Kapitel 3: Datenmanipulationssprache SQL

SQL (Structured Query Language) ist die Standardsprache für die Datendefinition und Datenmanipulation in relationalen Datenbanksystemen. Sie umfaßt:

- Interaktives (”stand−alone”) SQL (z.B. Oracle SQL*Plus)
- Eingebettetes (”embedded”) SQL (für die gängigen Programmiersprachen)
- Anweisungen zur Integritätssicherung und Zugriffskontrolle
- Anweisungen zur physischen Datenorganisation (Indizes etc.)

Zusätzlich in vielen Produkten:

- Integration von SQL in ”4th Generation Language” (z.B. Oracle PL/SQL)

3.1 Einfache Datendefinition

”Grobsyntax” (in EBNF = Extended Backus−Naur Form):

```
CREATE TABLE [user "."] table "("  column_element {"," column_element}
                                   {"," table_constraint} ")"
```

mit

```
column_element =     column data_type [DEFAULT expr] [column_constraint]
column_constraint =  [NOT NULL]
                     [PRIMARY KEY | UNIQUE]
                     [REFERENCES [user "."] table [ "(" column ")" ]]
                     [CHECK "(" condition ")" ]
table_constraint =   [ (PRIMARY KEY | UNIQUE) "(" column {"," column} ")" ]
                     [ FOREIGN KEY "(" column {"," column} ")"
                        REFERENCES [user "."] table [ "(" column {"," column} ")" ]
                     [CHECK "(" condition ")" ]
```

Beispiele:

```
CREATE TABLE Produkte ( PNr INTEGER CHECK (PNr > 0) PRIMARY KEY,
                        Bez VARCHAR(30) NOT NULL UNIQUE,
                        Gewicht FLOAT CHECK (Gewicht > 0.0),  ...)
CREATE TABLE Bestellungen (
   BestNr INTEGER CHECK (BestNr > 0) PRIMARY KEY,
   Monat INTEGER NOT NULL CHECK (Monat BETWEEN 1 AND 12),  ...,
   FOREIGN KEY (PNr) REFERENCES Produkte (PNr),
   FOREIGN KEY (KNr) REFERENCES Kunden (KNr),
   UNIQUE (Monat, Tag, PNr, KNr) )
```

Unterschiede zum ”reinen” Relationenmodell:

- Die Festlegung eines Primärschlüssels wird nicht erzwungen.
  Wenn aber ein Primärschlüssel spezifiziert ist, dann wird die Einhaltung der Primärschlüsselbedingung garantiert.
  Analoges gilt für die Fremdschlüsselbedingung.
- Falls kein Primärschlüssel (und kein Schlüsselkandidat) spezifiziert ist, dürfen Tabellen Duplikate enthalten, d.h. es handelt sich um Multimengen (”bags”).

3.2 Anfragen (Queries)

"Grobsyntax" von SQL−Abfragen:

select_block { (UNION | INTERSECT | EXCEPT) select_block }
[ORDER BY result_column [ASC | DESC] {"," result_column [ASC | DESC]}
mit
select_block = SELECT [ALL | DISTINCT] (column | expression) {"," (column | expression)}
FROM table [correlation_var] {"," table [correlation_var]}
[WHERE search_condition]
[GROUP BY column {"," column} [HAVING search_condition]]

Vergleich mit der Relationenalgebra und dem Tupel−Relationenkalkül:

SELECT A, B, ... FROM R, S, ..., T, ... WHERE F
(so daß A, B, ... zu R, S, ... gehören, nicht aber zu T, ..., und F über R, S, ..., T, ... definiert ist)
enspricht in der Relationenalgebra
π [A, B, ...] (σ [F] (R $\times$ S $\times$ ... $\times$ T $\times$ ...)
und im Tupel−Relationenkalkül
$\{x.A, y.B, ... \mid x \in R \wedge y \in S \wedge ... \wedge \exists z: z \in T ... \wedge F(x, y, ..., z, ...)\}$

Achtung:
- Alle Joinbedingungen müssen in SQL explizit in der WHERE−Klausel angegeben werden (vgl. Tupel−Relationenkalkül).
- Standardmäßig erfolgt keine Duplikateliminierung; diese kann mit DISTINCT explizit spezifiziert werden.
  Bei UNION gibt es u.U. zwei Varianten: mit (UNION ALL) und ohne Duplikateliminierung.

Beispiele:

1) Finden Sie (die Namen) alle(r) Kunden mit negativem Saldo.
   → SELECT KNr, Name, Stadt, Saldo, Rabatt FROM Kunden WHERE Saldo < 0.0
   oder: SELECT * FROM Kunden WHERE Saldo < 0.0
   bzw.: SELECT Name FROM Kunden WHERE Saldo < 0.0

2) Finden Sie die Namen aller Kunden, die eine unbezahlte Bestellung haben, die vor Anfang Oktober erfolgte.
   → SELECT Name FROM Kunden, Bestellungen
   WHERE Monat < 10 AND Status <> 'bezahlt'
   AND Kunden.KNr = Bestellung.KNr

3) Finden Sie die Namen der Homburger Kunden, die seit Anfang September ein Produkt aus Homburg geliefert bekommen haben, jeweils mit der Bezeichnung des entsprechenden Produkts.
   → SELECT Name FROM Kunden, Bestellungen, Produkte
   WHERE Stadt='Homburg'
   AND Monat>9 AND Status<>'neu'
   AND Lagerort='Homburg'
   AND Kunden.KNr=Bestellungen.KNr AND Bestellungen.PNr=Produkt.PNr

4) Finden Sie die Rechnungssume der Bestellung mit BestNr 111 (ohne auf das Attribut Summe der Relation Bestellungen zuzugreifen).
   → SELECT Menge*Preis*(1.0−Rabatt) FROM Bestellungen, Produkte, Kunden
   WHERE BestNr=111
   AND Bestellungen.PNr=Produkte.PNr AND Bestellungen.KNr=Kunden.KNr

Allgemeinere Form der FROM−Klausel: Korrelationsvariable (Tupelvariable)

a) Eindeutige Benennung mehrerer ”Inkarnationen” derselben Relation (in θ−Joins)

Beispiel:
Finde alle Paare von Kunden, die in derselben Stadt wohnen.

```
→ SELECT K1.Name, K2.Name FROM Kunden K1, Kunden K2
  WHERE K1.Stadt=K2.Stadt AND K1.KNr < K2.KNr
```

b) Outer Join

Beispiel:
Gib alle Kunden mit 5 % Rabatt zusammen mit ihren Bestellungen und den bestellten Produkten aus, und zwar auch wenn für einen Kunden gar keine Bestellungen vorliegen.

```
→ SELECT *
  FROM Kunden FULL OUTER JOIN Bestellungen
          ON (Kunden.KNr=Bestellungen.KNr),
          Produkte
  WHERE Rabatt = 0.05
  AND Produkte.PNr=Bestellungen.PNr
```

Allgemeinere Form der WHERE−Klausel: Komplexere Suchprädikate

Achtung: SQL hat (zu) viele Alternativen für dieselbe Anfrage !

Vereinfachte Formulierung von Bereichsanfragen (”range queries”)

Beispiel:
Finden Sie die Kunden, deren Rabatt zwischen 10 und 20 Prozent liegt.
→ SELECT * FROM Kunden WHERE Rabatt BETWEEN 0.10 AND 0.20

Pattern−Matching in Zeichenketten

mit % als Platzhalter (”wild card”) für eine beliebige Zeichenkette der Länge ≥ 0
und _ als Platzhalter für ein beliebiges Zeichen

Beispiele:

1) Finden Sie alle Kunden, deren Name mit A beginnt.
   → SELECT * FROM Kunden WHERE Name LIKE ’A%’

2) Finden Sie alle Kunden mit Namen Meier, Maier, Meyer, ...
   → SELECT * FROM Kunden WHERE Name LIKE ’M_ _er’

weitergehende Möglichkeiten in einigen Produkten (z.B. Ingres)

Beispiel 2:

   → SELECT * FROM Kunden WHERE Name LIKE ’M[a,e][i,y]er’

Test auf Nullwert

Beispiel:

Finden Sie alle Produkte, die grundsätzlich in keinem Lager geführt werden.
→ SELECT * FROM Produkte WHERE Lagerort IS NULL

IS NULL ist das einzige Vergleichsprädikat, das von einem Nullwert erfüllt wird.

Konsequenz: Die Anfrage
SELECT * FROM Produkte WHERE Vorrat >= 100 OR Vorrat < 100
liefert (nur) alle Tupel der Relation Produkte, deren Wert bzgl. des Attributs Vorrat kein Nullwert ist.

Test auf Mitgliedschaft eines Werts in einer Menge (IN−Prädikat)

Einfaches Beispiel:

Finden Sie alle Kunden aus Homburg, Merzig und Saarlouis.
→ SELECT * FROM Kunden Where Stadt IN ('Homburg', 'Merzig', 'Saarlouis')

Allgemeine Form mit *"Subqueries"* (geschachtelten Select−Blöcken):
(column | expression) [NOT] IN "(" (value {"," value} | select_block) ")"

Beispiele:

1) Finden Sie die Namen aller Kunden, die eine unbezahlte Bestellung haben, die vor Anfang Oktober erfolgte.
→ SELECT Name FROM Kunden
WHERE KNr IN (SELECT KNr FROM Bestellungen
WHERE Status<>'bezahlt' AND Monat < 10)

2) Finden Sie alle Kunden aus Städten, in denen es mindestens zwei Kunden gibt.
→ SELECT * FROM Kunden K1
WHERE Stadt IN (SELECT Stadt FROM Kunden K2
WHERE K2.KNr <> K1.KNr)
Die Subquery wird in diesem Fall auch als "korrelierte Subquery" bezeichnet.

Achtung:
Es handelt sich hierbei um Tests, ob ein Wert in einer Menge von Werten enthalten ist.
Ein Test, ob ein Tupel (mit mindestens zwei Attributen) in einer Menge von Tupeln enthalten ist, ist auf diese Weise nicht möglich.
Teilmengentests zwischen Mengen sind in SQL ebenfalls nicht möglich.

Beispiel:

Finden Sie die Kundennummern der Kunden, die alle überhaupt lieferbaren Produkte irgendwann bestellt haben.

Ansatz:

```
SELECT KNr FROM Kunden
WHERE   (  SELECT PNr FROM Produkte )
           IN
           (  SELECT PNr FROM Bestellungen
              WHERE KNr = Kunden.KNr )
```

ist keine korrekte SQL−Anweisung!
(IN wäre hier im Sinne von $\subseteq$ gemeint, nicht − wie zuvor − im Sinne von $\in$.)

”Quantifizierte” Vergleiche zwischen einem Wert und einer Menge

Die Bedingung *Wert* θ*ANY Menge* mit $\theta \in \{=, \neq, <, >, \leq, \geq\}$ ist erfüllt, wenn es in der Menge ein Element gibt, für das *Wert* θ *Element* gilt.
(=ANY ist äquivalent zu IN.)

Die Bedingung *Wert* θ*ALL Menge* mit $\theta \in \{=, \neq, <, >, \leq, \geq\}$ ist erfüllt, wenn für alle Elemente der Menge gilt: *Wert* θ *Element*.
(<>ALL ist äquivalent zu NOT IN.)

Beispiel: Finden Sie die Kunden mit dem geringsten Rabatt.

```
→ SELECT * FROM Kunden
    WHERE Rabatt <=ALL (SELECT Rabatt FROM Kunden)
```

Achtung:
Das Prädikat *Wert =ALL Menge* ist nur bei einer einelementigen Menge erfüllbar.
Die Anfrage

```
SELECT KNr FROM Bestellungen
WHERE PNr =ALL (SELECT PNr FROM Produkte)
```

kann nur dann ein nichtleeres Resultat liefern, wenn es nur ein einziges Produkt gibt!

Test, ob eine Menge leer oder nichtleer ist (Existenztest)

Beispiele:

1) Finden Sie die Namen aller Kunden, die eine unbezahlte Bestellung haben, die vor Anfang Oktober erfolgte.

```
→ SELECT Name FROM Kunden
  WHERE EXISTS (   SELECT * FROM Bestellungen
                   WHERE Status <> 'bezahlt' AND Monat < 10
                   AND Bestellungen.KNr = Kunden.KNr )
```

2) Finden Sie die Kunden, für die keine Bestellung registriert ist.

```
→ SELECT * FROM Kunden
  WHERE NOT EXISTS (   SELECT * FROM Bestellungen
                       WHERE Bestellungen.KNr = Kunden.KNr )
```

Suchbedingungen der Form ”... für alle ...” (Verwendung des Allquantors im Relationenkalkül bzw. der Division in der Relationalgebra) können in SQL mit NOT EXISTS ”emuliert” werden.

Beispiel:

Finden Sie die Kunden, die alle überhaupt lieferbaren Produkte irgendwann bestellt haben.

```
→   SELECT * FROM Kunden
    WHERE   NOT EXISTS
              (  SELECT * FROM Produkte
                 WHERE   NOT EXISTS
                           (  SELECT * FROM Bestellungen
                              WHERE Bestellungen.PNr = Produkte.PNr
                              AND Bestellungen.KNr = Kunden.KNr ) )
```

Semantik von SQL: Abbildung auf den Tupelrelationenkalkül

... unter Vernachlässigung von Multimengen, Nullwerten, u.ä.
und der Annahme der eindeutigen Benennung von Tupelvariablen

μ [SELECT A, B, ... FROM R, S, ..., T, ... WHERE F]
(so daß A, B, ... zu R, S, ... gehören, nicht aber zu T, ..., und F über R, S, ..., T, ... definiert ist)
=
$\{x.A, y.B, ... \mid x \in R \wedge y \in S \wedge \ ... \ \wedge \exists z: z \in T ... \ \wedge \mu[F] \}$

μ [A θ B]
(so daß A, B zu R, S mit Tupelvariablen x, y gehören)
= x.A θ y.B

μ [F AND G] = $\mu[F] \wedge \ \mu[G]$
μ [F OR G] = $\mu[F] \vee \ \mu[G]$
μ [NOT F] = $\neg \ \mu[F]$

μ [A IN Subquery S]
(so daß S die Form SELECT ... FROM ... WHERE F hat und A an t gebunden ist)
= $\exists$ s: s.B=t.A $\wedge \ \mu[S]$

μ [A θALL Subquery S]
(so daß S die Form SELECT ... FROM ... WHERE F hat und A an t gebunden ist)
= $\forall$ s: $\mu[S] \Rightarrow$ t.A θ s.B

μ [A θANY Subquery S]
(so daß S die Form SELECT ... FROM ... WHERE F hat und A an t gebunden ist)
= $\exists$ s: $\mu[S] \wedge$ t.A θ s.B

μ [EXISTS Subquery S]
(so daß S die Form SELECT ... FROM ... WHERE F hat)
= $\exists$ s: $\mu[S]$

Aggregationsfunktionen

Zweck: Abbildung einer (Multi−) Menge von Werten auf einen Wert.

”Grobsyntax”:
(MAX | MIN | AVG | SUM | COUNT) ”(” (ALL | DISTINCT) (column | expression | ”*”) ”)”

Beispiele:

1) Finden Sie den höchsten Rabatt aller Kunden.
   → SELECT MAX (Rabatt) FROM Kunden
   Finden Sie den durchschnittlichen Rabatt aller Kunden.
   → SELECT AVG (Rabatt) FROM Kunden

2) An wievielen Lagerorten werden Produkte gelagert?
   → SELECT COUNT (DISTINCT Lagerort) FROM Produkte

3) Wieviele Kunden haben einen Rabatt von mehr als 15 Prozent?
   → SELECT COUNT (*) FROM Kunden WHERE Rabatt > 0.15

4) Welche Kunden haben einen überdurchschnittlichen Rabatt?
   → SELECT * FROM Kunden
   WHERE Rabatt > SELECT AVG (Rabatt) FROM Kunden

5) Wie hoch ist der Gesamtumsatz?
   → SELECT SUM (Menge*Preis*(1.0−Rabatt))
   FROM Bestellungen, Produkte, Kunden
   WHERE Bestellungen.PNr=Produkte.PNr AND Bestellungen.KNr=Kunden.KNr

Einfache "Built−in"−Funktionen

Zweck: Transformation von Werten

produktspezifisch, z.B. in Oracle:

```
SELECT SUBSTR (Name, INSTR(Name, ' ')+1) FROM Kunden
```

zur Extraktion der Nachnamen (unter der Annahme, daß Name den Vornamen und Nachnamen durch Leerzeichen getrennt enthält)

Ideal: Erweiterbarkeit des Datenbanksystems (siehe Kapitel 6)

Oracle−spezifische Erweiterung zur Textsuche (Oracle ConText)

Die CONTAINS−Funktion liefert für einen Attributwert und eine Textsuchbedingung einen numerischen Score−Wert (z.B. die Anzahl der Treffer oder eine Ähnlichkeitsangabe)

Beispiele:

1) SELECT PNr, Bez FROM Produkte
   WHERE CONTAINS (Beschreibung, 'Internet') > 0 bzw. >= 5

   liefert alle Produkte, deren Beschreibung das Wort 'Internet' enthält
   bzw. mindestens fünfmal enthält

2) SELECT PNr, Bez FROM Produkte
   WHERE CONTAINS (Beschreibung, 'Oracle AND (Internet OR Web) MINUS SQL') > 0

   liefert alle Produkte, deren Beschreibung das Wort 'Oracle' und mindestens eines der Wörter 'Internet' oder 'Web', nicht jedoch das Wort 'SQL' enthält

3) SELECT PNr, Bez FROM Produkte
   WHERE CONTAINS (Beschreibung, 'Oracle*2 OR SQL*1') >= 5

   liefert alle Produkte, deren Beschreibung die Wörter 'Oracle' oder 'SQL' insgesamt mindestens fünfmal enthält, wobei jedes Vorkommen von 'Oracle' doppelt zählen soll

4) SELECT PNr, Bez FROM Produkte
   WHERE CONTAINS (Beschreibung, '$Query') > 0

   liefert alle Produkte, deren Beschreibung ein Wort mit dem Wortstamm 'Query' enthält, also z.B. auch "Querying" oder "Queries"

5) SELECT PNr, Bez FROM Produkte
   WHERE CONTAINS (Beschreibung, '?Oracle') > 0

   liefert alle Produkte, deren Beschreibung ein Wort enthält, das dem Wort 'Oracle' ähnlich ist (wie z.B. 'Orakel')

6) SELECT PNr, Bez FROM Produkte
   WHERE CONTAINS (Beschreibung, '!Oracle') > 0 : 10

   liefert alle Produkte, deren Beschreibung ein Wort enthält, das in phonetischer Hinsicht dem Wort 'Oracle' ähnlich ist (wie z.B. 'Awewreckle'), und zwar nur die (maximal) zehn ähnlichsten Treffer

Aggregation mit Gruppenbildung

Zweck:
Eine Tupelmenge wird aufgrund der Werte eines Attributs oder eines Ausdrucks in ”Gruppen” (Teilmengen) partitioniert.

Beispiel:
Bestimmen Sie für alle Produkte deren Gesamtverkaufszahl (ab Anfang September).
→ SELECT PNr, SUM(Menge) FROM Bestellungen WHERE Monat >= 9
GROUP BY PNr

Zwischenresultat nach der Gruppierung:

PNr	**Gruppe**		
	BestNr	**...**	**Menge**
1	9		100
2	3		4
3	4		1
4	5		10
5	6		50
	10		50
	11		50
6	7		2
7	8		5
	12		10

Endresultat:

PNr	**SUM(Menge)**
1	100
2	4
3	1
4	10
5	150
6	2
7	15

Beispiel mit Auswahl von Gruppen:

Bestimmen Sie alle Kunden mit mindestens zwei Bestellungen seit Anfang Oktober, deren Gesamtwert mindestens 2000 DM betragen hat.

→ SELECT KNr FROM Bestellungen
WHERE Monat >= 10
GROUP BY KNr
HAVING COUNT(*) >= 2 AND SUM(Summe) >= 2000.00

Zwischenresultat nach der Gruppierung:

KNr	**Gruppe**		
	BestNr	**...**	**Summe**
1	6		900.00
	7		180.00
	8		900.00
2	9		1600.00
	10		800.00
3	7		1800.00

nach der Aggregation:

KNr	**COUNT(*)**	**SUM(Summe)**
1	3	1980.00
2	2	2400.00
3	1	1800.00

Endresultat:

KNr
2

weitere Beispiele:

1) Bestimmen Sie für alle Produkte, die mehr als 1000−mal verkauft worden sind, deren Gesamtverkaufszahl (ab Anfang September).
   → SELECT PNr, SUM(Menge) FROM Bestellungen WHERE Monat >= 9
   GROUP BY PNr HAVING SUM(Menge) > 1000

2) Bestimmen Sie für jede Stadt mit mehr als 10 Kunden den durchschnittlichen Rabatt dieser Kunden. Die Ausgabe soll nach Rabatten absteigend sortiert sein.
   → SELECT Stadt, AVG(Rabatt) AS MittlRabatt FROM Kunden
   GROUP BY Stadt HAVING COUNT(*) > 10
   ORDER BY 2 DESC

3) Finden Sie die Kunden, die alle überhaupt lieferbaren Produkte irgendwann bestellt haben.
   → SELECT KNr FROM Bestellungen
   GROUP BY KNr
   HAVING COUNT (DISTINCT PNr) =
   (SELECT COUNT(*) FROM Produkte)

Semantik der Gruppierung / Aggregation von SQL: Abbildung auf Relationenalgebra−Programme

Gegeben sei eine Anfrage der Form

```
SELECT A, f(B)
FROM ...
WHERE ...
GROUP BY A
HAVING P(A, g(C))
```

mit Aggregationsfunktionen f, g und einem Prädikat P.

Sei Q(A,B,C) das Ergebnis der WHERE−Klausel−Auswertung.
Dann ergibt sich das Ergebnis der Gesamtanfrage durch:

```
R (A,B') := {}
for each x ∈ π[A](Q) do
    G_x := π[A,B,C] (σ[A=x] (Q))
    if P(x, g(π[C](G_x))) then
        y := f (π[B](G_x))
        R := R ∪ {(x,y)}
    fi
od
```

Abbildung der Anfrage auf die Prädikatenlogik höherer Ordnung:

$$\{ (x,y) \mid \exists G: y=f(G.B) \wedge \forall t: (t \in G \Rightarrow (t.A=x \wedge (\exists z: z=g(G.C) \wedge P(x,z)) \wedge (\exists q: q \in Q \wedge q.A=x \wedge q.B=t.B \wedge q.C=t.C))) \}$$

Mächtigkeit und Grenzen von SQL (genauer: des SQL92−Standards)

Satz:

SQL ist relational vollständig.

SQL ist nicht Turing−vollständig;
d.h., es gibt berechenbare Funktionen, die nicht in SQL ausdrückbar sind.

Beispiele für Anfragen, die nicht in SQL ausdrückbar sind:

1) Bestimmen Sie die Varianz der Rabatte, die den Kunden gewährt werden.

2) Erstellen Sie eine Liste der folgenden Kundenkategorien:
   alle Kunden mit einem Umsatz zwischen 0.00 und 10000.00 DM,
   alle Kunden mit einem Umsatz zwischen 10000.01 und 20000.00 DM,
   ...
   alle Kunden mit einem Umsatz über 100000.00 DM.
   Für jede Kategorie sind die Kundenanzahl und der Gesamtumsatz auszudrucken.

3) Gegeben sei eine Relation mit direkten Flugverbindungen.
   Bestimmen Sie die Orte, die man mit dem Flugzeug von Frankfurt aus erreichen kann.

Flüge

FlugNr	**Abflugort**	**Zielort**	**...**
LH58	Frankfurt	Chicago	
AA371	Chicago	Phoenix	
DA77	Phoenix	Flagstaff	
. . .			

Ansatz:

```
SELECT Fn.Zielort
FROM Flüge F1, Flüge F2, Flüge F3, ..., Flüge Fn
WHERE  F1.Abflugort = 'Frankfurt' AND
       F1.Zielort = F2.Abflugort AND
       F2.Zielort = F3.Abflugort AND
       ... AND
       Fn−1.Zielort = Fn.Abflugort
```

... ist keine korrekte SQL−Anweisung (für variables n) !

Rekursive Anfragen in SQL3 bzw. Oracle:

Der mit der Jahrtausendwende erwartete Standard SQL3 sieht die Unterstützung linearer Rekursion vor, wobei akkumulierende Berechnungen in die Rekursion eingebettet werden können.

Beispiel:

```
WITH RECURSIVE Verbindungen (Start, Ziel, Gesamtdistanz) AS
   ( ( SELECT Abflugort, Zielort, Distanz
       FROM Flüge
       WHERE Abflugort = 'Frankfurt' )
     UNION
     ( SELECT V.Start, V. Ziel, V.Gesamtdistanz + F.Distanz
       FROM Verbindungen V, Flüge F
       WHERE V.Ziel = F.Abflugort )
   )
SELECT Ziel, Gesamtdistanz
FROM Verbindungen
```

In Oracle ist Rekursion ebenfalls in eingeschränkter Form möglich, und zwar im wesentlichen für die Berechnung transitiver Hüllen und mit einer Ad−hoc−Erweiterung der SQL−Syntax.

Beispiel:

```
SELECT F.Zielort
FROM Flüge F
START WITH Abflugort = 'Frankfurt'
CONNECT BY Abflugort = PRIOR Zielort
```

3.3 Datenmodifikation

Einfügen von Tupeln

”Grobsyntax”:
INSERT INTO table [”(” column {”,” column} ”)”]
(VALUES ”(” expression {”,” expression} ”)” | subselect)

Beispiele:
1) INSERT INTO Kunden VALUES (7, ’Kunz’, ’Neunkirchen’, 0.0, 0.0)
2) INSERT INTO Kunden (KNr, Name, Stadt) VALUES (7, ’Kunz’, ’Neunkirchen’)
3) CREATE TABLE Mahnungen (BestNr ...) mit demselben Schema wie Bestellungen
   INSERT INTO Mahnungen
      SELECT * FROM Bestellungen WHERE Status=’geliefert’ AND Monat<10

Attributwerte, die beim Einfügen nicht spezifiziert sind, werden auf den Default−Wert oder den Nullwert gesetzt.

Ändern von Tupeln

”Grobsyntax”:
UPDATE table [correlation_var]
SET column ”=” expression {”,” column ”=” expression} WHERE search_condition

Beispiele:
1) UPDATE Kunden SET Stadt = ’Saarbrücken’ WHERE KNr=1
2) UPDATE Kunden SET Rabatt = Rabatt + 0.05
   WHERE Saldo > −10000.0
   AND KNr IN SELECT KNr FROM Bestellungen
              GROUP BY KNr HAVING SUM(Summe) > 100000.0

Löschen von Tupeln

”Grobsyntax”:
DELETE FROM table [correlation_var] [WHERE search_condition]

Beispiel: DELETE FROM Bestellungen WHERE Monat < 7

Schemaänderung

”Grobsyntax”:
ALTER TABLE table
ADD column datatype [column_constraint] {”,” column data_type [column_constraint]}

Beispiel: ALTER TABLE Bestellungen ADD Frist INTEGER CHECK (Frist > 0)

Existierende Tupel werden bezüglich neuer Attribute implizit auf den Nullwert gesetzt.

Ergänzende Literatur zu Kapitel 3:

J. Melton, A. Simon, Understanding the New SQL: A Complete Guide, Morgan Kaufmann, 1993

C.J. Date, H. Darwen, A Guide to the SQL Standard, 4th Edition, Addison−Wesley, 1997

Oracle 8: The Complete Reference, Osborne, 1997

M. Negri, S. Pelagatti, L. Sbattella, Formal Semantics of SQL Queries, ACM Transactions on Database Systems, Vol. 16 No. 3, Sept. 1991

M. Gogolla, An Extended Entity−Relationship Model, Chapter 5: Formal Semantics of SQL, Springer−Verlag, Lecture Notes in Computer Science 767, 1994