Kapitel 7: Die Datenbanksprache 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 Skriptsprachen für sog. Stored Procedures (z.B. Oracle PL/SQL)

Zur Historie von SQL:

Entwicklung der Sprache Sequel (Structured English Query Language) in den Siebziger Jahren am IBM Almaden Research Lab sowie der Prototypimplementierung System R.
Erste kommerzielle Produkte seit Anfang der Achtziger Jahre (SQL/DS, Oracle, usw.).
In Form des ANSI- und ISO-Standards SQL-92 (SQL2) produktunabhängig definiert (mit den Stufen „Entry Level", „Intermediate Level" und „Full Level").
Von allen marktrelevanten Datenbanksystemen unterstützt (Oracle, IBM DB2, Informix, Sybase, MS SQL Server) sowie in stark eingeschränktem Umfang auch von dem via GNU-Lizenz kostenfrei verfügbaren System MySQL.
Mit dem neueren ANSI-Standard SQL-99 (SQL3) um mächtige objekt-orientierte bzw. objekt-relationale Features erweitert (die allerdings – soweit sie über den sog. „Core" hinausgehen – in Produkten uneinheitlich unterstützt werden).
Ein weiteres Standard-Release SQL-2003, in dem auch XML-Unterstützung vorgesehen ist, ist in Vorbereitung.

Kapitel 7 orientiert sich primär an SQL-92. Kapitel 10 geht auf die objekt-relationalen Erweiterungen von SQL-99 ein.

7.1 Einfache Datendefinition

Exkurs: Syntaxbeschreibung

Die Syntax einer formalen Sprache wird durch eine **kontextfreie Grammatik** beschrieben, ein 4-Tupel $G = (\Sigma, V, P, S)$ bestehend aus:

- einer endlichen Menge Σ von Terminalsymbolen (Literalen), den Zeichen (Symbolen) der definierten Sprache,
- einer endlichen Menge V von Nonterminalsymbolen (Variablen) (mit $V \cap \Sigma = \varnothing$) zur Definition der grammatikalischen Struktur,
- einer endlichen Menge $P \subseteq V \times (V \cup \Sigma)^*$ von Produktionsregeln mit
  - einem Nonterminalsymbol als linker Seite und einem
  - String aus Terminal- und Nichtterminalsymbolen auf der rechten Seite,
- einem ausgezeichneten Nonterminalsymbol $S \in V$ als Startsymbol.

Die von G erzeugte Sprache $L(G) \subseteq \Sigma^*$ ist die Menge aller aus Terminalsymbolen gebildeten Wörter (Sätze) $w \in \Sigma^*$, so daß es eine endliche Folge von Phrasen x0, ..., xn gibt mit

xi $\in$ (V$\cup\Sigma$)* für alle i, x0=S, xn=w, so daß xi aus x(i-1) durch Anwendung einer Produktionsregel r aus P entsteht (wobei in x(i-1) ein Vorkommen der linken Seite von r durch die rechte Seite von r ersetzt werden).

Einfaches Beispiel:
V={Address, Name, Firstname, Lastname, Street, City, Country, String, Letter, Number, Digit},
Σ={a, b, c, ..., 0, 1, …}, S=Address,
P={Address $\rightarrow$ Name Street City, Address $\rightarrow$ Name Street City Country,
Name $\rightarrow$ Firstname Lastname, Name $\rightarrow$ Letter Lastname,
Firstname $\rightarrow$ String, Lastname $\rightarrow$ String,
Street $\rightarrow$ String Number, Country $\rightarrow$ String,
String $\rightarrow$ Letter, String $\rightarrow$ Letter String, Number $\rightarrow$ Digit Number, Number $\rightarrow$ ε,
Letter $\rightarrow$ a, Letter $\rightarrow$ b, …, Digit $\rightarrow$ 0, Digit $\rightarrow$ 1, …}

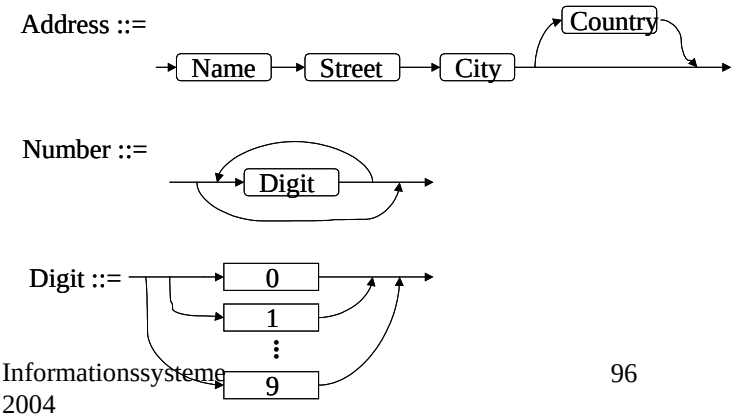
Eine kompaktere "Makroschreibweise" für kontextfreie Grammatiken (insbesondere für rekursive Produktionsregeln) ist die **EBNF (Extended Backus-Naur-Form)** in Verbindung mit notationellen Konventionen:

Nichtterminalsymbole werden klein, Terminalsymbole groß geschrieben (und können ggf. Sonderzeichen beinhalten).
Als Metazeichen für die Regeln selbst dienen
::= zur Trennung von linker und rechter Seite
| zur Trennung von Alternativen für die rechte Seite einer (Gruppe von) Regel(n)
[ ] für optionale Phrasen (die also keinmal oder einmal auf der rechten Seite stehen dürfen)
{ ...} für wiederholbare Phrasen (die keinmal, einmal oder beliebig oft vorkommen dürfen)
{ } zur Klammerung, um die Struktur eindeutig zu machen.
Semantische Optionen, die standardmäßig gesetzte sind (Default-Werte), auch wenn sie syntaktisch gar nicht erzeugt werden, werden unterstrichen.

Beispiel:
Address ::= Name Street City [Country], String ::= Letter {Letter …},
Number ::= {Digit …}, Digit ::= 0|1|2|3|4|5|6|7|8|9, usw.

EBNF-Regeln werden häufig auch in Form von **Syntaxdiagrammen** visualisiert. Dabei werden Nichtterminalsymbole durch Ovale und Terminalsymbole durch Rechtecke dargestellt. Die Nichtterminal- und Terminalsymbole der rechten Seite einer (Gruppe von) Regel(n) sind so durch gerichtete Kanten verbunden, daß jede aus der rechten Seite konstruierbare Ableitung einem Pfad dieses Minigraphen von einer Quelle zu einer Senke entspricht. Rekursive Regeln bzw. wiederholbare Phrasen führen also auf Schleifen in diesem Minigraphen.

Beispiel:



"Grobsyntax" der SQL-Anweisung CREATE TABLE (in EBNF)
(unter Vernachlässigung relativ unwichtiger Features)

```
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 Kunden (  KNr INTEGER CHECK (KNr > 0) PRIMARY KEY,
   Name VARCHAR(30), Stadt VARCHAR(30),
   Saldo FLOAT,
   Rabatt FLOAT CHECK (Rabatt >= 0.0) )
CREATE TABLE Produkte (PNr INTEGER CHECK (PNr > 0) PRIMARY KEY,
   Bez VARCHAR(30) NOT NULL UNIQUE, Gewicht FLOAT CHECK (Gewicht > 0.0),
   Preis FLOAT CHECK (Preis > 0.0),
   Lagerort VARCHAR(30), Vorrat INTEGER CHECK (Vorrat >= 0) )
 CREATE TABLE Bestellungen (
   BestNr INTEGER CHECK (BestNr > 0) PRIMARY KEY,
   Monat INTEGER NOT NULL CHECK (Monat BETWEEN 1 AND 12),
   Tag INTEGER NOT NULL CHECK (Tag BETWEEN 1 AND 31),
   /*** oder besser: Datum DATE, ***/
   KNr INTEGER CHECK(KNr > 0), PNr INTEGER CHECK (PNr > 0),
   Menge INTEGER CHECK (Menge > 0),
   Summe FLOAT, Status VARCHAR(9) CHECK (Status IN ('neu', 'geliefert', 'bezahlt')),
   FOREIGN KEY (PNr) REFERENCES Produkte (PNr),
   FOREIGN KEY (KNr) REFERENCES Kunden (KNr),
   UNIQUE (Monat, Tag, PNr, KNr) )
```

Semantik von CREATE TABLE:
Es wird ein Relationenschema definiert (ggf. mit zusätzlichen Integritätsbedingungen).

Unterschiede zum "reinen" Relationenmodell:

- Die Festlegung eines Primärschlüssels wird nicht unbedingt 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.
- SQL unterstützt eine Vielfalt elementarer Datentypen wie z.B. DATE, Varianten von Strings (VARCHAR etc.) Zahlen (NUMBER etc.) und Binärobjekte (RAW, BLOB, etc.)

Zur Semantik der gemäß column_constraint und table_constraint spezifizierbaren Integritätsbedingungen siehe Kapitel 7.

7.2 Einfache Anfragen (Queries)

"Grobsyntax" von SQL-Anfragen:

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

Ein erster 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 × S × ... × T × ...)
und im Tupel-Relationenkalkül
{x.A, y.B, ... | x ∈ R ∧ y ∈ S ∧ ... ∧ ∃ z: z ∈ T ... ∧ F(x, y, ..., z, ...)}

Achtung:

- Alle Joinbedingungen müssen in SQL explizit in der WHERE-Klausel (oder in SQL-99 alternativ in der FROM-Klausel) angegeben werden.
- Standardmäßig sind Anfrageresultate Multirelationen. Es erfolgt also keine Duplikateliminierung; diese kann mit DISTINCT explizit spezifiziert werden.
- Bei der Vereinigung (UNION), dem Durchschnitt (INTERSECT) und der Differenz (EXCEPT) von Teilergebnissen gibt es zwei Varianten: für Multimengen (mit ALL) und für Mengen.

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
   WHEREMonat < 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
   WHEREStadt='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
   oder:
   SELECT Menge*Preis*(1.0-Rabatt) AS Rechnungssumme FROM ... WHERE ...

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

Analog mit LEFT OUTER JOIN und RIGHT OUTER JOIN.

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 (oder gar keines) 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 ) )
```

7.3 Semantik einfacher SQL-Anfragen

Abbildung von SQL auf den Tupelrelationenkalkül (TRK)

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

Beispiel: SELECT A FROM REL WHERE EXISTS (SELECT B FROM REL WHERE B>0)
würde ggf. umbenannt in
SELECT R1.A FROM REL R1 WHERE EXISTS (SELECT R2.B FROM REL R2 WHERE R2.B>0)

Definition einer Abbildungsfunktion
sql2trc: sql query → trc query
von syntaktischen Konstruktionen der Form select_block auf sichere TRK-Anfragen
unter Verwendung der Funktion
sql2trc': sql where clause → trc formula
von syntaktischen Konstruktionen der Form search_condition auf TRK-Formeln.

sql2trc [SELECT A1, A2, ... FROM REL1 R1, REL2 R2, ..., RELm Rm,
TAB1 T1, ..., TABk Tk WHERE F]
(so daß A1, A2, ..., An zu REL1, REL2, …, RELm gehören, nicht aber zu TAB1, …, TABk und F über REL1, ..., RELm, TAB1, …, TABk definiert ist)
= $\{ri_1.A1, ri_2.A2, ... \mid r1 \in REL1 \wedge r2 \in REL2 \wedge \ ... \wedge rm \in RELm \wedge$
$\exists t1 ... \exists tk (t1 \in TAB1 \wedge ... \wedge tk \in TABk \wedge sql2trc'[F]) \}$

sql2trc [select-block1 UNION select-block2]
(mit select-block1: SELECT A1, A2, ... FROM REL1 R1, REL2 R2, ..., RELm Rm,
TAB1 T1, ..., TABk Tk WHERE F
und select-block2: SELECT B1, B2, ... FROM SET1 S1, SET2 S2, ..., SETm' Sm',
PAR1 P1, ..., PARk' Pk' WHERE G)
= $\{u1, u2, ... \mid (\exists r1 ... \exists rm (u1 = r1.A1 \wedge \ u2 = r2.A2 \wedge \ldots \wedge$
$r1 \in REL1 \wedge r2 \in REL2 \wedge \ ... \wedge rm \in RELm \wedge$
$\exists t1 ... \exists tk \ (t1 \in TAB1 \wedge ... \wedge tk \in TABk \wedge sql2trc'[F] \ ) \ )$
$\vee \quad (\exists s1 ... \exists sm' (u1 = s1.B1 \wedge \ u2 = s2.B2 \wedge \ldots \wedge$
$s1 \in SET1 \wedge s2 \in SET2 \wedge \ ... \wedge sm' \in SETm' \wedge$
$\exists p1 ... \exists pk' (p1 \in PAR1 \wedge ... \wedge pk' \in PARk' \wedge sql2trc'[G] \ ) \ ) \}$

sql2trc [select-block1 INTERSECT select-block2] und
sql2trc [select-block1 EXCEPT select-block2]:
analog

sql2trc' [Ri.Aj θ Tk.Bl] = ri.Aj θ tk.Bl
sql2trc' [Ri.Aj θ c] (mit einer Konstanten c) = ri.Aj θ c
sql2trc' [F AND G] = sql2trc'[F] ∧ sql2trc'[G]
sql2trc' [F OR G] = sql2trc'[F] ∨ sql2trc'[G]
sql2trc' [NOT F] = ¬ sql2trc'[F]

sql2trc' [Ri.Aj IN subquery]
(so daß subquery die Form SELECT Qk.C
FROM QUELL1 Q1, ..., QUELLm' Qm' WHERE H hat)
$= \exists q1 \ldots \exists qm' (qk.C = ri.Aj \wedge q1 \in QUELL1 \wedge \ldots qm' \in QUELLm' \wedge sql2trc'[H])$

sql2trc' [Ri.Aj θANY subquery]
(so daß subquery die Form SELECT Qk.C
FROM QUELL1 Q1, ..., QUELLm' Qm' WHERE H hat)
$= \exists qk (qk \in QUELLk \wedge (\exists q1 \ldots \exists q(k-1) \ \exists q(k+1) \ldots \exists qm'$
$(ri.Aj \ \theta \ qk.C \ \wedge q1 \in QUELL1 \wedge \ldots \wedge q(k-1) \in QUELL(k-1) \wedge$
$q(k+1) \in QUELL(k+1) \wedge \ldots \wedge qm' \in QUELLm' \wedge \ sql2trc'[H])))$

sql2trc' [Ri.Aj θALL subquery]
(so daß subquery die Form SELECT Qk.C
FROM QUELL1 Q1, ..., QUELLm' Qm' WHERE H hat)
$= \forall qk ((qk \in QUELLk \wedge \ (\exists q1 \ldots \exists q(k-1) \ \exists q(k+1) \ldots \exists qm'$
$(q1 \in QUELL1 \wedge \ldots \wedge q(k-1) \in QUELL(k-1) \wedge$
$q(k+1) \in QUELL(k+1) \wedge \ldots \wedge qm' \in QUELLm' \wedge \ sql2trc'[H])))$
$\Rightarrow (ri.Aj \ \theta \ qk.C))$

sql2trc' [EXISTS subquery]
(so daß subquery die Form SELECT C1, C2, ...
FROM QUELL1 Q1, ..., QUELLm' Qm' WHERE H hat)
$= \exists q1 \ldots \exists qm' \ (q1 \in QUELL1 \wedge \ldots \wedge qm' \in QUELLm' \wedge \ sql2trc'[H])$

Beispiel:

query = SELECT K.KNr, K.Name FROM Kunden K
WHERE K.Ort='Saarbrücken'
AND NOT EXISTS (SELECT P.PNr FROM Produkte P, Bestellungen B
WHERE P.Preis > 100 AND P.PNr=B.PNr AND B.KNr=K.KNR)

sql2trc [query] = {k.KNr, k.Name | $k \in$ Kunden $\wedge$ sql2trc'[K.Ort=... AND NOT EXISTS …]}
= {k.KNr, k.Name | $k \in$ Kunden $\wedge$ k.Ort=... $\wedge \neg$ sql2trc'[NOT EXISTS ...]}
= {k.KNr, k.Name | $k \in$ Kunden $\wedge$ k.Ort=... $\wedge$
$\neg (\exists p \exists b (p \in$ Produkte $\wedge b \in$ Bestellungen $\wedge$
sql2trc'[P.Preis>... AND ... AND ...])) }
= {k.KNr, k.Name | $k \in$ Kunden $\wedge$ k.Ort=... $\wedge$
$\neg (\exists p \exists b (p \in$ Produkte $\wedge b \in$ Bestellungen $\wedge$
p.Preis>... $\wedge$ p.PNr=b.PNr $\wedge$ b.KNr=k.KNr)) }

Der Kern der einfachen SQL-Anfragen ist also komplett in den sicheren TRK übersetzbar und hat damit ein präzise definierte Semantik. Umgekehrt sind auch alle möglichen sicheren TRK- (oder RA-) Anfragen in SQL ausdrückbar. Es gilt also:

Satz:

SQL ist relational vollständig.

Abbildung von SQL auf die Relationenalgebra (RA)

unter Vernachlässigung von Multimengen, Nullwerten u.ä.
und der Annahme der eindeutigen Benennung von Tupelvariablen und eindeutigen Zuordnung von Attributen zu Tupelvariablen (wie bei der Abbildung auf den sicheren TRK)

Definition einer Abbildungsfunktion
sql2ra: sql query → ra query
von syntaktischen Konstruktionen der Form select_block auf RA-Anfragen
unter Verwendung der Funktion
sql2ra': sql where clause × ra query → ra query
von syntaktischen Konstruktionen der Form search_condition auf RA-Ausdrücke
sowie der Hilfsfunktion
sql2ra-: sql where clause × ra query → ra query
mit sql2ra- [F](E) = E - π[sch(E)] (sql2ra'[F](E))

sql2ra [SELECT A1, A2, ... FROM REL1 R1, REL2 R2, ..., RELm Rm,
TAB1 T1, ..., TABk Tk WHERE F]
(so daß A1, A2, ..., An zu REL1, REL2, …, RELm gehören, nicht aber zu TAB1, …, TABk und F über REL1, ..., RELm, TAB1, …, TABk definiert ist)
= R1 := REL1; …; Rm := RELm; T1 := TAB1; ...; Tk := TABk;
π[Ri_1.A1, Ri_2.A2, ...] (sql2ra'[F](R1 × … × Rm × T1 × … × Tk))

sql2ra [select-block1 UNION select-block2]
(mit select-block1: SELECT A1, A2, ... FROM REL1 R1, REL2 R2, ..., RELm Rm,
TAB1 T1, ..., TABk Tk WHERE F
und select-block2: SELECT B1, B2, ... FROM SET1 S1, SET2 S2, ..., SETm' Sm',
PAR1 P1, ..., PARk' Pk' WHERE G)
= sql2ra[select-block1] ∪ sql2ra[select-block2]
(mit ggf. notwendigen Umbenennungen von Attributen)

sql2ra [select-block1 INTERSECT select-block2] und
sql2ra [select-block1 EXCEPT select-block2]:
analog

sql2ra' [Ri.Aj θ Tk.Bl] (E) = σ[Ri.Aj θ Tk.Bl](E)
sql2ra' [Ri.Aj θ c] (E) (mit einer Konstanten c) = σ[Ri.Aj θ c](E)
sql2ra' [F AND G] (E) = sql2ra'[F](E) ∩ sql2tra'[G](E)
sql2ra' [F OR G] (E) = sql2ra'[F](E) ∪ sql2ra'[G](E)
sql2ra' [NOT F] (E) = sql2ra-[F](E) = E - π[sch(E)] (sql2ra'[F](E))

sql2ra' [Ri.Aj IN subquery](E)
(so daß subquery die Form SELECT Qk.C
FROM QUELL1 Q1, ..., QUELLm' Qm' WHERE H hat)
= Q1 := QUELL1; … Qm' := QUELLm';
π[sch(E)] (sql2ra'[H] (σ[Ri.Aj = Qk.C] (E × Q1 × … × Qm')))

sql2ra' [Ri.Aj θANY subquery](E)
(so daß subquery die Form SELECT Qk.C
FROM QUELL1 Q1, ..., QUELLm' Qm' WHERE H hat)
= Q1 := QUELL1; … Qm' := QUELLm';
π[sch(E)] (sql2ra'[H] (σ[Ri.Aj θ Qk.C] (E × Q1 × … × Qm')))

sql2ra' [Ri.Aj θALL subquery] (E)
(so daß subquery die Form SELECT Qk.C
FROM QUELL1 Q1, ..., QUELLm' Qm' WHERE H hat)
= sql2ra- [Ri.Aj θ'ANY subquery](E)
mit θ' gleich ≠ für θ gleich =, = für ≠, < für ≥, > für ≤, usw.
= E - π[sch(E)] (sql2ra'[H] (σ[Ri.Aj θ' Qk.C] (E × Q1 × … × Qm')))

sql2ra' [EXISTS subquery] (E)
(so daß subquery die Form SELECT C1, C2, ...
FROM QUELL1 Q1, ..., QUELLm' Qm' WHERE H hat)
= Q1 := QUELL1; … Qm' := QUELLm';
π[sch(E)] (sql2ra'[H] (E × Q1 × … × Qm'))

Beispiel:

query = SELECT K.KNr, K.Name FROM Kunden K
WHERE K.Ort='Saarbrücken'
AND NOT EXISTS (SELECT P.PNr FROM Produkte P, Bestellungen B
WHERE P.Preis > 100 AND P.PNr=B.PNr AND B.KNr=K.KNR)

sql2ra [query] = K := Kunden; B := Bestellungen; P := Produkte;
π[K.KNr, K.Name] (sql2ra'[K.Ort=… AND NOT EXISTS …] (K))
= … π[K.KNr, K.Name] (σ[Ort=…](K) ∩ sql2ra'[NOT EXISTS …](K))
= … π[K.KNr, K.Name] (σ[Ort=…](K) ∩
sql2ra- [EXISTS ...](K))
= … π[K.KNr, K.Name] (σ[Ort=…](K) ∩
(K - π[sch(K)](sql2ra' [EXISTS ...](K))))
= ... π[K.KNr, K.Name] (σ[Ort=…](K) ∩
(K - π[sch(K)](sql2ra'[P.Preis...AND…AND…](K × P × B)))
= ... π[K.KNr, K.Name] (σ[Ort=…](K) ∩
(K - π[sch(K)](σ[P.Preis>100](K × P × B) ∩
σ[P.PNr=B.PNr](K × P × B) ∩
σ[P.PNr=B.PNr](K × P × B))))

Für den Umgang mit Multimengen könnte man eine Übersetzung in die erweiterte Relationenalgebra in analoger Weise angeben. Beispielsweise würde sql2ra [SELECT DISTINCT A FROM ... WHERE ...] übersetzt werden in χ (sql2ra [SELECT A FROM ... WHERE ...).

7.4 Erweiterte SQL-Anfragen

Aggregationsfunktionen

Zweck: Abbildung einer Menge oder Multimenge von skalaren 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 skalaren 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)
SELECT TO_CHAR(SYSDATE, 'DY DD MONTH YYYY, HH24:MI:SS')
zur Konvertierung des aktuellen Tagesdatums (Datentyp DATE) in einen String

Ideal: Erweiterbarkeit des Datenbanksystems (siehe Kapitel 8)

Oracle-spezifische Erweiterung zur Textsuche (Oracle interMedia)

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, SCORE(1) FROM Produkte
   WHERE CONTAINS (Beschreibung, 'Internet', 1) > 0 bzw. >= 5
   ORDER BY SCORE(1) DESC

   liefert alle Produkte, deren Beschreibung das Wort 'Internet' enthält
   bzw. mindestens fünfmal enthält, in einer nach Relevanz absteigend sortierten Rangliste

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, SCORE(1) FROM Produkte
   WHERE CONTAINS (Beschreibung, '$Query', 1) > 0 ORDER BY SCORE(1) DESC

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

4) SELECT PNr, Bez, SCORE(1) FROM Produkte
   WHERE CONTAINS (Beschreibung, '?Oracle', 1) > 0 ORDER BY SCORE(1) DESC

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

5) 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 (z.B. 'Awewreckle'), und zwar nur die (maximal) zehn ähnlichsten Treffer

6) SELECT PNr, Bez, SCORE(1)*SCORE(2) AS TOTALSCORE FROM Produkte
   WHERE CONTAINS (Beschreibung, 'SYN(database system)', 1) > 0
   AND CONTAINS (Beschreibung, 'NT(search)', 2) > 0 ORDER BY SCORE(1)*SCORE(2) DESC

   mit Nachschlagen in einem (ggf. benutzerspezifischen) Thesaurus (Wörterbuch, Ontologie) zur Berücksichtigung von Synonymen (SYN) und Unterbegriffen (NT)

Aggregation mit Gruppenbildung

Zweck:
Eine Tupelmenge wird aufgrund der Werte eines Attributs oder einer Attributkombination oder eines Ausdrucks in "Gruppen" (Äquivalenzklassen) 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

Achtung:
In der SELECT-Klausel (d.h. der Projektionsliste) dürfen nur Resultate von Aggregationsfunktionen stehen oder Attribute, die auch Gruppierungsattribute sind (also in der GROUP-Klausel vorkommen).

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 bzw. ORDER BY MittlRabatt 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
mit Aggregationsfunktion f und Attributmengen A, A' mit $A' \subseteq A$
(falls $A' \subseteq A$ nicht gilt, hat die Anfrage keine wohldefinierte Semantik und wird vom SQL-Compiler als fehlerhaft abgewiesen).
Die Semantik dieser Anfrage ist durch Übersetzung in die erweiterte Relationenalgebra wie folgt definiert:
sql2ra [SELECT A', f(B) FROM … WHERE … GROUP BY A]
= R(A, F) := γ_+[A,B,f] (sql2ra[SELECT A,B FROM … WHERE …]); π_+[A', F] (R)
wobei sql2ra auf Multirelationen zu erweitern ist.

Gegeben sei eine Anfrage der Form
SELECT A', f(B)
FROM ...
WHERE ...
GROUP BY A
HAVING cond(A, g(C))
mit Aggregationsfunktionen f, g und einer Suchbedingung cond über A und g(C) (die auch wieder Subqueries beinhalten darf, vom äußeren Block aber nur auf Attribute aus A oder Aggregationsfunktionen g(C) Bezug nehmen darf). Der Einfachheit halber sei nur der Fall ohne DISTINCT betrachtet.

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

R (A,F) := $\emptyset$
for each x $\in \pi$ [A](Q) do
$G_X := \pi_+$[A,B,C] (σ_+[A=x] (Q))
if sql2ra'[cond(A,g(C))](G_X) $\neq \emptyset$ then
y := f (π_+[B](G_X))
R := R $\cup_+$ {(x,y)}
fi
od;
π_+[A', F] (R)

Rekursive Anfragen in SQL-99 bzw. Oracle:

Der neuere Standard SQL-99 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, F. 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
AND PRIOR Ankunftszeit < Abflugzeit - 0.05
```

7.5 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 ggf. spezifizierten Default-Wert oder den Nullwert gesetzt.

Transaktionen

Es ist häufig notwendig, mehrere SQL-Anweisungen zu einer atomaren Einheit zu klammern, die nur als Ganzes einen konsistenten Datenbankzustand in einen anderen konsistenten Zustand überführen (insbesondere ist dies für logisch gekoppelte Änderungen mehrerer Tabellen unumgänglich). Diese Klammerung erfolgt durch Abschluß einer Folge von SQL-Anweisungen mit der Anweisung COMMIT WORK bzw. ROLLBACK WORK, wobei letzteres alle Änderung der Transaktionen rückgängig macht. Mit dem Abschluß einer Transaktion (und initial beim Verbindungsaufbau mit dem Datenbanksystem) wird implizit eine neue Transaktion geöffnet. In manchen Programmierumgebungen gibt es einen Autocommit-Modus, bei dem automatisch jede einzelne SQL-Anweisung eine Transaktion bildet, indem implizit nach jeder Anweisung ein "Commit Work" durchgeführt wird.

Beispiel einer Transaktion: Bearbeitung einer neuen Lieferung

```
UPDATE Produkte SET Vorrat = Vorrat - 10 WHERE PNr = 4711;
UPDATE Bestellungen SET Status = 'geliefert' WHERE BestNr = 333444555;
COMMIT WORK;
```

Transaktionen werden ausführlich in späteren Kapiteln behandelt.

Ergänzende Literatur zu Kapitel 7:

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, Addison-Wesley, 1997

P. Gulutzan, T. Pelzer, SQL-99 Complete, Really, R&D Books, 1999

C. Türker, SQL:1999 & SQL:2003, dpunkt-Verlag, 2003

Oracle9i SQL Reference,
http://tahiti.oracle.com/pls/db92/db92.show_toc?partno=a96540

Oracle9i SQL*Plus User's Guide and Reference
http://tahiti.oracle.com/pls/db92/db92.show_toc?partno=a90842

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