6.2 Einfügen, Löschen und Ändern von Daten

Nachdem das Tabellengerüst erzeugt wurde, sollen jetzt Daten in die Tabellen gefüllt und später verändert werden können. Zum Einfügen von Daten wird der INSERT-Befehl genutzt. In der einfachsten Form dieses Befehls ist es wichtig, die Reihenfolge der Attribute in der Tabellendefinition zu kennen. Die Daten der Tabelle Verkäufer aus Abb. 6.1 können wie folgt eingefügt werden:

```
INSERT INTO Verkaeufer
 VALUES(1001,'Udo','Junior',1500);
INSERT INTO Verkaeufer
 VALUES(1002,'Ute','Senior',1900);
INSERT INTO Verkaeufer
 VALUES(1003,'Uwe','Senior',2000)
```

Die generelle Form des Befehls lautet:

```
INSERT INTO <Tabellenname>
   VALUES (<WertFürSpalte1>, ... , <WertFürLetzteSpalte>)
```

Zur Trennung von SQL-Befehlen wird in diesem Buch ein Semikolon genutzt. Das Trennsymbol für Befehle ist im Standard nicht festgeschrieben und kann in Datenbank-Managementsystemen variieren.

Es muss für jede Spalte ein Wert angegeben werden. Etwas flexibler ist die Variante des INSERT-Befehls, bei der die Spalten angegeben werden, in die Werte eingetragen werden sollen. Zwei Beispiele sind:

```
INSERT INTO Verkaeufer(Vnr,Name,Status)
 VALUES(1004,'Ulf','Junior');
INSERT INTO Verkaeufer(Vnr,Gehalt,Name)
 VALUES(1005,1300,'Urs')
```

Die generelle Form lautet:

```
INSERT INTO <Tabellenname> (<Spaltenname1>, ... ,
                            <SpaltennameK>)
  VALUES (<WertFürAngegebeneSpalte1>, ... ,
                <WertFürAngegebeneSpalteK>)
```

Es stellt sich die Frage, was mit Spalten passiert, die nicht im INSERT-Befehl genannt sind. Dabei gibt es zwei Alternativen:

- Sind die Attribute in der bisher bekannten Form definiert, wird in die Spalten ein NULL-Wert eingetragen.

- Ist das Attribut mit einem Standard-Wert definiert, wird dieser Standardwert genommen. Die Definition eines Attributs mit einem Standardwert sieht beispielhaft wie folgt aus:

```
Status VARCHAR(7) DEFAULT 'Junior'
```

Nachdem man Werte in die Tabellen eingetragen hat, möchte man sich diese gerne ansehen. Dazu muss eine Anfrage an die Datenbank formuliert werden. Ohne weiteren Details von Anfragen aus den folgenden Kapiteln vorzugreifen, sei hier eine einfache Anfrageform zur Ausgabe des vollständigen Tabelleninhalts angeben. Sie lautet

```
SELECT * FROM <Tabellenname>
```

Gehen wir davon aus, dass das Attribut Status mit dem angegebenen Standardwert definiert ist, bekommt man als Ergebnis der Anfrage

```
SELECT * FROM Verkaeufer
```

folgende Ausgabe

```
       VNR NAME   STATUS       GEHALT
---------- ------ ------- ----------
      1001 Udo    Junior         1500
      1002 Ute    Senior         1900
      1003 Uwe    Senior         2000
      1004 Ulf    Junior
      1005 Urs    Junior         1300
```

Man erkennt, dass NULL-Werte hier einfach als leere Ausgaben erscheinen, die damit in der Ausgabe nicht vom leeren Text zu unterscheiden sind.

Versucht man einen Tabelleneintrag zu machen, der gegen eine der Constraints, z. B. gegen die Eindeutigkeit des Primärschlüssels verstößt, so wird die Aktion mit einer Fehlermeldung abgebrochen. Wird z. B. folgende Eingabe versucht

```
INSERT INTO Verkaeufer
  VALUES (1001,'Alf','Senior',3000)
```

kann die Reaktion wie folgt aussehen

```
Verstoß gegen Eindeutigkeit, Regel
(SCOTT.SYS_C003014)
```

Durch den „Verstoß gegen die Eindeutigkeit“ wird die Verletzung der gewünschten Primärschlüsseleigenschaft angezeigt. Im Detail steht Scott für den Eigentümer der Tabelle, d. h. die Tabelle wurde unter dem Nutzer Scott angelegt und SYS_C0003014 für den intern vergebenen Namen für das Primärschlüssel-Constraint.

Zur Vervollständigung der Tabellen gehören folgende Einträge in die Tabelle Kunde.

```
INSERT INTO Kunde VALUES(1,'Egon',1001);
INSERT INTO Kunde VALUES(2,'Erwin',1001);
INSERT INTO Kunde VALUES(3,'Erna',1002)
```

Versucht man, einen Kunden mit einem Betreuer einzutragen den es nicht gibt, z. B.

```
INSERT INTO Kunde VALUES(4,'Edna',999);
```

wird der Eintrag nicht vorgenommen und man erhält z. B. die Fehlermeldung

```
Verstoß gegen Constraint (SCOTT.FK_KUNDE).
Übergeordn. Schlüssel nicht gefunden
```

Man beachte, dass hier der Name des verletzten Constraints ausgegeben wird.

Interessant ist der Fall, bei dem ein NULL-Wert für die Referenz zum Betreuer angegeben wird. Diese Möglichkeit wird also explizit in SQL erlaubt. D. h., die folgenden Befehle

```
INSERT INTO Kunde VALUES(4,'Edna',NULL);
SELECT * FROM Kunde
```

führen zu folgender Ausgabe

```
       KNR NAME       BETREUER
---------- ------- ----------
         1 Egon          1001
         2 Erwin         1001
         3 Erna          1002
         4 Edna
```

Neben den vorgestellten `INSERT`-Varianten, gibt es eine dritte Variante, bei der das Ergebnis einer Anfrage zeilenweise in eine Tabelle eingetragen wird. Die Syntax lautet:

`INSERT INTO` <Tabellenname> <Anfrage>

Anfragen werden detailliert ab dem folgenden Kapitel besprochen. Zu dieser Einfügemöglichkeit ist sehr kritisch anzumerken, dass dabei eine Kopie von schon existierenden Daten erzeugt wird und man dem Problem der Redundanz Tür und Tor öffnet. Diese Einfügemöglichkeit ist nur mit sehr viel Bedacht einzusetzen. Oftmals gibt es andere Lösungsmöglichkeiten, wie den Einsatz von Views, siehe Kap. 11.

Nachdem Daten in Tabellen eingetragen wurden, kann der Wunsch bestehen, die Inhalte zu ändern. Hierfür steht der `UPDATE`-Befehl zur Verfügung. Soll für die Tabelle Kunde bei der Kundin mit der Knr 4 der Name auf Edwina geändert und ihr als Betreuer

der Verkäufer mit der Vnr 1002 zugeordnet werden, kann dies mit folgendem Befehl geschehen:

```
UPDATE Kunde
 SET Name='Edwina',
     Betreuer=1002
 WHERE Name='Edna'
```

Die allgemeine Syntax sieht wie folgt aus:

```
UPDATE <Tabellenname>
   SET <SpaltennameI> = <Wert> | <Anfrage>, ...
       <SpaltennameJ> = <Wert> | <Anfrage>
   WHERE <Bedingung>
```

Bei der Ausführung wird für jede Zeile der betroffenen Tabelle untersucht, ob die Bedingung erfüllt ist. Ist dies der Fall, wird die im `SET`-Block beschriebene Änderung durchgeführt. Dies geschieht nur, wenn kein Constraint verletzt wird, sonst gibt es eine Fehlermeldung, und keine Änderung wird ausgeführt.

Die mit <Anfrage> beschriebene Möglichkeit, einen Wert berechnen zu lassen, wird in den späteren Unterkapiteln deutlicher, in denen der Aufbau und das Ergebnis von Anfragen beschrieben werden. Man kann sich hier bereits vorstellen, dass nur Anfragen genutzt werden können, die exakt ein Ergebnis zurückliefern. Eine genauere Beschreibung der Möglichkeiten, die Bedingung anzugeben, erfolgt ebenfalls in den weiteren Unterkapiteln. Grundsätzlich stehen in der Bedingung zu überprüfende Eigenschaften von Attributwerten, die mit AND und OR verknüpft sowie mit NOT negiert werden können.

Betrachtet man die letzte Änderung der Tabelle kritisch, so kann man feststellen, dass die Bedingung nicht optimal ist. Falls es mehrere Kunden mit dem Namen Edna gäbe, würden die beschriebenen Änderungen für alle diese Kunden vorgenommen. Eine wesentlich bessere Bedingung wäre `Knr = 4` gewesen.

Lässt man die WHERE-Bedingung ganz weg, wird sie als „wahr“ interpretiert, d. h. die Änderungen beziehen sich auf alle Zeilen. Innerhalb des SET-Blocks wird zunächst für jede Zuweisung die rechte Seite ausgewertet und dann der Wert des auf der linken Seite genannten Attributs entsprechend verändert. Dies ermöglicht auch eine Änderung eines Attributwertes in Abhängigkeit von seinem alten Wert. Durch den folgenden Befehl kann man z. B. das Gehalt aller Verkäufer um 5 % erhöhen.

```
UPDATE Verkaeufer
 SET Gehalt=Gehalt * 1.05
```

Die resultierende Tabelle sieht wie folgt aus:

```
       VNR NAME   STATUS       GEHALT
---------- ------ ------- ----------
      1001 Udo    Junior         1575
      1002 Ute    Senior         1995
      1003 Uwe    Senior         2100
      1004 Ulf    Junior
      1005 Urs    Junior         1365
```

Man erkennt, dass irgendwelche Operationen auf NULL-Werten, hier z. B. die Multiplikation, keinen Effekt haben. Das Ergebnis bleibt NULL.

Nach dem Einfügen und Verändern fehlt noch als dritter Befehl der Befehl zum Löschen von Daten. Dies geschieht durch `DELETE`. Will man z. B. den Kunden mit der Knr 3 löschen, so kann man dies durch folgenden Befehl erreichen:

```
DELETE FROM Kunde
 WHERE Knr=3
```

Die allgemeine Syntax lautet:

```
DELETE FROM <Tabellenname>
  WHERE <Bedingung>
```

Für die ausgewählte Tabelle wird für jede Zeile geprüft, ob die Auswertung der Bedingung „wahr“ ergibt. Ist dies der Fall, so wird versucht, die Zeile zu löschen. Sollte für eine Zeile die Bedingung erfüllt sein und kann diese wegen irgendwelcher Randbedingungen nicht gelöscht werden, gibt es eine Fehlermeldung und es findet kein Löschvorgang statt.

Diese Situation soll durch ein Beispiel konkretisiert werden. Dabei wird von folgenden Tabelleninhalten ausgegangen:

```
SELECT * FROM Verkaeufer;
SELECT * FROM Kunde
```

führen zu folgenden Ausgaben

```
       VNR NAME   STATUS       GEHALT
---------- ------ ------- ----------
      1001 Udo    Junior         1575
      1002 Ute    Senior         1995
      1003 Uwe    Senior         2100
      1004 Ulf    Junior
      1005 Urs    Junior         1365


       KNR NAME      BETREUER
---------- ------- ----------
         1 Egon          1001
         2 Erwin         1001
         4 Edwina        1002
```

Jetzt soll der Verkäufer mit der Vnr 1001 gelöscht werden. Der Befehl lautet

```
DELETE FROM Verkaeufer
 WHERE Vnr=1001
```

Die Ausgabe dazu kann wie folgt aussehen:

```
Verstoß gegen Constraint (SCOTT.FK_KUNDE).
Untergeordneter Datensatz gefunden.
```

Das Datenbank-Managementsystem erkennt, dass es in der Tabelle Kunde mindestens einen davon abhängigen Datensatz gibt und verweigert die Ausführung des Löschbefehls. Man kann daraus ableiten, dass der Fremdschlüssel zwar in der Tabelle Kunde definiert wird, diese Information der Datenabhängigkeit aber auch ein nicht direkt sichtbares Constraint für die Tabelle Verkäufer liefert. D. h., dass die Kenntnis der Definition der Tabelle Verkäufer alleine nicht ausreicht, um bestimmen zu können, ob eine gewisse Löschoperation ausgeführt werden kann.

Mit dem bisherigen Wissen ist es aber trotzdem möglich, den Verkäufer mit der Vnr 1001 zu löschen. Dazu müssen zunächst die Betreuerinformationen in der Tabelle Kunde, die sich auf 1001 beziehen auf einen anderen Wert, also die VNr von einem anderen Verkäufer oder NULL, gesetzt werden. Danach kann der Verkäufer mit der Vnr 1001 ohne Probleme gelöscht werden. Die zugehörigen SQL-Befehle lauten:

```
UPDATE Kunde
 SET Betreuer=NULL
 WHERE Betreuer=1001;
DELETE FROM Verkaeufer
 WHERE Vnr=1001
```

Das hier vorgestellte Verfahren ist grundsätzlich immer anwendbar, man muss allerdings beachten, dass in realen Projekten die Datenabhängigkeit wesentlich komplexer sein kann. Die Änderungen der Tabelle Kunde könnten sich z. B. auf weitere Tabellen fortpflanzen, vergleichbar dem Domino-Prinzip, bei dem ein Stein angestoßen wird und alle davon abhängigen Steine auch nacheinander umfallen. Da diese Datenanalyse von Hand eventuell sehr aufwendig ist, gibt es eine Alternative bei der Definition von Fremdschlüsseln. Diese Definition sieht für die Tabelle Kunde wie folgt aus:

```
CREATE Table Kunde(
 Knr INTEGER,
 Name Varchar(7),
 Betreuer INTEGER,
 PRIMARY KEY (Knr),
 CONSTRAINT FK_Kunde
  FOREIGN KEY (Betreuer)
   REFERENCES Verkaeufer(Vnr)
   ON DELETE CASCADE
)
```

Durch den Zusatz ON DELETE CASCADE wird der Fremdschlüssel in der Form erweitert, dass, wenn der übergeordnete Datensatz gelöscht wird, damit auch alle davon abhängigen Datensätze in dieser Tabelle gelöscht werden.

Ausgehend von der gleichen Situation, die vor dem gescheiterten DELETE-Befehl vorlag, führt das folgende SQL-Skript

```
DELETE FROM Verkaeufer
 WHERE Vnr=1001;
SELECT * FROM Verkaeufer;
SELECT * FROM Kunde
```

zu folgender Ausgabe

```
       VNR NAME   STATUS      GEHALT
---------- ------ ------- ----------
      1002 Ute    Senior        1995
      1003 Uwe    Senior        2100
      1004 Ulf    Junior
      1005 Urs    Junior        1365

KNR NAME      BETREUER
 ---------- ------- ----------
          4 Edwina        1002
```

Man sieht, dass alle Kunden mit dem Betreuer 1001 gelöscht wurden. Diese Form der Fremdschlüsseldefinition führt alle Löscharbeiten automatisch aus. Dabei gilt weiterhin, dass die gesamte Aktion nicht durchgeführt wird, falls es an nur einer Stelle Widerstand gegen das Löschen geben sollte, da ein Constraint verletzt wird.

Beim aufmerksamen Lesen des letzten Beispiels fällt kritisch auf, dass es häufig nicht gewünscht wird, dass Kundeninformationen verloren gehen. Man muss sich bei der Implementierung der Tabellenstrukturen möglichst für eine einheitliche Art der Fremdschlüsseldefinition mit oder ohne ON DELETE CASCADE entscheiden. Wenn man weiß, dass später auch unerfahrene Personen Zugang zu kritischen Daten haben müssen, sollte man genau überlegen, ob das eigentlich sehr praktische ON DELETE CASCADE wirklich die bessere Lösung ist, da der vorher beschriebene Dominoeffekt, dem man ohne diese Ergänzung von Hand abarbeiten muss, mit ON DELETE CASCADE schnell zu gravierenden und meist nur schwer wieder behebbaren Datenverlusten führen kann.

Im SQL-Standard gibt es folgende Möglichkeiten, die Reaktion beim DELETE in den FOREIGN KEYs nach ON DELETE zu spezifizieren:

► **Umgangsmöglichkeiten mit abhängigen Daten beim Löschen**

NO ACTION: entspricht der Ursprungseinstellung, das Löschen wird abgelehnt, wenn ein abhängiger Datensatz existiert

CASCADE: entspricht der vorgestellten Löschfortpflanzung

SET NULL: für abhängige Datensätze wird die Referenz auf den gelöschten Datensatz automatisch auf NULL gesetzt

SET DEFAULT: wenn es einen Default-Wert gibt, wird dieser im abhängigen Datensatz eingetragen; existiert ein solcher Wert nicht, wird der Löschvorgang nicht durchgeführt

Beim kritischen Lesen fällt auf, dass man die Überlegungen zum DELETE auch zum UPDATE anstellen kann. Dazu ist anzumerken, dass der SQL-Standard Möglichkeiten wie ON UPDATE CASCADE vorsieht, dieses aber nur bei wenigen großen Datenbank-Managementsystemen implementiert ist. Die Möglichkeiten und Argumentationen zum Für und Wider lassen sich dabei direkt von der Diskussion zum DELETE auf eine Diskussion zum UPDATE übertragen.

Vergleichbare Überlegungen wie zu DELETE und UPDATE kann man auch zum Löschen von Tabellen anstellen. Mit dem Befehl

DROP TABLE <Tabellenname> RESTRICT

wird die Tabelle mit dem Namen < Tabellenname > nur gelöscht, wenn es keine anderen Tabellen gibt, die von dieser Tabelle abhängig sind. D. h. es gibt keine andere Tabelle, in der ein Fremdschlüssel auf die zu löschende Tabelle referenziert. Gibt es solch eine Tabelle, wird der Löschbefehl nicht durchgeführt.

Die Nutzung des Schlüsselworts RESTRICT macht das Löschen sehr sicher, da man so nicht zufällig eine zentrale Tabelle mit vielen Abhängigkeiten löschen kann. Allerdings bedeutet dies auch für Löschvorgänge, bei denen mehrere Tabellen gelöscht werden sollen, dass man zunächst die Tabelle herausfinden muss, auf die es keine Verweise gibt und dann die weiteren Tabellen schrittweise löscht. Dies kann man sich wie eine Kette von Dominos vorstellen, bei der man vorsichtig die letzten Dominos entfernt, da man möchte, dass die Dominoschlange beim Umkippen in eine andere Richtung läuft.

Alternativ gibt es den nicht von allen Datenbank-Managementsystemen unterstützen Befehl

DROP TABLE <Tabellenname> CASCADE

mit dem garantiert wird, dass die Tabelle mit dem Namen <Tabellenname> gelöscht wird. Durch das CASCADE wird ausgedrückt, dass Tabellen, die Referenzen auf die zu löschende Tabelle haben, so verändert werden, dass diese Referenzen gelöscht werden. Dies bedeutet konkret, dass Constraints mit Fremdschlüsseln, die sich auf die zu löschende Tabelle beziehen, mit gelöscht werden. Dabei bleiben aber alle anderen Daten in der vorher abhängigen Tabelle erhalten, man kann sicher sein, dass die zu löschende Tabelle mit allen Spuren im System gelöscht wird.

Generell gibt es auch den kürzeren Befehl

```
DROP TABLE <Tabellenname>
```

dabei ist allerdings zu beachten, dass dieser abhängig vom Datenbank-Managementsystem einmal in der Form mit `RESTRICT` und in einem anderen System in der Form mit `CASCADE` interpretiert wird.

6.3 Datentypen in SQL

In den vorherigen Unterkapiteln wurden bereits einige Datentypen benutzt, die sehr häufig in Datenbanken zu finden sind. In diesem Unterkapitel werden wichtige Informationen zu weiteren Typen zusammengefasst. Dabei ist zu beachten, dass neben den Basistypen in den schrittweisen Erweiterungen des Standards Möglichkeiten zur Definition eigener Typen aufgenommen wurden. Mit diesen Typen wird aber häufig gegen die erste Normalform verstoßen. Sie sind dann sinnvoll, wenn man genau weiß, welche Auswertungen mit der Datenbank gemacht werden sollen. Generell gilt, dass die Arbeit mit eigenen Typen das spätere Erstellen von Anfragen leicht komplizierter machen kann. Mit den in diesem Unterkapitel vorgestellten Typen können ohne Probleme alle Standardtabellen typischer Datenbank-Einsatzbereiche beschrieben werden.

Selbst bei den Standarddatentypen muss man aufpassen, ob und teilweise unter welchen Namen sie im Datenbank-Managementsystem unterstützt werden. Einen guten Überblick über das standardisierte Verhalten und die Umsetzung in führenden Datenbank-Managementsystemen findet man in [Tür03].

Generell gilt, dass man ohne Typen bzw. mit nur einem allgemeinen Typen auskommen kann. Dies ist ein Typ, der Texte beliebiger Länge aufnehmen kann. Man kann Zahlen, z. B. ‚`123`' und Wahrheitswerte wie ‚`true`' einfach als Texte oder Strings interpretieren. Dieser Ansatz wird in SQL nicht gewählt, da man als Randbedingung präziser angeben möchte, welcher Typ erwartet wird. Weiterhin erleichtert die Nutzung der Typen die Werteänderungen. Es ist z. B. zunächst unklar, was ein Minus-Zeichen für Texte bedeuten soll. Geht man davon aus, dass Zahlen gemeint sind, müssten die Texte zunächst in Zahlen verwandelt, dann subtrahiert und letztendlich das Ergebnis in einen String zurückverwandelt werden.

Es bleibt aber festzuhalten, dass man eventuell nicht vorhandene Datentypen durch Texte simulieren kann. Dieser Ansatz ist z. B. bei einigen größeren Datenbank-Managementsystemen notwendig, wenn man Wahrheitswerte darstellen möchte, da der SQL-Standardtyp `BOOLEAN` nicht immer vorhanden ist.

Generell sollte man vor der Erstellung einer Tabelle deshalb prüfen, welche der vorgestellten Datentypen genutzt werden können und welche Typen es noch zusätzlich im Datenbank-Managementsystem gibt. Dabei ist daran zu denken, dass man mit jedem benutzten Typen, der nicht unter diesem Namen zum Standard gehört, Schwierigkeiten beim Wechsel oder der parallelen Nutzung eines anderen Datenbank-Managementsystems bekommen kann.