- UNIQUE lässt keine gleichen Werte aus dem entsprechenden Wertebereich in dieser Spalte zu. Jeder Wert darf nur einmal in dieser Spalte auftreten.
- DEFAULT setzt einen Vorgabewert, falls Sie für diese Spalte keinen Wert einfügen.
- NULL ist die Standardeinstellung und muss deshalb nicht explizit angegeben werden. In diese Spalte dürfen auch Nullwerte geschrie-

ben werden. Nullwerte sind nicht mit der Ziffer 0 oder einem leeren String gleichzusetzen. Es sind Platzhalter, falls für diese Spalte kein Wert vergeben wurde oder deren Wert unbekannt ist. (Mehr über Nullwerte lesen Sie im im nächsten Kapitel.)

- NOT NULL bestimmt, dass in dieser Spalte immer ein Wert aus dem zugrunde liegenden Wertebereich eingegeben werden muss.
- PRIMARY KEY zeichnet dieses Attribut als Primärschlüssel aus. In PostgreSQL sind Primärschlüssel »per definitionem« UNIQUE und NOT NULL. Eine Tabelle kann nur einen Primärschlüssel haben.

Duplikate verhindern mit UNIQUE

Um zu erzwingen, dass nur eindeutige Autoren_IDs in die Tabelle geschrieben werden dürfen – das ist der Sinn von Identifikatoren –, muss die Zeile in der Tabellendefinition so geschrieben werden:

```
docgen=# CREATE TABLE autoren (
docgen(#a_id integer UNIQUE ...
```

Wenn Sie jetzt versuchen, zweimal denselben Wert für a_id einzugeben, meldet der Server zurück, dass er kein Duplikat einfügen kann.

Eingabewerte erzwingen mit NOT NULL

Damit sind aber immer noch in mehreren Zeilen Nullwerte zugelassen, was Ihnen vielleicht nicht ganz logisch erscheint. Der Server interpretiert einen Nullwert als ***unbekannt*** und kann nicht entscheiden, ob zwei unbekannte Werte gleich sind und damit UNIQUE oder nicht. Um Nullwerte als Identifikatoren ebenfalls auszuschließen, definiert man:

```
docgen=# CREATE TABLE autoren (
docgen(#a_id integer UNIQUE NOT NULL ...
```

Spalten als Primärschlüssel auszeichnen

Wenn a_id, in Übereinstimmung zu dem Entwurf von Kapitel 1, als Primärschlüssel dieser Relation ausgezeichnet werden soll und damit automatisch NOT NULL und UNIQE ist, lautet die Spaltendefinition:

```
docgen=# CREATE TABLE autoren (
docgen(#a_id integer PRIMARY KEY ...
```

Die allgemeine Syntax zur Definition von Spalten innerhalb eines CREATE TABLE-Statements ist:

```
CREATE TABLE NameDerTabelle (
    feldname feldtyp [modifizierer]
    [, feldname feldtyp [modifizierer]]
    );
```

Wenn Sie diese Syntax nicht einhalten, kann PostgreSQL die Tabelle nicht erstellen und meldet einen parse error zurück, was bedeutet, dass die Anfrage nicht interpretiert werden kann.

Werfen wir noch einen zweiten Blick auf die Form der Query:

- Sie können so viele Leerzeichen eingeben, wie Sie wollen.
- Schlüsselwörter können in Groß- oder Kleinbuchstaben angegeben werden.
- Eine Abfrage kann auf mehrere Zeilen verteilt werden, sie wird erst mit dem abschließenden Semikolon (oder \g) gesendet. Nicht abgeschlossene Abfragen stellt psql durch (-# oder (# anstelle der Zeichen =# dar. Das ist bei langen Abfragen, die über mehrere Zeilen gehen, nützlich, weil es sehr viel übersichtlicher ist. Solange Sie kein Semikolon eingeben, wird alles, was Sie schreiben, in den Query-Puffer geschrieben, dessen Inhalt Sie mit \p anzeigen können, und \r leert den Puffer wieder. Wenn Sie den Query-Puffer mit einem externen Editor bearbeiten möchten, geben Sie das Kommando \e ein, dann wird standardmäßig der Editor vi aufgerufen, in dem Sie Ihr Kommando editieren können.

Mit dem psql-Kommando \dt (display) können Sie alle Tabellen anzeigen und mit \dt autoren wird Ihnen die Definition der Tabelle autoren angezeigt:

```
docgen=# \d autoren
           Table "autoren"
 Attribute |         Type          | Modifier
-----------+-----------------------+----------
 a_id      |        integer        |
 a_vorname | character varying(20) |
 a_name    | character varying(20) |
 a_ort     | character varying(20) |
 a_gebj    |        integer        |
```

Wenn Sie die Definition der Tabelle oben mit der Ausgabe von psql vergleichen, fällt auf, dass Felder, die mit dem Typ varchar(20) definiert wurden, hier als character varying(20) angezeigt werden. Der Grund dafür ist einfach: Sie bezeichnen denselben Typ und können alternativ verwendet werden.

3.5 Die Tabellen mit Daten füllen – INSERT

Der Zweck von Datenbanken ist die Speicherung gleich strukturierter Daten. Mit dem SQL-Kommando INSERT wird ein einzelner neuer Datensatz in eine Tabelle eingefügt.

```
INSERT INTO tabelle (spalte1, spalte2, ...)
VALUES (wert1, wert2, ...);
```

Damit weiß der Datenbankserver, dass Sie in der Tabelle tabelle in die Spalten spalte1, spalte2, ... die Werte (VALUES) wert1, wert2, ... eintragen wollen. Wichtig ist vor allem, dass die einzutragenden Werte in derselben Reihenfolge in der zweiten Klammer stehen, wie Sie die Spalten in der ersten Klammer angeordnet haben, weil sie in dieser Reihenfolge zugeordnet werden.

```
docgen=# INSERT INTO autoren (a_ID, a_vorname, a_name, a_ort,
a_gebj)
docgen-# VALUES (34, 'Hans', 'Dampf', 'Berlin', 1965);
INSERT 18740 1
```

OID, Object Identifier

Wenn PostgreSQL die Operation mit INSERT 18740 bestätigt, wurden die angegebenen Werte in die Tabelle geschrieben. Dem Feld a_ID wurde der Wert 34 zugewiesen, a_name hat nun den Wert Dampf, a_vorname erhielt den Wert Hans, a_ort den Wert Berlin und a_gebj den Wert 1965. Beachten Sie, dass Zeichenketten in Hochkommas eingeschlossen werden müssen, Zahlenwerte jedoch nicht. Die Nummer hinter dem INSERT bezeichnet den so genannten OID (Object Identifier). Das ist eine Datensatznummer, die für jeden eingefügten Datensatz von einem internen Zähler generiert wird. Bei objektrelationalen Datenbanksystemen wird gefordert, dass alle Datenbankobjekte eindeutig identifizierbar sind. Aus diesem Grund besitzen alle Tabellen zusätzlich ein Feld tableoid, das den Object Identifier der Tabelle enthält, zu der dieser Datensatz gehört. PostgreSQL vergibt bei jedem INSERT, egal in welche Tabelle und egal in welcher Datenbank, eine Datensatznummer. Bei sehr großen oder langlebigen Tabellen kann es vorkommen, dass der Datensatzzähler sich überrundet und zurücksetzt. Sie sollten deshalb nicht davon ausgehen, dass die Datensatznummern eindeutig sind. Zusammen mit dem tableoid können Datensätze eindeutig identifiziert werden. Ab der Version 7.2 von PostgreSQL ist es auch möglich, bei der Erzeugung einer Tabelle das Generieren dieser Datensatznummern zu unterdrücken. Dazu wird am Ende der Felddefinitionen WITHOUT OIDS angegeben.

```
CREATE TABLE tabellenname (
Felddefinitionen )
 [ WITH OIDS | WITHOUT OIDS ]
```

Kompletten Datensatz einfügen mit vereinfachter Syntax

Wenn Sie neue Datensätze einfügen und in alle Spalten einen Wert eintragen möchten, brauchen Sie die Spaltennnamen nicht anzugeben. Es gibt dafür eine vereinfachte Syntax. Sie müssen sich dann allerdings an die Reihenfolge halten, in der die Felder definiert wurden. Dazu können Sie sich zur Sicherheit die Tabellendefinition mit \dt autoren anzeigen lassen.

```
INSERT INTO autoren VALUES (51, 'Margret', 'Sommer', 'Köln', 1968);
INSERT INTO autoren VALUES (14, 'Rosa', 'Vetter', 'Dresden', 1972);
INSERT INTO autoren VALUES (22, 'Lisa', 'Dilger', 'Berlin', 1948);
```

Probieren Sie es aus und fügen Sie neue Autoren in die Tabelle ein. So wie die Tabelle definiert ist, sind Sie in der Wahl der Werte für die Identifizierer an keine Reihenfolge gebunden. Oft müssen Daten aus anderen Anwendungen oder alten Datenbeständen übernommen werden, die schon IDs haben.

Einzelne Felder einfügen

Wenn Sie nur einzelne Felder beschreiben wollen, geben Sie auch nur diese Felder an:

```
docgen=# INSERT INTO autoren (a_ID, a_gebj) values (44, 1970);
INSERT 18765 1
```

Es wird ein neuer Datensatz angelegt, der nur eine Autoren_ID und das Geburtsdatum enthält.

```
docgen=# INSERT INTO autoren (a_name, a_vorname, a_ort)
docgen=# values ('Pflug', 'Johann', 'Köln');
INSERT 18766 1
```

Hier wird ein neuer Datensatz angelegt, der nur Name und Vorname eines Autors enthält. Auch wenn die Reihenfolge der Attribute gegenüber dem obigen INSERT-Befehl vertauscht ist, ordnet PostgreSQL die Werte korrekt zu. Diese beiden Beispiele sind im richtigen Leben wertlos, aber syntaktisch korrekt. Lassen wir uns den derzeitigen Inhalt der Tabelle anzeigen:

```
docgen=# SELECT * FROM autoren;
 a_id |  a_vorname  |    a_name   |    a_ort    | a_gebj
------+-----------..+-----------..+----------..+-------
 34   | Hans        | Dampf       | Berlin      | 1965
 51   | Margret     | Sommer      | Köln        | 1968
 22   | Lisa        | Dilger      | Berlin      | 1948
 14   | Rosa        | Vetter      | Dresden     | 1972
 44   |             |             |             | 1970
      | Johann      | Pflug       | Köln        |
(6 rows)
```

Nullwerte

Die nicht beschriebenen Felder werden einfach ausgelassen. In Kapitel 1 hatten wir aber behauptet, dass es in einer relationalen Datenbank keine leeren Felder geben kann. Es sieht aus wie ein Widerspruch, ist aber keiner, denn: PostgreSQL schreibt in alle undefinierten Tabellenzellen einen speziellen NULL-Wert. Dieses NULL kann man so interpretieren, dass kein Wert für diese Spalte existiert, oder dass der Wert für dieses Feld unbekannt ist, auf *keinen* Fall kann man NULL mit der Zahl

0 gleichsetzen. NULL kann in jeder Spalte stehen, egal welcher Spaltentyp zugeordnet wurde. Mehr zu NULL finden Sie im Kapitel 4.

Daten aus Dateien einfügen

Sie können auch eine Datei im ASCII-Format mit folgendem Inhalt erstellen und im Verzeichnis /home/IhrBenutzername/ unter dem Dateinamen ins_autoren.sql speichern:

```
INSERT INTO autoren VALUES (11, 'Bruno', 'Maier', 'Hamburg', 1959);
INSERT INTO autoren VALUES (7, 'Josef', 'Amann', 'Dresden', 1978);
INSERT INTO autoren VALUES (27, 'Lisa', 'Amann', 'Stuttgart', 1975);
INSERT INTO autoren VALUES (16, 'Hugo', 'Müller', 'München', 1962);
INSERT INTO autoren VALUES (31, 'Anne', 'Frei', 'Aachen', 1974);
```

Danach starten Sie den Datenbankmonitor erneut mit psql docgen. Nach der Begrüßungsmeldung geben Sie

```
docgen=# \i ins_autoren.sql
```

ein. PostgreSQL liest die Anweisungen aus der Datei und führt sie aus. Das können Sie am Bildschirm mitverfolgen. Dies ist aber nur bei wenigen einzufügenden Datensätzen sinnvoll, da diese Methode nicht sehr schnell ist. Wenn Sie viele Datensätze eingeben müssen, sollten Sie den COPY-Befehl benutzen, der im Kapitel 8 beschrieben wird. Das Kommando \i kann man nicht nur zum Einfügen verwenden, Sie können jedes SQL-Kommando oder Abfolgen von SQL-Kommandos in eine ASCII-Datei schreiben und mit \i in psql einlesen.

Mit dem INSERT-Kommando können auch mehrere Datensätze in die aktuelle Tabelle eingefügt werden, die von einem SELECT generiert wurden. Hierfür benutzt man die folgende Syntax:

```
INSERT INTO tabelle [(spalte1, spalte2, ...)]
SELECT abfrage;
```

Die Ergebnisse des SELECTs werden in die Tabelle eingefügt. Ein Beispiel dazu sehen Sie im Abschnitt 3.10.

3.6 Auf die Daten zugreifen – SELECT

Wie man eine Tabelle mit Daten füllt, wissen Sie jetzt. Wie man sie ausliest, ist unser nächstes Thema. Dazu gibt es in SQL nur ein Kommando: SELECT. Wahrscheinlich ist das der komplexeste Befehl in SQL und der, den Sie am häufigsten benutzen werden. Die komplette Syntax dieses Statements finden Sie im PostgreSQL Reference Manual. Wir wollen uns den Inhalt der Tabelle autoren ansehen:

```
docgen=# SELECT * FROM autoren;
a_id |  a_vorname | a_name    |  a_ort     | a_gebj
-----+---------...+---------...+---------...+-------
  34 | Hans      | Dampf     | Berlin     | 1965
  51 | Margret   | Sommer    | Köln       | 1968
  22 | Lisa      | Dilger    | Berlin     | 1948
  14 | Rosa      | Vetter    | Dresden    | 1972
  44 |           |           |            | 1970
     | Johann    | Pflug     | Köln       |
  11 | Bruno     | Maier     | Hamburg    | 1959
   7 | Josef     | Amann     | Dresden    | 1978
  27 | Lisa      | Amann     | Stuttgart  | 1975
  16 | Hugo      | Müller    | München    | 1962
  31 | Anne      | Frei      | Aachen     | 1974
(11 rows)
```

Alle Spalten ausgeben

Dies ist die einfachste Form des SELECT. Mit dem Stern * wird dem Server mitgeteilt, dass er alle Spalten der Tabelle, die nach FROMangegeben wird, ausgeben soll. Sie sehen, dass die Spaltennamen, die Sie bei der Tabellendefinition angegeben haben, auch in der Ausgabe als Spaltennamen angeschrieben sind. Dieses Statement kann auch so geschrieben werden:

```
SELECT a_id, a_name, a_vorname, a_ort, a_gebj FROM autoren;
```

Die allgemeine (stark vereinfachte) Syntax für SELECT ist:

```
SELECT feld1, feld2, ... FROM tabelle;
```

Ausgewählte Spalten ausgeben

Will man nur bestimmte Felder anzeigen, gibt man die Spalten in einer kommaseparierten Liste (Feldliste) an, wobei die Reihenfolge der zurückgegebenen Felder der Reihenfolge in der Feldliste entspricht:

```
SELECT a_id, a_name, a_gebj FROM autoren;
```

Wenn Sie ein Feld angeben, das nicht in dieser Tabelle definiert ist, erhalten Sie eine Fehlermeldung, ebenso, wenn Sie aus einer Tabelle lesen wollen, die es nicht gibt. Vielleicht haben Sie dann einfach nur einen Tippfehler gemacht.

```
docgen=# SELECT name FROM autoren;
ERROR: attribute 'name' not found
docgen=# Select * from autor;
ERROR: Relation 'autor' does not exist
docgen=#
```

3.6.1 Ergebnisse sortieren – ORDER BY

Ihnen fällt bei den bisherigen Ergebnissen bestimmt auf, dass die Datensätze vollkommen unsortiert ausgegeben werden. Das ist bei relationalen Datenbanken üblich, denn es gibt keine Vorschriften, in welcher Reihenfolge Daten zurückgegeben werden. Normalerweise werden die Datensätze in derselben Reihenfolge zurückgegeben, wie sie gespeichert wurden, was in den meisten Fällen unpraktisch und schlecht lesbar ist.

Hier hilft uns, dass der SELECT-Befehl um so genannte »Klauseln« erweitert werden kann. Eine solche Klausel ist ORDER BY, mit der man eine Tabelle nach einem oder mehreren Feldern sortiert ausgeben kann:

```
SELECT feldliste
FROM tabelle
ORDER BY feldname [, feldname] [ASC|DESC];
```

[ASC|DESC] am Ende des Befehls gibt an, ob die Tabelle aufsteigend oder absteigend sortiert werden soll. Fehlt diese Angabe, wird aufsteigend (= ASC) sortiert.

```
docgen=# SELECT a_name, a_vorname, a_gebj
docgen-# FROM autoren
docgen-# ORDER BY a_name DESC;
      a_name       |     a_vorname      | a_gebj
-------------------+-------------------+--------
 Vetter            | Rosa              | 1972
 Sommer            | Margret           | 1968
 Pflug             | Johann            |
 Miller            | Hugo              | 1962
 Maier             | Bruno             | 1959
 Frei              | Anne              | 1974
 Dilger            | Lisa              | 1948
 Dampf             | Hans              | 1978
 Amann             | Josef             | 1975
 Amann             | Lisa              | 1975
                   |                   | 1970
(11 rows)
```

Erweiterte Sortiermöglichkeiten in PostgreSQL

Abweichend vom SQL-Standard kann man in PostgreSQL Tabellen nach frei wählbaren Ausdrücken sortieren. Wenn feld1 und feld2 zum Beispiel numerische Werte repräsentieren, kann man eine Abfrage

```
SELECT feld1, feld2 FROM tabelle ORDER BY feld1 + feld2;
```

formulieren, die nach der Summe der beiden Felder sortiert ist. In unserer Tabelle könnte man, auch wenn es keinen Sinn macht, die ID und das Geburtsjahr addieren und diese Summe als Sortierungskriterium benutzen. Ebenfalls nicht standardkonform akzeptiert PostgreSQL, dass eine Ergebnismenge nach einem Feld, das nicht in der Feldliste enthalten ist, sortiert werden kann. Abfragen wie

```
SELECT a_name, a_vorname FROM autoren ORDER BY a_id;
```

sind möglich. Von ihrer Verwendung wird aber abgeraten, weil sie nicht dem SQL-Standard entsprechen und eventuelle Portierungen erschweren.

3.6.2 Feldnamen mit Aliasnamen überschreiben – AS

Angenommen, Sie wollen ein Abfrageergebnis direkt auf einer Webseite ausgeben, dann sollten die Spaltenüberschriften in einer verständlichen Form dargestellt werden können. Die bei der Tabellendefinition vergebenen Spaltennamen sind oft Abkürzungen oder für den Besucher unverständliche Akronyme. Es gibt eine einfache Möglichkeit, um Spaltennamen bei der Ausgabe umzubenennen. Sie brauchen nur das Schlüsselwort AS und den neuen Namen in den Befehl einfügen. Diese Spaltennamen werden als »Aliase« bezeichnet. Spalten mit Aliasnamen sind temporäre Spalten nur für dieses SELECT-Statement, die in folgenden Abfragen nicht weiterverwendet werden:

```
docgen=# SELECT a_vorname AS "Vorname",
docgen-#     a_name AS "Name",
docgen-#     a_ort AS "Wohnort"
docgen-# FROM autoren ORDER BY "Name";
```

Probieren Sie es aus. Das Abfrageergebnis zeigt alle Autoren nach den Familiennamen sortiert und mit allgemein verständlichen Spaltenüberschriften an. Beachten Sie die doppelten Anführungszeichen, dadurch werden die Spaltenüberschriften mit Großbuchstaben ausgegeben.

3.6.3 Duplikate unterdrücken – DISTINCT

Wenn Sie herausfinden wollen, aus welchen Orten die Autoren kommen, senden Sie folgende Anfrage:

```
docgen=# SELECT a_ort FROM autoren ORDER BY a_ort;
       a_ort
--------------------
 Aachen
 Berlin
 Berlin
 Dresden
 Dresden
 Hamburg
 Köln
 Köln
 München
 Stuttgart
(11 rows)
```

Das ist zwar korrekt, aber wahrscheinlich mehr, als Sie wissen wollten. Normalerweise zeigt PostgreSQL alle Zeilen an, die den Angaben in dem SELECT-Befehl entsprechen. Das kann bei großen Datenbeständen mit vielen gleichen Einträgen in den ausgewählten Spalten schnell lästig werden. In SQL kann man die Ausgabe von Duplikaten unterdrücken, indem man in der SELECT-Abfrage das Schlüsselwort DISTINCT direkt hinter SELECT einfügt. Das weist den Datenbankserver an, Duplikate zu unterdrücken.

```
docgen=# SELECT DISTINCT a_ort FROM autoren;
       a_ort
--------------------
 Aachen
 Berlin
 Dresden
 Hamburg
 Köln
 München
 Stuttgart

(8 rows)
```

Wie Sie sehen, sind die Orte zusätzlich sortiert. Das liegt daran, dass der Suchalgorithmus in PostgreSQL so implementiert wurde. Sie sollten sich nicht darauf verlassen, denn wenn die zugrunde liegende Implementierung sich ändert, ändert sich möglicherweise auch dieses Verhalten. Der richtige Weg zum Sortieren der Ergebnisse ist immer die Verwendung von ORDER BY. Außerdem fällt auf, dass psql zwar acht Ergebniszeilen angibt, wir aber nur sieben Ergebnisse erhalten. Wir haben einen Datensatz, der keine Angabe zu dem Ort enthält. Dieser wird als leerer Eintrag am Ende der Liste angezeigt.

DISTINCT ist nicht an ein bestimmtes Feld gebunden, sondern bezieht sich auf alle Felder, die Sie im SELECT angeben. Hier werden alle Orte angezeigt, da es keine doppelten Ort/Name-Paare gibt. Am Ende der Tabelle erscheint der Datensatz, der keinen Eintrag für den Namen und den Ort hat, mit zwei leeren Feldern.

```
docgen=# SELECT DISTINCT a_name, a_ort
docgen-# FROM autoren ORDER BY a_ort, a_name;
      a_name         |       a_ort
---------------------+--------------------
 Frei                | Aachen
 Dampf               | Berlin
 Dilger              | Berlin
 Amann               | Dresden
 Vetter              | Dresden
 Maier               | Hamburg
 Pflug               | Köln
 Sommer              | Köln
 Müller              | München
 Amann               | Stuttgart
                     |
 (11 rows)
```

DISTINCT ist noch in einer anderen Syntax verfügbar. Damit kann man die mehrfache Ausgabe von Datensätzen, die in einem oder mehreren Ausdrücken gleiche Werte haben, unterdrücken.

```
SELECT DISTINCT ON (ausdruck [, ausdruck])
```

verhindert die Ausgabe von Duplikaten, die im ausdruck denselben Wert haben. Der ausdruck kann beispielsweise eine Spalte oder eine Verknüpfung von Spalten sein.

```
SELECT DISTINCT ON (a_ort) a_ort FROM autoren;
```

gibt dasselbe Ergebnis wie

```
SELECT DISTINCT a_ort FROM autoren;
```

zurück.

3.6.4 Zeilen auswählen – WHERE

Ausgaben an Bedingungen knüpfen

In allen bisher gezeigten Beispielen haben sich die SELECTs immer auf die ganze Tabelle bezogen. Bei jeder Abfrage umfasste die Ergebnismenge alle Zeilen der Tabelle. Bei großen Tabellen ist diese Vorgehensweise uneffizient und oft unerwünscht, weil man sich meistens nur für

einen Teil der Datensätze mit bestimmten Attributwerten interessiert, die in der Tabelle gesucht und ausgegeben werden sollen. Die Formulierung einer oder mehrerer verknüpfter Bedingungen, denen die Datensätze genügen müssen, geschieht in der WHERE-Klausel, die in den SELECT-Befehl eingefügt wird. Die Bedingung muss entweder zu TRUE oder FALSE ausgewertet werden können.

```
SELECT feldliste FROM tabelle WHERE bedingung;
```

Boolesche Ausdrücke

Dieser Befehl bewirkt, dass nur noch Datensätze, auf die die Bedingungen zutreffen, in der Ergebnismenge erscheinen. Sie können mehrere Bedingungen angeben, um ganz bestimmte Datensätze auszusortieren. Die Bedingungen folgen den Regeln der booleschen Algebra und können mit den Operatoren AND, OR und NOT verknüpft werden. Demzufolge werden Sie auch als boolesche Ausdrücke (boolean expressions) bezeichnet.

In den Bedingungen dürfen die folgenden Vergleichsoperatoren verwendet werden:

< kleiner als

<= kleiner oder gleich als

= gleich

>= größer oder gleich als

> größer als

<> oder != not

Beginnen wir mit einem einfachen Beispiel, das uns die Namen aller Berliner Autoren auflistet:

```
docgen=# SELECT * FROM autoren WHERE a_ort = 'Berlin';
a_id |  a_vorname   |   a_name   |    a_ort   | a_gebj
-----+----------...+---------...+--------...+-------
 34  | Hans         | Dampf      | Berlin     | 1965
 22  | Lisa         | Dilger     | Berlin     | 1948
(2 rows)
```

Nun wollen wir alle Autoren, nach Namen sortiert, die jünger als 30 Jahre sind und in Dresden wohnen:

```
docgen=# SELECT a_vorname, a_name FROM autoren
docgen-# WHERE a_gebj > 1972 AND a_ort = 'Dresden'
docgen-# ORDER BY a_name;
```

Ad-hoc-Abfragen

Die nächste Abfrage soll uns alle Autoren nach Namen sortiert zurückgeben, die jünger als 30 Jahre sind und in Dresden oder Stuttgart wohnen. (Solche Abfragen mit Berechnungen des Alters eignen sich nur für

Ad-hoc-Abfragen, aber nicht für Anwendungen. Jedesmal, wenn sich das aktuelle Jahr ändert, wird die Altersangabe falsch.)

```
docgen=# SELECT a_vorname, a_name FROM autoren
docgen-# WHERE a_gebj > 1972
docgen-# AND (a_ort = 'Dresden' OR a_ort = 'Stuttgart')
docgen-# ORDER BY a_name;
     a_vorname     |      a_name
-------------------+--------------------
 Josef             | Amann
 Lisa              | Amann
 Rosa              | Vetter
(3 rows)
```

Prüfen auf Enthaltensein in einer Menge

Im letzten Beispiel ist eine Klammer gesetzt, um zusammengehörige Bedingungen zu gruppieren. Die Klammern sind hier nicht unbedingt erforderlich, erhöhen aber die Lesbarkeit. Diese Abfrage kann mit dem Schlüsselwort `IN` einfacher formuliert werden. Mit `IN` wird auf Enthaltensein in einer Menge geprüft. Hier geht es darum, ob a_ort des jeweils aktuellen Datensatzes in der Menge `('Dresden''Stuttgart')` enthalten ist.

```
docgen=# SELECT a_vorname, a_name FROM autoren
docgen-# WHERE a_gebj > 1972
docgen-# AND a_ort IN ('Dresden', 'Stuttgart')
docgen-# ORDER BY a_name;
```

Bereichsabfragen

Wenn Sie in Ihrer Bedingung einen Bereich angeben wollen, (beispielsweise wollen Sie nur die Datensätze mit IDs von 14 bis 27 anzeigen), dann können Sie die Bedingung auf unterschiedliche Weise formulieren:

- Sie können eine Reihe von `OR`-Bedingungen definieren.
- Sie können den fraglichen Bereich in einer Menge angeben und das Enthaltensein mit `IN` prüfen.
- Sie können die Vergleichsoperatoren in Verbindung mit `AND` benutzen:

  ```
  WHERE a_id >= 14 AND a_id < 28
  ```

- Oder Sie benutzen das SQL-Schlüsselwort `BETWEEN` mit `AND`:

  ```
  WHERE a_id BETWEEN 14 AND 28
  ```

Mit `BETWEEN` gibt man ein Intervall an, das die beiden Grenzen enthält. `BETWEEN` kann man auch auf Strings anwenden, die lexikalisch sortiert werden können.

3.6.5 Die Ausgabemenge beschränken – LIMIT und OFFSET

Es gibt Situationen, in denen man nur die nächsten n Datensätze anzeigen möchte. Denken Sie an Suchergebnisse auf einer Internetseite. Mit dem Schlüsselwort LIMIT können Sie bestimmen, wie viele Zeilen die Ergebnismenge enthalten soll, und mit OFFSET geben Sie an, von welcher Zeile an die Ausgabe erfolgen soll. LIMIT/OFFSET sollten Sie in Verbindung mit ORDER BY benutzen, da sonst die Reihenfolge der Datensätze in der Ausgabe nicht vorhersehbar ist (s. o.). Die Angabe von LIMIT bzw. LIMIT/OFFSET erhöht die Ausführungsgeschwindigkeit, da der Server nur einen Teil der Datensätze zurückgeben muss.

```
docgen=# SELECT a_id FROM autoren ORDER BY a_id LIMIT 3;
 a_id
-------
     7
    11
    14
(3 rows)

docgen=# SELECT a_id FROM autoren ORDER BY a_name LIMIT 3 OFFSET 4;
 a_id
-------
    22
    27
    31
(3 rows)
```

Die erste Abfrage sortiert die Tabelle nach a_id und liefert mit LIMIT die ersten drei Datensätze zurück. Im zweiten Beispiel wird die Tabelle nach dem Namen des Autors sortiert, dann werden vier Datensätze übersprungen und dann die nächsten drei Sätze ausgegeben.

3.6.6 Die Ausgabe an eine Bedingung knüpfen – CASE

Auch wenn SQL keine prozedurale Sprache ist, erlaubt Sie Ihnen doch, Ausgaben aus Tabellen an Bedingungen zu knüpfen und so die Werte in den Spalten innerhalb einer Abfrage miteinander zu vergleichen. Die neu generierte Spalte erscheint nur in der Ausgabe, nicht in der Tabelle. Angenommen, Sie wollen die Autoren nach dem Jahrzehnt, in dem sie geboren sind, zusammenfassen und jedem Jahrzehnt eine Kennung zuordnen, beispielsweise eine Farbkodierung:

```
docgen=# SELECT a_vorname, a_name, a_gebj,
docgen-#   CASE
docgen-#     WHEN a_gebj BETWEEN 1940 AND 1949 THEN 'rot'
docgen-#     WHEN a_gebj BETWEEN 1950 AND 1959 THEN 'gelb'
docgen-#     WHEN a_gebj BETWEEN 1960 AND 1969 THEN 'blau'
docgen-#     WHEN a_gebj BETWEEN 1970 AND 1979 THEN 'grün'
docgen-#     ELSE 'lila'
docgen-#   END AS farbe
docgen-# FROM autoren
docgen-# ORDER BY a_gebj;
  a_vorname  |   a_name    | a_gebj | farbe
-------------+-------------+--------+---------
 Lisa        | Dilger      | 1948   | rot
 Bruno       | Maier       | 1959   | gelb
 Hugo        | Sommer      | 1962   | blau
 ...
 Rosa        | Vetter      | 1972   | grün
 Anne        | Frei        | 1974   | grün
 ...           ...           ...      ...
(10 rows)
```

CASE kann überall dort eingesetzt werden, wo ein Ausdruck erlaubt ist. Die Bedingung gibt ein boolesches Ergebnis zurück. Wenn dieses Ergebnis TRUE ist, wird der Wert wert zugeordnet, ansonsten wird die nächste WHEN-Bedingung ausgewertet, usw. Wenn keine Bedingung zu TRUE ausgewertet werden kann, wird der wert im ELSE-Zweig zugeordnet. Die Angabe eines ELSE-Zweigs ist optional. Wenn auf diese Weise kein wert ermittelt werden kann, ist das Ergebnis von CASE NULL. Die Definition einer CASE-Bedingung hat folgende Syntax:

```
CASE WHEN bedingung THEN wert
     [WHEN ...]
     [ELSE wert]
END
```

3.6.7 Erweiterte Syntax des SELECT-Kommandos

Im Laufe dieses Abschnitts wurde das SELECT-Kommando mehrfach erweitert. Zum Schluss deshalb eine Zusammenfassung der Syntax mit allen Erweiterungen dieses Abschnitts. Die vollständige Syntax aller SQL-Befehle in PostgreSQL können Sie im Reference Manual nachlesen.

```
SELECT [ ALL | DISTINCT ]
    * | ausdruck [ AS aliasname ] [, ...]
    [ FROM tabelle ]
    [ WHERE bedingung ]
    [ ORDER BY ausdruck [ ASC | DESC ] ]
    [ LIMIT { anzahl | ALL } [ OFFSET start ]]
```

3.7 Datensätze löschen – DELETE

Datensätze, die in eine Tabelle eingefügt wurden, müssen auch wieder gelöscht werden können. Mit dem SQL-Befehl DELETE können Sie einzelne oder alle Zeilen einer Tabelle löschen.

```
DELETE FROM autoren;
```

löscht die ganze Tabelle. Dasselbe erledigt in PostgreSQL auch die Anweisung `TRUNCATE tabelle;` oder `TRUNCATE TABLE tabelle;`, mit der Sie ebenfalls den Inhalt einer Tabelle löschen können. TRUNCATE ist allerdings kein SQL-Standard, sondern eine Erweiterung von PostgreSQL. Da TRUNCATE im Gegensatz zu DELETE die Tabelle nicht liest, wird dieses Kommando schneller ausgeführt.

Einzelne Zeilen aus einer Tabelle löschen Sie, indem Sie in einer WHERE-Klausel eine Bedingung angeben. Trifft die Bedingung auf eine Zeile zu, wird sie gelöscht. Wir haben in unserer Tabelle zwei unvollständige Datensätze, die jetzt gelöscht werden sollen:

```
docgen=# DELETE FROM autoren WHERE a_id = 44 OR a_name = 'Pflug';
DELETE 2
```

PostrgeSQL löscht ohne Nachfrage

PostgreSQL bestätigt die Löschoperation und gibt die Anzahl der gelöschten Datensätze aus. Wenn Sie danach die Tabelle mit `SELECT * FROMautoren;` anzeigen lassen, sehen Sie, dass diese beiden Sätze nicht mehr vorhanden sind. Seien Sie aber vorsichtig. Wenn mehrere Autoren in der Tabelle den Namen Pflug haben, werden alle ohne Nachfrage gelöscht. Bevor Sie das Löschkommando ausführen, können Sie mit derselben WHERE-Klausel in einem SELECT die Datensätze, die Sie löschen wollen, anzeigen lassen, um sicher zu sein, dass Sie nicht unbeabsichtigt Datensätze löschen.

3.8 Datensätze aktualisieren – UPDATE

Eine Datenbank ist nicht für die Ewigkeit. Die Daten in den Tabellen verändern sich möglicherweise laufend. Denken Sie an ein Skript, das Zugriffe auf Webseiten protokolliert, oder an Umfragen im Internet.

Jedesmal, wenn ein Benutzer eine Seite aufruft oder an einer Abstimmung teilnimmt, müssen die Daten in den entsprechenden Tabellen aktualisiert werden. Bei unseren Autoren kann es vorkommen, dass sich der Familienname oder der Wohnort ändert. Mit unseren bisher bekannten Mitteln könnte man den Datensatz, der aktualisiert werden soll, zuerst mit DELETE löschen und dann mit INSERT neu einfügen. Das geht aber wesentlich einfacher und eleganter mit dem SQL-Befehl UPDATE:

```
UPDATE tabelle SET feld = neuerWert [, ...]
[FROM fromliste]
[WHERE bedingung]
```

Updates aus mehreren Tabellen

Die optionale FROM-Klausel ist eine PostgreSQL-Erweiterung und erlaubt, dass Spalten aus anderen Tabellen in der WHERE-Klausel auftreten dürfen. Dadurch können Sie die Aktualisierung von Datensätzen in der aktuellen Tabelle von Spaltenwerten in anderen Tabellen abhängig machen, wodurch dieses Statement eine große Flexibilität bekommt. Die WHERE-Klausel ist ebenfalls optional und hat dieselbe Semantik wie bei SELECT beschrieben. Geben Sie UPDATE ohne WHERE-Klausel ein, so wird die angegebene Spalte der ganzen Tabelle auf den neuen Wert gesetzt. In den wenigsten Fällen wird das Ihre Absicht sein. Da SQL-Befehle nicht rückgängig gemacht werden können, ist hier Vorsicht geboten. Ein erfolgreiches Update wird vom Datenbankserver mit der Meldung UPDATE # quittiert, wobei # die Anzahl der aktualisierten Datensätze angibt.

In unserem Autorenteam hat sich einiges verändert: Herr Hugo Müller aus München und Frau Margret Sommer aus Köln haben geheiratet. Herr Müller wird zukünftig Sommer heißen und das Paar wohnt jetzt in Frankfurt.

```
docgen=# UPDATE autoren SET a_name = 'Sommer', a_ort = 'Frankfurt'
docgen-# WHERE a_name = 'Müller'
docgen-# AND a_ort = 'München'
docgen-# AND a_vorname = 'Hugo';
UPDATE 1
docgen=# UPDATE autoren SET a_ort = 'Frankfurt'
docgen-# WHERE a_name = 'Sommer'
docgen-# AND a_vorname = 'Margret'
docgen-# AND a_ort = 'Köln';
UPDATE 1
docgen=# SELECT * FROM autoren;
UPDATE 1
```

Im Unterschied dazu würden die beiden folgenden Befehle in der ganzen Tabelle alle Werte der Spalte a_name auf Maier setzen und alle Werte der Spalte a_ort auf Berlin. Dann hießen alle Autoren Maier und wohnten in Berlin. Beachten Sie, dass PostgreSQL eine UPDATE-Aktion bestätigt und die Zahl der aktualisierten Datensätze anzeigt.

```
UPDATE autoren SET a_name = 'Maier';
UPDATE autoren SET a_ort = 'Berlin';
```

3.9 Tabellen löschen – DROP TABLE

Das Gegenteil von CREATE ist DROP. So wie Sie mit CREATE DATABASE meinedb eine Datenbank erzeugen können, wird sie mit DROP DATABASE meinedb wieder gelöscht. Genauso wird eine Tabelle, die mit CREATE TABLE meinetabelle erstellt wurde, mit DROP TABLE meinetabelle gelöscht. Seien Sie vorsichtig mit diesen Kommandos, PostgreSQL macht keine Sicherheitsabfrage, bevor es die Daten aus dem System entfernt. Wie bei diesen Kommandos sehen Sie oft schon am ersten Schlüsselwort, wofür der SQL-Befehl verwendet wird.

```
DROP TABLE autoren;
```

Dieser Befehl löscht unsere Tabelle aus der Datenbank und alle darin enthaltenen Datensätze. Nur der Besitzer einer Tabelle kann diese auch löschen. Mit DROP TABLE können mehrere Tabellen gleichzeitig gelöscht werden, indem man die Namen der Tabellen, durch Kommas getrennt, angibt. Sie sollten diesen Befehl noch nicht ausführen, da die Tabelle im folgenden Kapitel noch als Beispiel dient. Später werden wir sie löschen und neu erzeugen und unsere bis dahin erworbenen Kenntnisse anwenden.

3.10 Die Tabellenstruktur ändern – ALTER TABLE

Auch wenn Sie Ihre Datenbankstruktur sorgfältig geplant haben, es wird immer einmal vorkommen, dass Sie eine vorhandene Struktur verändern müssen, beispielsweise, wenn Sie eine Applikation erweitern. Dazu stellt SQL das Kommando ALTER TABLE zur Verfügung, allerdings sind in PostgreSQL nicht alle Features des SQL-Standards implementiert. Mit ALTER TABLE können Sie (unter anderem):

- Spalten hinzufügen,
- einen Standardwert für eine Spalte vergeben oder löschen,
- eine Spalte umbenennen und
- eine Tabelle umbenennen.

In PostgreSQL noch nicht implementiert:

- eine Spalte löschen,
- eine Spalte von NULL auf NOT NULL setzen und umgekehrt,
- den Datentyp einer Spalte ändern.

Diese Operationen lassen sich auf Umwegen nachbilden, indem man temporäre Tabellen erstellt und die Daten hin- und herkopiert. Ab der PostgreSQL Version 7.2 ist es möglich, eine Spalte im Nachhinein über einen Tabellen-Constraint (siehe Kapitel 7) als Primärschlüssel auszuzeichnen.

Der Tabelle autoren soll eine neue Spalte für das exakte Geburtsdatum zugefügt werden. Neue Spalten werden immer am Ende der Tabelle eingefügt.

```
docgen=# ALTER TABLE autoren ADD COLUMN geburtsdatum date;
ALTER
```

Danach fällt auf, dass das neue Feld nicht nach der bisherigen Schreibweise mit dem Präfix a_ definiert wurde, und außerdem ist der Feldname zu lang. Das wird nun geändert.

```
docgen=# ALTER TABLE autoren
docgen-# RENAME COLUMN geburtsdatum TO a_gebdat;
ALTER
docgen=# \d autoren
       Table autoren
 Attribute |         Type          | Modifier
-----------+-----------------------+----------
 a_id      | integer               |
 a_vorname | character varying(20) |
 a_name    | character varying(20) |
 a_ort     | character varying(20) |
 a_gebj    | integer               |
 a_gebdat  | date                  |
```

Nachdem nun ein Feld für das genaue Geburtsdatum in die Tabelle eingefügt wurde, ist das Feld a_gebj überflüssig geworden. Zum Löschen einer Spalte muss man einen Umweg gehen, da PostgreSQL diese Operation erst ab Version 7.2 unterstützt. Zunächst wird eine neue Tabelle autoren2 definiert, die alle Felder von autoren außer dem Feld a_gebj enthält, denn dieses soll ja in der neuen Tabelle nicht mehr enthalten sein. Mit INSERT INTO ... SELECT werden nun die Daten aus autoren in die neu erzeugte Tabelle autoren2 übernommen. Danach wird mit DROP TABLE die ursprüngliche Tabelle autoren gelöscht. Zum Schluss muss die Tabelle autoren2 noch in autoren umbenannt werden. Das Resultat ist eine Tabelle autoren ohne das Feld a_gebj.

```
docgen=# CREATE TABLE autoren2 (
docgen(# a_id integer,
docgen(# a_vorname varchar(20),
docgen(# a_name varchar(20),
docgen(# a_ort varchar(20),
docgen(# a_gebdat date);
CREATE
docgen=# \d
   List of relations
    Name    | Type  | Owner
------------+-------+-------
 autoren    | table | conni
 autoren2   | table | conni
(2 rows)

docgen=# INSERT INTO autoren2 (a_id, a_vorname, a_name, docgen-#
a_ort, a_gebdat) SELECT
docgen-# a_id, a_vorname, a_name, a_ort, a_gebdat
docgen-# FROM autoren;
INSERT 0 10
docgen=# DROP TABLE autoren;
docgen=# ALTER TABLE autoren2 RENAME TO autoren;
```

Wenn mehrere Datensätze in eine Tabelle eingefügt wurden, kann PostgreSQL keinen Object Identifier zurückgeben und gibt stattdessen eine 0 zurück. Danach folgt die Anzahl der eingefügten Datensätze.

Wenn Sie die Strukturen Ihrer Tabellen häufig ändern und anpassen müssen, scheint Ihr Entwurf nicht optimal gelungen zu sein und Sie sollten ihn insgesamt überdenken. Wenn Sie erstmal Daten in Ihren Tabellen haben, sind alle nachträglichen Änderungen aufwändig und fehleranfällig.

3.11 Tabellen anderen Benutzern verfügbar machen

Immer, wenn ein Datenbankobjekt (Tabelle, Sequenz oder View) erstellt wird, ist der Erzeuger auch gleichzeitig der Besitzer des Objekts mit allen Zugriffsrechten. Dieser Besitzer kann mit dem Kommando

```
ALTER TABLE tabelle OWNER TO neuerBesitzer
```

geändert werden. Um anderen Benutzern den Zugriff auf die Objekte zu erlauben, können Privilegien definiert werden. Dies sind Lese- oder Schreibrechte, die einzelnen Benutzern oder Gruppen übertragen werden können oder allen Benutzern des Systems. Bis zur Version 7.1.3 von PostgreSQL gab es die Privilegien SELECT, INSERT, UPDATE und

DELETE sowie RULE, die Erlaubnis, eine Regel für eine Tabelle zu erstellen. (Regeln sind PostgreSQL-spezifische Möglichkeiten, auf die im Kapitel 14 eingegangen wird.) In der Version 7.2 kommen Rechte zum Erzeugen von Tabellen mit Fremdschlüsseln und zum Erzeugen von Triggern für die aktuelle Tabelle hinzu (siehe Kapitel 7 bzw. 16). Um eine Tabelle mit einem Fremdschlüssel zu erzeugen, muss der Benutzer ein REFERENCES-Privileg auf der Tabelle haben, das sich auf deren Primärschlüssel bezieht. Erteilte Privilegien können von dem Besitzer des Datenbankobjekts auch wieder zurückgezogen werden.

```
GRANT UPDATE, DELETE ON tabelle TO benutzer;
```

gibt dem angegebenen Benutzer das Recht, die Tabelle zu aktualisieren und Datensätze zu löschen. Wenn Sie einem Benutzer alle Privilegien geben möchten, können Sie eine Kurzform mit dem Schlüsselwort ALL benutzen. Hier bekommt der Webserver apache alle Privilegien:

```
docgen=# GRANT ALL ON autoren TO apache;
CHANGE
```

Wenn Sie die Tabelle allen Nutzern auf dem System zugänglich machen möchten:

```
GRANT ALL ON tabelle TO PUBLIC;
```

Mit dem SQL-Befehl REVOKE können Sie vergebene Privilegien wieder zurücknehmen, sowohl einzelne als auch alle.

```
REVOKE DELETE ON tabelle FROM benutzer;
```

entzieht dem angegebenen Benutzer das Recht, Datensätze zu löschen. Andere Rechte des Benutzers werden dadurch nicht berührt. Will man dem Benutzer alle Rechte entziehen, gibt man ein:

```
REVOKE ALL ON tabelle FROM benutzer;
```

Sie können sich alle Zugriffsrechte mit dem psql-Kommando \z anzeigen lassen:

```
docgen=# \z
Access permissions for database "docgen"
Relation  | Access permissions
----------+----------------------------------
autoren   | {"=","conni=arwR","apache=arwR"}
```

Der Benutzer conni als Besitzer der Tabelle hat automatisch alle Privilegien, dem Benutzer apache wurden mit GRANT ALL ebenfalls alle Privilegien erteilt. Dabei steht

a für INSERT (append),
r für SELECT (read),
w für UPDATE (write),
d für DELETE,
R für Rule,
x für REFERENCES,
t für TRIGGER.

Es ist nicht möglich, für einzelne Spalten einer Tabelle Zugriffsrechte zu vergeben. Dies ist nur über einen Umweg möglich: Man erzeugt eine View mit den Spalten, die der Benutzer manipulieren darf, und vergibt die Rechte auf diese View. Views werden uns im Kapitel 14 beschäftigen.