SQL

SQL-Historie

- Seit 1974 viele Sprachentwürfe:
  - z.B. SQUARE, SEQUEL, QUEL
- Entwicklung einer vereinheitlichten DB-Sprache für alle Aufgaben der DB-Verwaltung
- Zielgruppe: Nicht-Programmierer
- Auswahlvermögen äquivalent dem Relationenkalkül und der Relationenalgebra (relational vollständig)
- Leichter Zugang durch verschiedene “Sprachebenen“ anwachsender Komplexität
  - Einfache Anfragemöglichkeiten für den gelegentlichen Benutzer
  - Mächtige Sprachkonstrukte für den besser ausgebildeten Benutzer
  - Spezielle Sprachkonstrukte für den DB-Administrator
- SQL wurde de facto Standard in der relationalen Welt (ANSI-Standard 1986)
- SQL-89 mit Erweiterungen für Integritätskontrolle
- Weiterentwicklung des Standards:
  - SQL2 (SQL-92) mit 3 Stufen: Entry SQL, Intermediate SQL, Full SQL
  - SQL-99: neue objektrelationale Erweiterungen

SQL: Grundkonzepte

- strukturierte Sprache, die auf englischen Schlüsselwörtern basiert
- Allgemeines Format:
  <Spezifikation der Operation>
  <Liste der referenzierten Tabellen>
  [WHERE Boolescher Prädikatausdruck]
- Mengenorientierte Anfragen deskriptiver Art (Retrieval)
- Möglichkeiten der Datenmanipulation (DML)
  - Insert
  - Update
  - Delete
- Möglichkeiten der Datendefinition (DDL)
  - Basisrelationen
  - Sichtkonzept
- Kopplung mit einer Wirtssprache
  - Cursor-Konzept
  - Dynamisches SQL
- Möglichkeiten der Datenkontrolle
  - Integritätskontrolle
  - Zugriffskontrolle
  - Transaktionskonzept

Beispiel-Relationen

- Beispiel-Relationen für nachfolgende Queries:

  Sailors (sid: integer, sname: string, rating: integer, age: real)

  Boats (bid: integer, bname: string, color: string)

  Reserves (sid: integer, bid: integer, day: date)

  – 2 Instanzen von *Sailor: S1* und *S2*

  – 1 Instanz von *Reserves: R*

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Datentypen in SQL

- String-Datentypen
  - CHARACTER[(n)] (Abk. CHAR)
  - VARCHAR(n) (Abk. VARCHAR)
  - BIT[(n)]
  - BIT VARYING[(n)]
- Numerische Datentypen
  - DECIMAL(p,q) (Abk. DEC)
  - INTEGER (Abk. INT)
  - SMALLINT halbes Wort
  - FLOAT(p) p=Präzision
  - REAL
  - DOUBLE PRECISION
- Besondere Typen
  - DATE Datum (Format wählbar)
  - TIME Datum und Uhrzeit
  - TIMESTAMP Zeitstempel mit Mikrosek.-Präzision
  - INTERVAL Datums- und Zeitintervalle
- Besonderheiten der Typsysteme in den einzelnen DBMS beachten (z.B. VARCHAR2 oder NUMBER = 40stellige Nummer in Oracle 8)

Erzeugen von Relationen

- CREATE TABLE

  Wird für Definition von Basis-Relationen benutzt (Basis-Relation=gespeicherte Relation, im Gegensatz zu Sichten)

CREATE TABLE *base-table*
(*column-definition* [, *column-definition*] ...
[, *primary-key-definition*]
[, *foreign-key-definition* [, *foreign-key-definition*] ...]);

Eine *column-definition* hat folgende Form:

column data-type [NOT NULL]

Beispiel:

```
CREATE TABLE Sailors
(sid      INTEGER          PRIMARY KEY,
 sname    VARCHAR(20) NOT NULL,
 rating   INTEGER;
 age      REAL);
```

Fremdschlüsselbeziehungen

```
CREATE TABLE Reserves
( sid      INTEGER ,
  bid      INTEGER,
  day      DATE,
  PRIMARY KEY (sid,bid),
  FOREIGN KEY (sid) REFERENCES Sailors,
  FOREIGN KEY (bid) REFERENCES Boats(bid));
```

- Integritätsbedingungen, die mehrere Attribute (einer oder mehrerer) Relationen betreffen, können als eigene *table constraints* definiert werden, siehe die PRIMARY KEY-Definition
- Fremdschlüsselbeziehungen können auf Spalten- oder Tabellenebene definiert werden
- FOREIGN KEY-Klausel muß die referenzierte Relation definieren (REFERENCES *base-table*)
- Optional ist die Angabe der Spalte(n) in der referenzierten Tabelle (per Default wird referentielle Integrität mit dem PRIMARY KEY überprüft)
- Weitere Definition von Integritätsbedingungen in SQL möglich (CHECK-Klausel)

Dynamische Änderung einer Relation

- Bei Relationen können dynamisch (während ihrer Lebenszeit) Schemaänderungen durchgeführt werden
  - Hinzufügen, Ändern und Löschen von Attributen
  - Hinzufügen und Löschen von Constraints
- Beispiel: Hinzufügen einer Spalte

```
ALTER TABLE base-table
ADD column-definition
```

```
ALTER TABLE Sailors
ADD phone  VARCHAR(20)
```

- Löschen von Spalten mittels DROP column

  Mögliche Probleme:

  - Was passiert mit Klauseln (z.B. Sichten oder Integritäts-bedingungen), die die betroffene Spalte referenzieren?
  - Was passiert mit Tupeln mit einem Wert <> NULL in dieser Spalte?

```
ALTER TABLE Sailors
DROP phone
```

Löschen von Objekten

- Falls Objekte (Relationen, Sichten) nicht mehr benötigt werden, können sie durch die DROP-Anweisung aus dem System entfernt werden
- Abhängige Objekte (z.B. darauf definierte Sichten oder Index-Strukturen) können wahlweise auch gelöscht werden (CASCADE-Option)
- Löschen von Relationen

DROP TABLE *base-table*

SQL-Query

```
SELECT    [DISTINCT] target-list
FROM      relation-list
WHERE     qualification
```

- *relation-list* Eine Liste von Relationen-Namen, eventuell mit einer Wertbereichs-Variablen (Alias) nach jedem Namen *(range-variable)*
- *target-list* Eine Liste von Attributen von Relationen aus der *relation-list.* Wahlweise auch * möglich: alle Attribute der Relationen aus *relation-list*.
- *qualification* Logische Prädikate (Attr *op* const oder Attr1 *op* Attr2, mit *op* aus (<, >, =, <=, >=, <>)
  kombiniert durch AND, OR und NOT.
- DISTINCT ist ein optionales Schlüsselwort, um Duplikate im Resultat zu unterdrücken. Per Default werden Duplikate nicht eliminiert!

Auswertungsstrategie für eine Anfrage

- Semantik einer SQL-Query wird bestimmt durch folgende Auswertungsstrategie:
  - Berechne das Kreuzprodukt von *relation-list*.
  - Beseitige Ergebnistupel, die nicht die WHERE-Bedingung erfüllen *(qualification)*: Selektion
  - Lösche Attribute, die nicht in der Auswahlliste (*target-list)* stehen: Projektion.
  - Wenn DISTINCT angegeben, Duplikate im Ergebnis eliminieren.
- Diese Strategie ist sehr ineffizient, um eine Query zu berechnen! Ein Optimierer kann effizientere Strategien finden, um die gleichen Antworten zu berechnen.

```
SELECT   A1, A2, ..., An
FROM     R1, R2, ..., Rm
WHERE    P
```

entspricht dem Ausdruck:

$$\pi_{A1,A2,\ldots,An} S = (\sigma_P (R1 \times R2 \times \ldots \times Rm))$$

Beispiel-Auswertung

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid AND R.bid=103
```

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

Tupel-Variablen

- Auch: Alias oder Wertbereichs-Variablen (*Range Variable*)
- Nur dann benötigt, wenn die gleiche Relation mehrfach in der FROM-Klausel erscheint (z.B. bei einem Self-Join)
- Beide Schreibweisen für gegebene Query erlaubt, also:

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid AND bid=103
```

ODER

Es ist jedoch guter Stil, immer Tupel-Variablen zu verwenden

```
SELECT  sname
FROM    Sailors, Reserves
WHERE   Sailors.sid=Reserves.sid AND bid=103
```

Einfache SQL-Queries

Finde Segler, die mindestens ein Boot reserviert haben.

```
SELECT  R.sid
FROM  Sailors S, Reserves R
WHERE  S.sid=R.sid
```

Einfügen von DISTINCT würde Tupel eliminieren, falls Segler mehrere Boote reserviert haben.

Finde die Namen der Segler, die mindestens ein rotes Boot reserviert haben.

```
SELECT  S.sname
FROM  Sailors S, Reserves R, Boats B
WHERE  S.sid=R.sid AND B.bid=R.bid AND B.color='red'
```

Einfügen von DISTINCT: - würde Tupel eliminieren, falls Segler mehrere rote Boote reserviert haben.

- Da *sname* nicht eindeutig ist, würden gleichnamige Segler nicht als solche angezeigt

Ausdrücke und Zeichenketten

```
SELECT  S.age, age1=S.age-5, 2*S.age AS age2
FROM  Sailors S
WHERE  S.sname LIKE 'B_%B'
```

- Illustriert den Gebrauch von arithmetischen Ausdrücken und Mustervergleichen: *Finde Tripel (Alter des Seglers und zwei Attribute, definiert durch Ausdrücke) für die Segler, deren Namen mit einem 'B' beginnen und enden und die mindestens drei Zeichen enthalten.*
- AS und = sind zwei Varianten, um Attribute im Resultat zu benennen
- LIKE wird verwendet für String-Vergleiche (Matching). `_' repräsentiert ein beliebiges Zeichen und '%' steht für 0 oder mehr beliebige Zeichen.

Mengenoperatoren

Finde die *Id's* der Segler, die ein rotes oder ein grünes Boot reserviert haben.

- UNION: für die Berechnung der Vereinigung zweier vereinigungs-verträglicher Mengen von Tupeln (welche selbst Ergebnis von SQL-Anfragen sind)
- vereinigungsverträglich: gleiche Anzahl von Spalten, die jeweils im Typ übereinstimmen

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND
R.bid=B.bid
  AND (B.color='red' OR
B.color='green')
```

Alternative Lösung:

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND
R.bid=B.bid
          AND B.color='red'
UNION
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND
R.bid=B.bid
          AND B.color='green'
```

Mengenoperatoren (Forts.)

Finde die *Id's* der Segler, die ein rotes und ein grünes Boot reserviert haben.

- INTERSECT: Für die Berechnung des Durchschnitts von zwei beliebigen vereinigungsverträglichen Tupelmengen
- Definiert im SQL/92 Standard, aber nicht von allen Systemen unterstützt
- Symmetrie mit UNION-Queries

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color='red'
INTERSECT
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color='green'
```

Schlüssel-Attribut!

```
SELECT  S.sid
FROM  Sailors S, Boats B1, Reserves R1, Boats B2, Reserves R2
WHERE  S.sid=R1.sid AND R1.bid=B1.bid AND  S.sid=R2.sid AND
       R2.bid=B2.bid AND (B1.color='red' AND B2.color='green')
```

Mengenoperatoren (Forts.)

Finde die *Id's* der Segler, die ein rotes, aber kein grünes Boot reserviert haben.

- EXCEPT: Für die Berechnung der Differenz von zwei beliebigen vereinigungsverträglichen Tupelmengen

```
SELECT  S.sid
FROM  Sailors S, Reserves R, Boats B
WHERE  S.sid=R.sid AND R.bid=B.bid
       AND  B.color='red'
EXCEPT
SELECT  S2.sid
FROM  Sailors S2, Reserves R2, Boats B2
WHERE  S2.sid=R2.sid AND R2.bid=B2.bid
       AND  B2.color='green'
```

Einfachere Lösung möglich, da *sid* bereits in Relation *Reserves* steht:

```
SELECT  R.sid
FROM  Boats B, Reserves R
WHERE  R.bid=B.bid AND
       B.color='red'
EXCEPT
SELECT  R2.sid
FROM  Boats B2, Reserves R2
WHERE  R2.bid=B2.bid AND
       B2.color='red'
```

Geschachtelte Anfragen (Nested Queries)

Finde die Namen der Segler, die das Boot #103 reserviert haben.

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.sid IN  (SELECT  R.sid
                  FROM  Reserves R
                  WHERE  R.bid=103)
```

- Ein sehr mächtiges Feature von SQL: eine WHERE Klausel kann selbst wiederum eine SQL-Query beinhalten! (gilt ebenso auch für FROM and HAVING Klauseln)
- Um Segler zu finden, die nicht #103 reserviert haben: verwende NOT IN.
- Zum Verständnis der Semantik geschachtelter Anfragen: vergleiche mit der Auswertung ineinander geschachtelter Queries: Für jedes Tupel in *Sailor*, überprüfe, ob es die Selektions-bedingung erfüllt durch Berechnung der Subquery

Geschachtelte Anfragen mit Korrelation

Finde die Namen der Segler, die das Boot #103 reserviert haben.

```
SELECT  S.sname
FROM  Sailors S
WHERE   EXISTS  (SELECT  *
                 FROM  Reserves R
                 WHERE S.sid=R.sid AND
                 R.bid=103)
```

Korrelation

- EXISTS ist ein weiterer Mengenvergleichsoperator, wie IN (Test, ob eine Menge leer ist)
- Alternative Lösung: Ersetze EXISTS durch UNIQUE und * durch *R.bid:* Finde Segler mit höchstens einer Reservierung für Boot #103. (UNIQUE prüft auf Duplikate, ist TRUE, wenn keine vorhanden)
- Wichtig: Ersetze * durch *R.bid*
- Beispiel zeigt, warum i. allg. die Subquery für jedes Tupel aus *Sailors* neu berechnet werden muß

Weitere Mengen-Operatoren

- Bisher: IN, EXISTS und UNIQUE. Können alle negiert werden: NOT IN, NOT EXISTS and NOT UNIQUE.
- Auch verfügbar: *op* ANY, *op* ALL,

  Mit *op* IN $>, <, =, \geq, \leq, \neq$

Finde Segler, deren Rating größer ist als das von irgendeinem Segler namens Horatio:

```
SELECT  *
FROM  Sailors S
WHERE  S.rating >
        ANY  (SELECT  S2.rating
            FROM  Sailors S2
            WHERE S2.sname='Horatio')
```

Finde die Segler mit dem höchsten Rating.

```
SELECT  *
FROM  Sailors S
WHERE  S.rating > =
        ALL  (SELECT  S2.rating
            FROM  Sailors S2)
```

Umschreiben von INTERSECT

Finde die *Id's* der Segler, die ein rotes und ein grünes Boot reserviert haben.

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color='red' AND S.sid
IN
        (SELECT  S2.sid
         FROM  Sailors S2, Boats B2, Reserves R2
         WHERE  S2.sid=R2.sid AND R2.bid=B2.bid
                AND  B2.color='green')
```

- Analog, EXCEPT-Queries umschreiben durch Verwendung von NOT IN.
- Um die Namen (nicht *sid*'s) der Segler zu bestimmen die ein rotes und ein grünes Boot reserviert haben: Ersetze *S.sid* durch *S.sname* in der SELECT Klausel.

Division in SQL*

Finde die Segler, die alle Boote reserviert haben.

```
SELECT  S.sname
FROM  Sailors S
WHERE  NOT EXISTS
        ((SELECT  B.bid FROM  Boats B)
         EXCEPT
          (SELECT  R.bid FROM  Reserves R
           WHERE  R.sid=S.sid))
```

Ohne EXCEPT:

Alle Segler S, für die ...

kein Boot B existiert ohne daß ...

es ein Tupel in *Reserves* gibt, in dem ein B durch S reserviert wird.

```
SELECT  S.sname
FROM  Sailors S
WHERE  NOT EXISTS
        (SELECT  B.bid
         FROM  Boats B
         WHERE  NOT EXISTS
                (SELECT  R.bid
                 FROM  Reserves R
                 WHERE  R.bid=B.bid
                 AND  R.sid=S.sid))
```

Aggregations-Operatoren

Erweiterung der Relationenalgebra durch Standardfunktionen

COUNT (*)	Anzahl Tupel
COUNT ([DISTINCT] A)	Anzahl (verschiedener) Werte in A
SUM ([DISTINCT] A)	Summe (verschiedener) Werte in A
AVG ([DISTINCT] A)	Durchschnitt (versch.) Werte in A
MAX (A)	Maximal-Wert in Spalte A
MIN (A)	Minimal-Wert in Spalte A

Finde die Anzahl aller Segler.

```
SELECT  COUNT (*)
FROM  Sailors S
```

Finde das Durchschnittsalter aller Segler.

```
SELECT  AVG (age)
FROM  Sailors
```

Finde das Durchschnittsalter aller Segler mit einem Rating = 10.

```
SELECT  AVG (S.age)
FROM  Sailors S
WHERE  S.rating=10
```

Ermittlung von Maximalwerten

Finde Name und Alter des ältesten Seglers.

```
SELECT  S.sname, MAX (S.age)
FROM  Sailors S
```

- Diese “Lösung“ ist unzulässig (vgl. Einführung von GROUP BY)

```
SELECT  S.sname, S.age
FROM  Sailors S
WHERE  S.age =
            (SELECT  MAX
(S2.age)
            FROM  Sailors S2)
```

- Die dritte Lösung ist äquivalent zur zweiten Query und ist erlaubt im SQL/92 Standard, aber wird in manchen Systemen nicht unterstützt.

```
SELECT  S.sname, S.age
FROM  Sailors S
WHERE  (SELECT  MAX (S2.age)
            FROM  Sailors S2)
            = S.age
```

Weitere Beispiele

Finde die Anzahl der unterschiedlichen Seglernamen

```
SELECT  COUNT (DISTINCT S.sname)
FROM  Sailors S
```

Finde die Namen der Segler, die älter als der älteste Segler mit dem Rating 10 sind.

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.age > (SELECT  MAX(S2.age)
                FROM  Sailors S2
                WHERE S2.rating=10 )
```

Äquivalente Query mit ALL-Klausel:

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.age > ALL (SELECT  S2.age
                    FROM  Sailors S2
                    WHERE S2.rating=10 )
```

GROUP BY und HAVING

- Bisher haben wir Aggregat-Operatoren auf alle Tupel (die durch WHERE qualifiziert sind) angewandt. Manchmal wollen wir die Tupel gruppieren und diese auf jede Gruppe von Tupeln anwenden.
- Beispiel: *Finde das Alter des jüngsten Seglers für jede Rating-Stufe.*
  - Im allgemeinen kennen wir die Anzahl der Rating-Stufen nicht und auch nicht die Werte dieser Stufen.
  - Angenommen, wir kennen 10 Rating-Stufen von 1 bis 10, so können wir 10 Queries schreiben, die so aussehen:

For i = 1, 2, ... , 10:

```
SELECT  MIN (S.age)
FROM  Sailors S
WHERE  S.rating = i
```

Queries mit GROUP BY und HAVING

```
SELECT      [DISTINCT]  target-list
FROM        relation-list
WHERE       qualification
GROUP BY    grouping-list
HAVING      group-qualification
```

- Die *target-list* enthält (i) Attributnamen (ii) Terme mit Aggregat-Operatoren (z.B. MIN (*S.age*)).
  - Die Liste der Attribute (i) muß eine Teilmenge der Liste in *grouping-list* sein. Jedes Ergebnistupel korrespondiert mit einer *Group.*
  - Eine *Group* ist eine Menge von Tupeln, die die gleichen Werte in den Attributen hat, die in *grouping-list* genannt sind
- Die Ausdrücke in *group-qualification* müssen einen einzelnen Wert (Skalar) pro Group liefern. Durch die HAVING-Klausel wird entschieden, ob ein Ergebnistupel für eine Group generiert wird.
- Wenn GROUP BY fehlt, wird die gesamte Tabelle als eine einzige Gruppe behandelt.

Berechnung von GROUP BY und HAVING

- Auswahl der Tupel durch die WHERE-Klausel

  Das Kreuzprodukt von *relation-list* wird berechnet, Tupel, die die WHERE-Bedingung (*qualification*) nicht erfüllen, werden entfernt, `*überflüssige'* Attribute werden gelöscht.
- Bildung von Gruppen durch die GROUP BY Klausel

  Die verbleibenden Tupel werden in Gruppen partitioniert, bestimmt durch die Werte der Attribute in *grouping-list*.
- Auswahl der Gruppen, die die HAVING-Klausel erfüllen

  Die HAVING-Klausel (*group-qualification*) wird angewandt, um einige Gruppen zu eliminieren. Ausdrücke in *group-qualification* müssen einen skalaren Wert pro Gruppe liefern!
  - Ein Attribut in *group-qualification*, das nicht Argument einer Standardfunktion ist, erscheint auch in der *grouping-list*. (SQL nutzt hier keine Primärschlüssel-Semantik!)
- Ein Antwort-Tupel wird pro Gruppe generiert, die die HAVING-Bedingung erfüllt.

Beispiel

Finde das Alter des jüngsten Seglers mit Alter >= 18, für jede Rating-Stufe mit mindestens zwei solcher Segler

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
71	zorba	10	16.0
64	horatio	7	35.0
29	brutus	1	33.0
58	rusty	10	35.0

```
SELECT S.rating, MIN (S.age)
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT (*) > 1
```

rating	age
1	33.0
7	45.0
7	35.0
8	55.5
10	35.0

rating	
7	35.0

Ergebnis-relation

- Nur S.rating und S.age werden genannt in den SELECT, GROUP BY oder HAVING Klauseln; andere Attribute `unnötig'.
- 2. Spalte des Ergebnisses ist nicht benannt (AS zur Benennung)

Weitere Beispiele

Bestimme die Anzahl der Reservierungen für jedes rote Boot.

```
SELECT  B.bid,  COUNT (*) AS scount
FROM    Boats B, Reserves R
WHERE   R.bid=B.bid AND B.color='red'
GROUP BY  B.bid
```

- Gruppierung über einen Join von zwei Relationen
- Es ist nicht erlaubt die Bedingung *B.color='red'* in der WHERE-Klausel wegzulassen und stattdessen eine HAVING-Klausel hinzuzufügen (nur Spalten aus der GROUP BY Klausel dürfen in HAVING verwendet werden

Weitere Beispiele (Forts.)

Finde das Durchschnittsalter der Segler, die mindestens 18 Jahre alt sind, für jede Rating-Stufe mit mindestens 2 Seglern (beliebigen Alters).

```
SELECT S.rating, AVG (S.age) AS avgage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING 1 < (SELECT COUNT (*)
            FROM Sailors S2
            WHERE S.rating=S2.rating)
```

- HAVING kann auch Subquery enthalten
- Vereinfachung der HAVING-Klausel möglich:

  ... HAVING COUNT(*) > 1
- Falls die Bedingung age >= 18 auch in der Subquery von HAVING getestet wird, kann ein anderes Resultat herauskommen (weil zusätzliche Einschränkung für die Gruppenbildung)

Verschachtelungen

Finde die Ratings, bei denen das Durchschnittsalter der Segler das Minimum über alle Ratings darstellt.

- Standardfunktionen können nicht geschachtelt werden

falsch!

```
SELECT  S.rating
FROM  Sailors S
WHERE  S.age =  (SELECT  MIN (AVG (S2.age))
                 FROM Sailors S2)
```

- Auch FROM-Klausel kann geschachtelt werden (SQL/92-Standard; nicht von allen Systemen unterstützt)

```
SELECT  Temp.rating, Temp.avgage
FROM  (SELECT  S.rating, AVG (S.age) AS avgage
       FROM  Sailors S
       GROUP BY  S.rating) AS Temp
WHERE  Temp.avgage =
                (SELECT  MIN (Temp.avgage)
                 FROM  Temp)
```

Einfügen von Tupeln

```
INSERT INTO table [(attribute-list)]
        { VALUES (value-list) |
          table-expr |
          DEFAULT VALUES }
```

```
INSERT INTO Students (sid, name, login, age, gpa)
VALUES (53688, 'Mike', 'mike@htwk', 22, 1.3)
```

- Falls Spalten nicht explizit genannt werden, wird ihnen ein NULL-Wert (oder falls definiert: ein Default-Wert zugewiesen)
- Falls alle Werte in der richtigen Reihenfolge genannt werden, kann Attributliste weggelassen werden
- Mengenorientiertes Einfügen ist möglich, wenn die einzufügenden Tupel aus einer anderen Relation mit Hilfe einer SELECT-Anweisung ausgewählt werden

```
INSERT INTO YoungStudents
        SELECT * FROM Students
        WHERE age < 20
```

Löschen von Tupeln

DELETE [FROM] *table*
WHERE *qualification*

```
DELETE FROM Students
WHERE sid=53688
```

- Der Aufbau der WHERE-Klausel entspricht dem in der SELECT-Anweisung.
- Eine fehlende WHERE-Klausel wird als TRUE interpretiert (Löschen aller Tupel, erhalten bleibt aber die Definition der Tabelle)
- WHERE-Klausel kann auch geschachtelt sein

```
DELETE FROM Sailors
WHERE rating <
        (SELECT AVG(rating)
        FROM Sailors)
```

```
DELETE FROM Sailors
WHERE sid NOT IN
        (SELECT sid
         FROM Reserves)
```

  - Tupel werden als gelöscht gekennzeichnet und nur am Ende einer Operation wirklich gelöscht - um Anomalien zu vermeiden (!!)

Ändern von Tupeln

UPDATE *table*
SET *update-assignment-list*
WHERE *qualification*

```
UPDATE Students S
SET s.age=s.age+1
```

```
UPDATE Students S
SET      s.gpa = s.gpa-0.1
WHERE  S.gpa >= 3.3
```

- Nicht alle Änderungen können als einfache delete/insert-Paare ausgedrückt werden
- UPDATE erlaubt, einzelne Attributwerte zu ändern, ohne den Rest des Tupels zu beeinflussen
- Der Aufbau der WHERE-Klausel entspricht dem in der SELECT-Anweisung.
- Einschränkung: Innerhalb der WHERE-Klausel in einer Lösch- oder Änderungsanweisung sollte die Zielrelation in einer FROM-Klausel nicht referenziert werden

Semantik von Null-Werten

- UNKNOWN (unbekannt)
  - Wert existiert, ist aber unbekannt
    (Unbekanntes Gehalt eines Angestellten)
  - Wert ist ungültig
    (Offensichtlich falsche Daten, z.B. Alter eines Angestellten 92 Jahre)
  - Wert wurde nicht angegeben
    (Verweigerte Aussage bei Zählungsdaten)
- INAPPLICABLE (nicht anwendbar)
  - Attribut trifft bei diesem Tupel nicht zu
    (Provision bei Angestelltem mit festem Gehalt, Geburten bei männlichen Patienten)
  - Wert existiert nicht
    (Bankverbindung)
  - Wert ist nicht definiert
    (Maximalwert einer leeren Menge)

Null-Werte in SQL

- SQL bietet einen speziellen Wert NULL
- NULL verursacht zusätzliche Probleme
  - Spezielle Operationen notwendig zum Test eines Wertes: IS NULL / IS NOT NULL?
  - Vergleiche mit Nullwerten ergeben immer FALSE, z.B. rating > 8 = FALSE wenn rating=NULL
  - Bedeutung von Klauseln muß sorgfältig definiert werden (z.B. WHERE eliminiert Tupel, die nicht TRUE sind)
  - Bei Aggregationssoperationen (z.B. AVG, MAX, MIN) werden Tupel mit Nullwerten ignoriert
  - AVG, MAX und MIN einer leeren Menge wird als unbekannt definiert, da kein undefiniert in SQL existiert
  - COUNT-Operator zählt Tupel mit Nullwerten mit
  - Nullwerte dürfen nicht bei Schlüsselattributen benutzt werden
  - Nullwerte dürfen nicht in indizierten Attributen erscheinen
- Einführung einer 3wertigen Logik notwendig
  - NOT: not u = u
  - AND: u and u = u, u and t = u, u and f = f
  - OR: u or u = u, u or f = u

Default-Werte in SQL

- Benutzer kann Default-Werte definieren, die durch System verwaltet werden
- Erlaubt automatische Eingabe oft benutzter Werte (z.B. Leipziger Vorwahl, falls nicht anders angegeben)
- Defaults sind auch benutzerdefinierte Nullwerte
- Defaults sind besonders nützlich, wenn ein Attribut als NOT NULL definiert wurde
- Kommandos: CREATE DEFAULT, DROP DEFAULT
- Ergebnis von INSERT je nach Definition eines Attributs

Definition	Kein Eintrag Kein Default	Kein Eintrag Default	Eingabe NULL Kein Default	Eingabe NULL Default
NULL	NULL	DEFAULT	NULL	NULL
NOT NULL	ERROR	DEFAULT	ERROR	ERROR

Sichtkonzept (Views)

- Sicht (View): mit Namen bezeichnet, aus Basisrelation abgeleitete, virtuelle Relation (View-Name wie Tabellen-Name verwendbar)
- Views sind das Ergebnis einer Query, auf dem weitere Operationen durchgeführt werden
- Sichten können jedesmal neu erzeugt werden oder nur einmal und dann gespeichert (Materialized View)
- Gespeicherte Sichten müssen nach jedem Update der Basisrelationen geändert werden
- Korrespondenz zum externen Schema bei ANSI SPARC (Benutzer sieht jedoch mehrere Views und Basisrelationen)

```
CREATE VIEW view [(attribute-list)]
AS table-exp
```

```
CREATE VIEW Bad_Sailors ( sid, rating, age)
AS SELECT sid, rating, age
    FROM Sailors
    WHERE rating < 3
```

Sichtkonzept (Forts.)

- Vorteile
  - Erhöhung der Benutzerfreundlichkeit (z.B. Verbergen komplexer Joins in einer View)
  - Datenschutz
  - Erhöhte Datenunabhängigkeit
- Abbildung von Sicht-Operationen auf Basisrelationen
  - Sichtoperationen werden in äquivalente Operationen auf Basisrelationen umgesetzt (bei nichtgespeicherten Views)
  - Umsetzung für Leseoperationen einfach

```
SELECT sid, rating
FROM Bad_Sailors
WHERE age > 50
```

- Einschränkungen
  - Keine Schachtelung von GROUP BY und Standardfunktionen
  - Keine Standardfunktionen in WHERE-Klausel

```
SELECT AVG(gsumme)
FROM Abtinfo
```

```
CREATE VIEW Abtinfo (anr,gsumme)
AS
AS SELECT anr, SUM(gehalt)
     FROM Pers
     GROUP BY anr
```

View-Update-Problem

- Änderungsoperationen auf Sichten erfordern, daß die Tupel der Basisrelationen, die jedem Tupel der Sicht zugrundeliegen, eindeutig identifizierbar sind
- Sichten auf einer Basisrelation sind nur änderbar, wenn der Primärschlüssel in der Sicht enthalten ist
- Beispiel

  INSERT INTO Bad_Sailors (sid, rating, age)
  VALUES (789, 1, 20)
- Dieses Tupel kann eindeutig auf die darunterliegende Basisrelation abgebildet werden (fehlende Werte wie *sname* werden durch NULL aufgefüllt); Constraints beachten!
- Sichten, die über Standardfunktionen oder Gruppenbildung definiert sind, sind nicht aktualisierbar
- Sichten über mehr als eine Relation sind i.allg. nicht aktualisierbar
- Löschen von Sichten:
  DROP VIEW Bad_Sailors

Embedded SQL

- SQL-Kommandos können innerhalb einer Wirtssprache (Host Language) aufgerufen werden, z.B. C, COBOL, Java
  - SQL-Anweisungen können sich auf Host-Variablen beziehen (einschließlich spezieller Status-Variablen)
  - Müssen einen Befehl zum Connect mit der richtigen Datenbank beinhalten

```
EXEC SQL SELECT  S.sname, S.age
         INTO    :c_sname, :c_age
         FROM    Sailors S
         WHERE S.sid=:c_sid;
```

- Impendance Mismatch zwischen SQL und Host Language Language), z.B. C, COBOL, Java
  - Problem 1: SQL ist mengenorientiert
    - SQL-Relationen sind Mengen von Sätzen ohne Festlegung der Anzahl (keine solche Datenstruktur in C)
    - SQL unterstützt einen Cursor-Mechanismus, um das behandeln
  - Problem 2: Datentypen von SQL vs. Datentypen der Host Language
    - SQL bietet Casting-Operator (wie andere Programmiersprachen auch)

Cursor-Konzept

- Cursor können auf einer Relation oder einer Query-Anweisung (welche eine Relation generiert) definiert werden
- Ablauf:
  - Cursor wird eröffnet (OPEN)
  - wiederholt wird ein Tupel aus der Ergebnismenge in der Hostvariablen gespeichert (FETCH)
  - Wenn keine Tupel mehr vorliegen (Test der Variablen SQLSTATE): Cursor schließen (CLOSE)
- ORDER BY-Klausel in Queries, auf die über Cursor zugegrifffen wird: bestimmt die Reihenfolge, in der die Tupel geliefert werden
  - Spalte in ORDER BY muß auch in SELECT-Klausel erscheinen
  - ORDER BY-Klausel nur erlaubt im Kontext eines Cursors
- Cursor auch anwendbar auf UPDATE/DELETE
- Queries, die nur ein Tupel liefern (z.B. bei Abfragen über dem Primärschlüssel) benötigen keinen Cursor zur Kommunikation mit der Host Language

Beispiel: Deklaration eines Cursors

Deklariere einen Cursor, der die Namen der Segler liefert, die ein rotes Boot reserviert haben, in alphabetischer Reihenfolge

```
EXEC SQL DECLARE sinfo CURSOR FOR
        SELECT  S.sname
        FROM  Sailors S, Boats B, Reserves R
        WHERE  S.sid=R.sid AND R.bid=B.bid AND
        B.color='red'
        ORDER BY  S.sname
```

- Beachte: Spalten in ORDER BY müssen auch in der SELECT-Klausel enthalten sein
  - Deshalb darf z.B. nicht sname durch sid ersetzt werden

Einbettung von SQL in C: Ein Beispiel

```
char SQLSTATE[6];
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20]; short c_minrating; float c_age;
EXEC SQL END DECLARE SECTION
c_minrating = random();
EXEC SQL DECLARE sinfo CURSOR FOR
  SELECT S.sname, S.age FROM Sailors S
  WHERE S.rating > :c_minrating
  ORDER BY S.sname;
do {
  EXEC SQL FETCH sinfo INTO :c_sname, :c_age;
  printf("%s is %d years old\n", c_sname, c_age);
} while (SQLSTATE != '02000');
EXEC SQL CLOSE sinfo;
```

Weitere Datenbank-API‘s

- Verwendung einer Library mit Datenbank-Calls (API): spezielle standardisierte Interfaces (Prozeduren, Objekte)
- Übergabe von SQL-Strings von der Sprache
- Darstellung der Ergebnismenge in einer sprach-freundlichen Weise
- Microsofts ODBC ist der C/C++ Standard auf Windows
- Suns JDBC ist das Äquivalent für Java
- DBMS-neutral:
  - Ein Driver erfaßt die Calls und übersetzt sie in DBMS-spezifischen Code
  - Datenbank kann überall auf dem Netzwerk sein

Architektur von ODBC/JDBC

- Application
  - Initiiert und terminiert die Verbindung zur Data Source
  - Sendet SQL-Statements, wertet die Resultate aus und bestimmt Transaktionsgrenzen
  - Nutzt als Schnittstelle das API von ODBC/JDBC
- Driver Manager
  - Laden von ODBC/JDBC Drivern
  - Weiterleiten und Loggen von ODBC/JDBC Funktionsaufrufen an den jeweils zuständigen Driver
  - ODBC/JDBC Initialisierung
- Driver
  - Stellt die Verbindung zur Data Source her
  - Absenden von Requests, Empfang der Resultate
  - Übersetzung von Daten- und Fehlerformaten, Fehlercodes (spezifisch für Data Source) in den ODBC/JDBC-Standard
- Data Sources
  - Ausführung der Kommandos und Rückgabe der Ergebnisse

Dynamisches SQL

- Dynamische Festlegung von SQL-Anweisungen bzw. deren Parameter
- Query-Optimierung erst zur Laufzeit möglich
- Sollte nur wo notwendig eingesetzt werden
- Weitere Aspekte: Parameter-Passing zur Host Language
- Beispiel:

```
char c_sqlstring[]={"DELETE FROM
Sailors WHERE rating > 5"};
EXEC SQL PREPARE readytogo FROM :c_sqlstring;
EXEC SQL EXECUTE readytogo;
```

- Deklaration einer String-Variable und Wertzuweisung
- Parsen und Compilieren des String als SQL-Kommando
- Ausführung des Kommandos
  - readytogo ist eine SQL-Variable

SQL API in Java (JDBC)

```
Connection con = // connect
   DriverManager.getConnection(url, ”login", ”pass");
Statement stmt = con.createStatement(); // set up stmt
String query = "SELECT name, rating FROM Sailors";
ResultSet rs = stmt.executeQuery(query);
try { // handle exceptions
    // loop through result tuples
    while (rs.next()) {
        String s = rs.getString(“name");
        Int n = rs.getFloat(“rating");
        System.out.println(s + "   " + n);
    }
} catch(SQLException ex) {
    System.out.println(ex.getMessage()
        + ex.getSQLState()+ ex.getErrorCode());
}
```

Zusammenfassung

- Einfachheit
  - Relativ leichte Erlernbarkeit (natürlicher als frühe prozedurale Sprachen oder Relationenalgebra)
  - Gleichförmigkeit der Syntax
  - Einheitliche Datendefinition und -manipulation
  - Benutzung “stand alone“ oder eingebettet in Host Language
  - Sichtkonzept
- Mächtigkeit
  - Relational vollständig (mehr Ausdrucksmöglichkeiten als relationale Algebra)
  - Standardfunktionen, Sortierung, Gruppenbildung
- Hohe Datenunabhängigkeit
  - Vollständiges Verbergen physischer Aspekte wie Existenz von Indexstrukturen: physische Datenunabhängigkeit
  - Leistungsoptimierung durch DBMS
  - Sichtkonzept unterstützt z.T. logische Datenunabhängigkeit
- Weitere Features
  - Integritätskontrolle (Trigger, Constraints)
  - Zugriffskontrolle

Nachteile von SQL

- Mangel an Orthogonalität
- Fehlende formale Definition
  - Dadurch oft “Ausnahmen“ zu beachten
- Rechnerisch unvollständig (z.B. Berechnung von transitiver Hülle nicht möglich)
- Mangelnde Zusammenarbeit mit Wirtssprachen
  - Cursor: “one record at a time“
- Unklare Semantik von Nullwerten
- Stark zunehmende Komplexität durch SQL-92 und SQL-99
- Query-Optimierung
  - Kenntnisse über Query-Optimierung hilfreich bei der Formulierung von Queries