SQL

DDL (Data Definition Language) Befehle und DML(Data Manipulation Language)

DML(Data Manipulation Language)
SQL Abfragen

Studenten

MatrNr	Name	Vorname	Email	Age	Gruppe
1234	Schmidt	Hans	schmidt@cs.ro	21	331
1235	Meisel	Amelie	meisel@cs.ro	22	331
1236	Krause	Julia	krause@cs.ro	21	332
1237	Rasch	Lara	rasch@cs.ro	21	331
1238	Schmidt	Christian	schmidtC@cs.ro	22	332

Kurse

KursId	Titel	ECTS
Alg1	Algorithmen1	6
DB1	Datenbanken1	6
DB2	Datenbanken2	5

Enrolled

MatrNr	KursId	Note
1234	Alg1	7
1235	Alg1	8
1234	DB1	9
1234	DB2	7
1236	DB1	10
1237	DB2	10

SELECT

```
SELECT *
FROM Studenten S
WHERE S.Age = 21
```

- gibt alle 21-jährige Studenten aus:

1234	Schmidt	Hans	schmidt@cs.ro	21	331
1236	Krause	Julia	krause@cs.ro	21	332
1237	Rasch	Lara	rasch@cs.ro	21	331

SELECT

- Um nur die Namen und Email Adressen auszugeben:

```
SELECT * → SELECT S.Name, S.Vorname, S.Email
```

Schmidt	Hans	schmidt@cs.ro
Krause	Julia	krause@cs.ro
Rasch	Lara	rasch@cs.ro

Was gibt die folgende Abfrage aus?

```
SELECT S.Name, E.KursId
FROM Studenten S, Enrolled E
WHERE S.MatrNr=E.MatrNr AND E.Note=10
```

Krause	DB1
Rasch	DB2

SELECT Klausel

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

wobei:

- *relation-list* – eine Liste von Relationen
- *target-list* – Liste von Attributen aus der Relation in *relation-list*
- *qualification* – Bedingungen mit Vergleichsoperatoren (<,>,=,...) und logischen Operatoren (AND, OR, NOT)
- *DISTINCT* – ist optional, eliminiert Duplikate aus dem Ergebnis

Achtung!

Die zwei Abfragen

```
SELECT S.Name, E.KursId
FROM Studenten S, Enrolled E
WHERE S.MatrNr=E.MatrNr AND E.Note=10
```

und

```
SELECT Name, KursId
FROM Studenten, Enrolled
WHERE Studenten.MatrNr=Enrolled.MatrNr AND  Enrolled.Note=10
```

sind äquivalent.

- *Range variables* (S,E) sind nötig wenn die gleiche Relation zwei mal in der FROM Klausel erscheint. Aber, es ist gut immer *range variables* zu benutzen.

Finde Studenten, die wenigstens eine Note haben.

```
SELECT S.MatrNr
FROM Studenten S, Enrolled E
WHERE S.MatrNr=E.MatrNr
```

- Würde **DISTINCT** einen Unterschied machen?
- Was passiert, wenn wir anstatt MatrNr den Namen ausgeben? Brauchen wir **DISTINCT** dann?

Eine Abfrage mit LIKE-Bedingung und arithmetische Ausdrücke

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

- Die **LIKE**-Bedingung vergleicht Zeichenketten „ungenau“. Dazu werden Wildcards benutzt:
  - Der Unterstrich '_' steht für ein beliebiges einzelnes Zeichen, das an der betreffenden Stelle vorkommen kann
  - Das Prozentzeichen '%' steht für eine beliebige Zeichenkette mit 0 oder mehr Zeichen
- S.Name LIKE ‘B_%B’ – der Name beginnt und endet mit B und enthält wenigstenst 3 Buchstaben
- „=“ und „AS“ haben die gleiche Rolle hier

UNION

- Vereinigung zweier Relationen, die kompatibel Wertebereiche haben; Duplikate werden eliminiert
- Z.B. Geben sie Studenten aus, die Noten in einem 5 ECTS oder in einem 6 ECTS Kurs haben

```
SELECT E.MatrNr
FROM Enrolled E, Kurse K
WHERE E.KursId = K.KursId
AND K.ECTS = 5
UNION
SELECT E.MatrNr
FROM Enrolled E, Kurse K
WHERE E.KursId = K.KursId
AND K.ECTS = 6
```

Alternative Abfrage:

```
SELECT E.MatrNr
FROM Enrolled E,Kurse K
WHERE E.KursId = K.KursId
AND (K.ECTS = 5 OR
     K.ECTS = 6)
```

INTERSECT

- Was passiert wenn wir “oder” mit “und” ersetzen?
- Gebe die Studenten aus, die Noten in einem 5 ECTS und in einem 6 ECTS Kurs haben
- INTERSECT = Durchschnitt zweier Relationen, die kompatibel Wertebereiche haben

INTERSECT

```
SELECT E.MatrNr
FROM Enrolled E, Kurse K
WHERE E.KursId = K.KursId
AND K.ECTS = 5
INTERSECT
SELECT E.MatrNr
FROM Enrolled E, Kurse K
WHERE E.KursId = K.KursId
AND K.ECTS = 6
```

Alternative:

```
SELECT E1.MatrNr
FROM Kurse K1, Enrolled E1,
      Kurse K2, Enrolled E2
WHERE E1.MatrNr = E2.MatrNr AND
E1.KursId = K1.KursId AND
E2.KursId = K2.KursId AND
K1.ECTS = 5 AND
K2.ECTS = 6
```

EXCEPT

Gibt alle Studenten aus, die in ‘Datenbank II’ angemeldet sind, aber nicht in ‘Datenbank I’:

```
SELECT E.MatrNr
FROM Enrolled E, Kurse K
WHERE E.KursId = K.KursId
AND K.Titel = ‘Datenbanken II’
EXCEPT
SELECT E.MatrNr
FROM Enrolled E, Kurse K
WHERE E.KursId = K.KursId
AND K.Titel = ‘Datenbanken I’
```

Nested Queries (Verschachtelte Abfragen)

- Eine WHERE Klausel kann in einer anderen Abfrage enthalten sein.
- Gebe die Namen der Studenten aus, die für den Kurs ‘Alg1’ angemeldet sind

```
SELECT S.Name
FROM Studenten S
WHERE S.MatrNr IN (SELECT E.MatrNr
                   FROM Enrolled E
                   WHERE E.KursId = ’Alg1’)
```

Nested Queries (Verschachtelte Abfragen)

- Alternative Abfrage:

```
SELECT S.Name
FROM Studenten S
WHERE EXISTS (SELECT *
              FROM Enrolled E
              WHERE E.MatrNr = S.MatrNr
              AND E.KursId = 'Alg1')
```

- **EXISTS** und **IN** sind Vergleichsoperatoren für Mengen.
- Wir können auch **NOT IN** und **NOT EXISTS** benutzen.

ANY, ALL

- **ANY** – das Ergebnis ist True (wahr) wenn die Bedingung True ist für **wenigstens ein** Element aus der Ergebnis der Unterabfrage (sub-query)
- **ALL** – das Ergebnis ist True (wahr) wenn die Bedingung True ist für **alle** Elemente aus der Ergebnis der Unterabfrage (sub-query)

Gebe alle Studenten aus, die älter sind als ein Student mit dem Namen „Hans“

```
SELECT *
FROM Studenten S
WHERE S.age > ANY (SELECT S2.age
                   FROM Studenten S2
                   WHERE S2.Name='Hans')
```

Gebe die Studenten aus, die Noten in einem 5 ECTS und in einem 6 ECTS Kurs haben

```
SELECT E.MatrNr
FROM Enrolled E, Kurse K
WHERE E.KursId = K.KursId
AND K.ECTS = 5
INTERSECT
SELECT E.MatrNr
FROM Enrolled E, Kurse K
WHERE E.KursId = K.KursId
AND K.ECTS = 6
```

- **INTERSECT** Abfragen können mit **IN** umgeschrieben werden:

```
SELECT E.MatrNr
FROM Enrolled E, Kurse K
WHERE E.KursId = K.KursId
AND K.ECTS = 5
AND E.MatrNr IN (SELECT E2.MatrNr
                 FROM Enrolled E2, Kurse K2
                 WHERE E2.KursId = K2.KursId
                 AND K.ECTS = 6)
```

- Ähnlich kann man **EXCEPT** Abfragen mit **NOT IN** umschreiben.

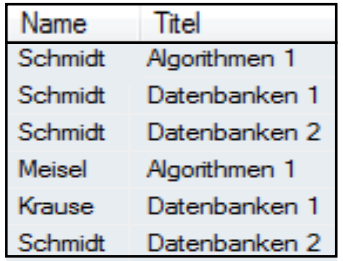
EXCEPT

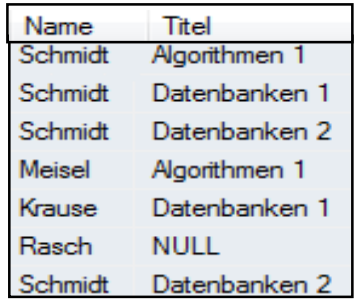
Gibt alle Studenten aus, die in ‘Datenbank II’ angemeldet sind, aber nicht in ‘Datenbank I’:

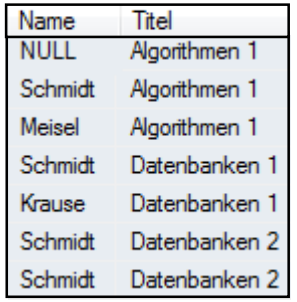
```
SELECT E.MatrNr
FROM Enrolled E, Kurse K
WHERE E.KursId = K.KursId
AND K.Titel = ‘Datenbanken II’
AND E.MatrNr NOT IN (SELECT E.MatrNr
                     FROM Enrolled E, Kurse K
                     WHERE E.KursId = K.KursId
                     AND K.Titel = ‘Datenbanken I’ )
```

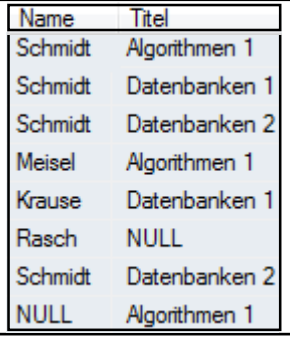
JOIN Abfragen

Join Typ	Abfrage	Ergebnis

INNER JOIN	`SELECT S.Name, K.Titel`
	`FROM Studenten S`
	`INNER JOIN Enrolled E`
	`  ON S.MatrNr = E.MatrNr`
	`INNER JOIN Kurse K`
	`  ON E.KursId = K.KursId`
		

LEFT OUTER JOIN
(Studenten, die nie für einen Kurs angemeldet waren)
Alle Tupel aus der linken Relation, die keinen Join-Partner in der rechten Relation haben, werden trotzdem ausgegeben
	`SELECT S.Name,K.Titel`
	`FROM Studenten S`
	`LEFT OUTER JOIN Enrolled E`
	`  ON S.MatrNr = E.MatrNr`
	`LEFT OUTER JOIN Kurse K`
	`  ON E.KursId=K.KursId`
		

RIGHT OUTER JOIN
(Finde alle Noten, die falsch eingetragen wurden/zu keinem Studenten gehören)
	`SELECT S.Name,K.Titel`
	`FROM Studenten S`
	`RIGHT OUTER JOIN Enrolled E`
	`  ON S.MatrNr = E.MatrNr`
	`RIGHT OUTER JOIN Kurse K`
	`  ON E.KursId=K.KursId`
		

FULL OUTER JOIN
(LEFT + RIGHT OUTER JOIN)
	`SELECT S.Name,K.Titel`
	`FROM Studenten S`
	`FULL OUTER JOIN Enrolled E`
	`  ON S.MatrNr = E.MatrNr`
	`FULL OUTER JOIN Kurse K`
	`  ON E.KursId=K.KursId`
		

NULL Werte

- Manchmal sind die Werte für bestimmte Attribute in einem Tupel unbekannt/unknown oder inapplicable (nicht anwendbar). Dann werden diese mit NULL bezeichnet.
- Wenn eine Tabelle NULL Werte enthält werden viele Sachen komplizierter:
  - man muss bestimmte Operatoren benutzen um zu prüfen ob ein Wert Null ist oder nicht
  - Wie sollte die Bedingung age>8 ausgewertet werden wenn age Null ist? Was passiert für AND, OR und NOT
- Lösung: wir brauchen 3-valued Logik : **true, false, unknown**
- Wir müssen manchmal die Nullwerte extra raussuchen um sie zu beseitigen für eine Abfrage.
- Outer Joins können benutzt werden um Null Werte rauszusuchen.

Aggregatfunktionen

- werden auf eine Menge von Tupeln angewendet
- Verdichtung einzelner Tupeln yu einem Gesamtwert
- SUM, AVG, MIN, MAX können nur auf Zahlen angewendet werden

X
1
1
4
6

- **SUM (X)** → 12
- **AVG(X)** → 3
- **MAX(X)** → 6
- **MIN(X)** → 1
- **COUNT(X)** → 4
- **Duplikat-Eliminierung:** **COUNT(DISTINCT X)** → 3
- **Behandlung von Null-Werten:** **COUNT(X)** zählt jeweils nur die Anzahl von Werten in X, die von NULL verschieden sind

Aggregation - GROUP BY und HAVING

- Anwendung: wenn wir Tupeln gruppieren wollen um Aggregatfunktionen auf bestimmte Gruppen anzuwenden
- z.B. Finde den Alter des jüngsten Studenten aus jeder Gruppe
  - wir wissen nicht wie viele Gruppen es gibt
  - es muss generell funktionieren, nicht nur für die Gruppen die jetzt in der Tabelle existieren

Anfragen mit GROUP BY und HAVING

- Basisschema:

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

Aufpassen!

- Alle Spalten bei **SELECT**, die nicht in einem Aggregat-Ausdruck (mit **SUM()**, **COUNT()** etc.) auftauchen, müssen in der **GROUP BY**-Klausel stehen
- D.h. `target-list` kann folgendes enthalten:
  - Attribute, die auch in der `grouping-list` sind
  - Aggregationsfunktionen (z.B. MIN(S.age))
- Ausdrücke im `group-condition` dürfen ein einziges Wert per Gruppe haben
  - eigentlich enthalten `group-condition` Attribute aus der `grouping-list` oder Aggregatfunktionen

Aufpassen!

- Intuitiv: jedes Tupel gehört zu einer Gruppe und diese Attribute (die wir für die Gruppierung benutzt haben) haben ein einziges Wert für die ganze Gruppe.
- Gruppe = Menge von Tupels, die denselben Wert haben für alle Attribute in der grouping-list

Group by konzeptuelle Evaluation

- das Kartesische Produkt der Relationen wird berechnet
- Tupeln, für welche condition nicht wahr ist werden rausgeworfen
- für den Rest: Tupel mit gleichen Werten für die angegebenen Attribute (`grouping-list`) werden in Gruppen zusammengefasst
- Gruppen für welche group-condition nicht wahr ist werden rausgeworfen.
- Pro Gruppe erzeugt die Anfrage ein Tupelder Ergebnisrelation (Deshalb: Hinter der **SELECT**-Klausel sind nur Attribute mit einem Wert pro Gruppe zugelassen)

Gruppieren mit Ordnen

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE condition
GROUP BY grouping-list
HAVING group-condition
ORDER BY attribute-list [ASC |DESC]
```

Finde das Alter des jünsten Studenten mit Alter >=20 für jede Gruppe mit wenigstens 2 solche Studenten

```
SELECT S.gruppe, MIN(S.age)AS Jungste
FROM Studenten S
WHERE S.age >= 20
GROUP BY S.gruppe
HAVING COUNT(*) > 1
```

Finde die Anzahl der angemeldeten Studenten und den Mittelwerte der Noten für alle 6 ECTS Kurse

```
SELECT K.KursId, COUNT(*) as Anzahl, AVG(Note)as
DurchschnittNote
FROM Enrolled E, Kurse K
WHERE E.KursId = K.KursId
AND K.ECTS = 6
GROUP BY K.KursId
```

BETWEEN

- eine Möglichkeit den Intervall für ein Attribut zu bestimmen ist BETWEEN

```
SELECT *
FROM Enrolled
WHERE NOT Note is NULL AND Note between 7 and 9
```

TOP

- TOP (expression) [PERCENT] [WITH TIES]

```
SELECT TOP(1) WITH TIES E.MatrNr
FROM Enrolled E
WHERE E.KursID='BD'
ORDER BY E.Note DESC
```

- `WITH TIES ->` gibt alle Tupeln aus, die denselben Wert für das Attribut nachdem geordnet wurde haben, auch wenn die totale Anzahl die angegebene Limit überschreitet
- Es ist ein guter Praxis `ORDER BY` zusammen mit `TOP` zu benutzen um genau zu wissen, welche Tupeln ausgegeben werden

DDL (Data Definition Language) Befehle

CREATE TABLE- Anlegen einer Relation

```
CREATE TABLE table-name (column-definition-list)
```
wobei `column-definition-list = attribut-name type [NOT NULL]`

Beispiel:

```
CREATE TABLE Kurse
(KursId CHAR(20),
 Titel CHAR(50),
 ECTS INTEGER)

CREATE TABLE Enrolled
(MatrNr CHAR(20),
 KursId CHAR(20),
 Note REAL)
```

DROP TABLE – Relationenschema löschen

```
DROP TABLE table-name
```

Bsp.

```
DROP TABLE Kurse
```

ALTER TABLE – Ändern einer Relation

- es gibt viele mögliche Änderungen an das Relationenschema

Beispiele:

- ein neues Attribut hinzufügen

```
ALTER TABLE table-name
ADD column-name type
```

Bsp.:

```
ALTER TABLE Studenten
ADD erstesJahr INTEGER
```

- ein Attribut löschen

```
ALTER TABLE table-name
DROP COLUMN column-name
```

Beachte: Werte des neuen Attributes bestehenderTupel werden mit Nullwerten belegt

Primärschlüssel und Fremdschlüssel

- beim Anlegen einer Relation können auch Primärschlüssel und Fremdschlüssel angegeben werden

```
CREATE TABLE Enrolled
(MatrNr CHAR(20),
 KursId CHAR(20),
 Note REAL,
 PRIMARY KEY (MatrNr, KursId),
 CONSTRAINT FK_Enrolled_Studenten FOREIGN KEY (MatrNr)
REFERENCES Studenten,
 FOREIGN KEY (KursId) REFERENCES Kurse)
```

- In SQL können auch Kandidatschlüssel definiert werden mit Hilfe von UNIQUE.

Referenz-Integritätsregel

ON DELETE/UPDATE:

- NO ACTION – Tupel wird nicht gelöscht (default Lösung)
- CASCADE – rekursives Löschen
- SET NULL/SET DEFAULT – Nullsetzen aller darauf verweisender Fremdschlüssel. Kann nur verwendet werden, wenn Null-Werte für das Attribut erlaubt sind

```
CREATE TABLE Enrolled
(MatrNr CHAR(20),
 KursId CHAR(20),
 Note REAL,
 PRIMARY KEY (MatrNr, KursId),
 FOREIGN KEY (MatrNr) REFERENCES Studenten
 ON DELETE CASCADE
 ON UPDATE SET NULL,
 FOREIGN KEY (KursId) REFERENCES Kurse)
```

Default Constraint

Wenn das Wert eines Attributes nicht explizit einen Wert bekommt, wird anstatt einen Null-Wert ein Default-Wert dafür genommen.

```
ALTER TABLE table-name
ADD CONSTRAINT constraint-name
DEFAULT value FOR column
```

Bsp.

```
ALTER TABLE Enrolled
ADD CONSTRAINT defaultNote
DEFAULT 0 FOR Note
```

Integritätsregeln

```
CREATE TABLE Studenten
(MatrNr CHAR(20),
 Name CHAR(50),
 Vorname CHAR(50),
 Email CHAR(30),
 Age INTEGER,
 Gruppe INTEGER,
 PRIMARY KEY (MatrNr),
 CONSTRAINT ageInterval
 CHECK (age >= 18
       AND age<=70))
```

Constraints löschen

```
ALTER TABLE table-name
DROP CONSTRAINT constraint-name
```

DML(Data Manipulation Language)
Einfügen, Löschen und Ändern von Tupeln

Einfügen von Tupeln

```
INSERT INTO table-name [(column-list)]
VALUES (values-list)
```

Bsp.

```
INSERT INTO Kurse (KursId, Titel, ECTS)
VALUES ('Alg1', 'Algebra 1', 5)
```

Bulk – Einfügen

```
INSERT INTO table-name [(column-list)]
<select statement>
```

Bsp.

```
INSERT INTO Enrolled (MatrNr, KursId, Note)
SELECT MatrNr, 'Alg1', 10
FROM Studenten
```

Einfügen

Spalten und Werte müssen nicht angegeben werden, wenn :

- NULL-Werte erlaubt sind oder
- DEFAULT-Werte gesetzt sind oder
- AUTO-INCREMENT angegeben wurde

Bsp.

- Titel kann NULL sein
- ECTS nicht angegeben → Default-Wert 0

```
INSERT INTO Kurse (KursId)
VALUES (‘Alg3’)
```

Beispiel AUTO-INCREMENT

- die Tabelle Reviews hat einen AUTO-INCREMENT Id
- AUTO-INCREMENT in MS SQL Server → IDENTITY(1,1)

```
CREATE TABLE Reviews
(ID int IDENTITY(1,1) PRIMARY KEY,
ReviewContent char(500))
```

Löschen und Aktualisieren
DELETE – Löschen von Tupeln

```
DELETE FROM table-name
[WHERE condition]
```

Bsp.

```
DELETE FROM Studenten S
WHERE S.Name = ‘Schmidt’
```

UPDATE – Verändern von Tupeln

```
UPDATE table-name
SET column-name = expression
[,column-name2 = expression2]
[WHERE condition]
```

Bsp.

```
UPDATE Studenten S
SET S.age = S.age + 1
WHERE S.MatrNr = 123
```