Zürcher Hochschule
für Angewandte Wissenschaften

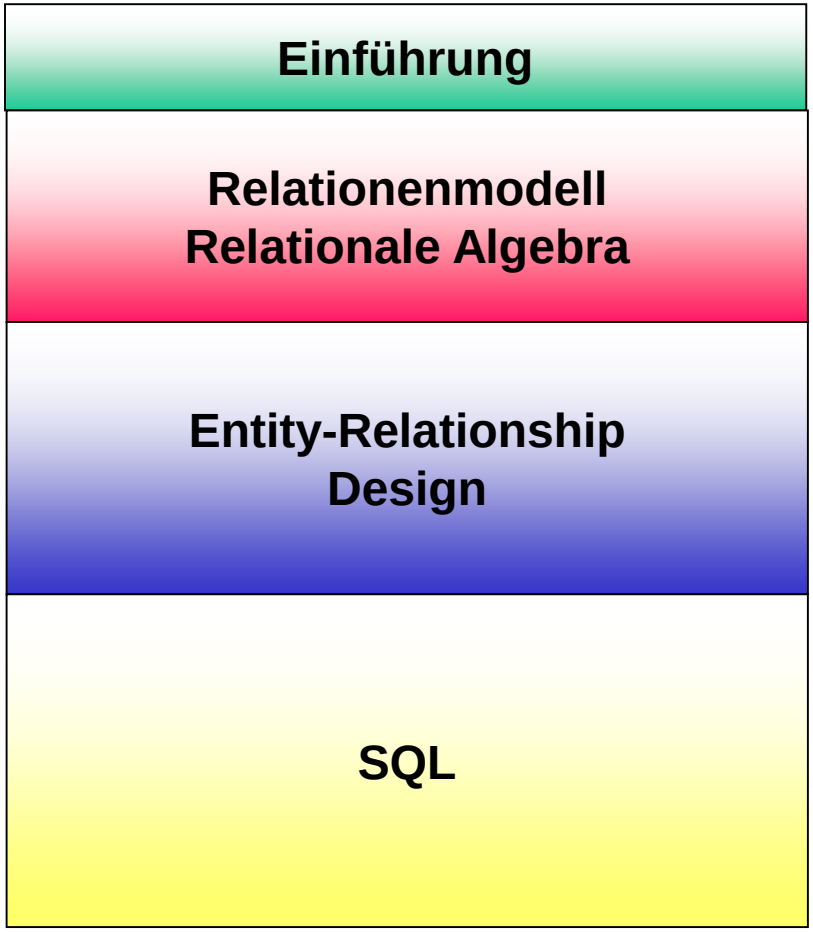
School of
Engineering
InIT Institut für angewandte
Informationstechnologie

DAB1 – Datenbanken 1

Dr. Daniel Aebi (aebd@zhaw.ch)

Lektion 10: SQL – DML, DQL

Wo stehen wir?



"You are here"

Rückblick

Diskutiert im Unterricht. Machen Sie Ihre eigenen Notizen.

Lernziele Lektion 10

- Begriff CONSTRAINT kennen
- Abbildung ER-Schema → SQL verstehen
- Elementare SQL-DML-Befehle anwenden können
- Grundstruktur der SQL-SELECT-Anweisung kennen

CONSTRAINTS

- CONSTRAINTS bezeichnen «Einschränkungen»

- Ein CONSTRAINT schränkt die Menge der möglichen Daten ein, die in eine Tabelle eingegeben werden kann, Beispiele:
  - Primärschlüssel: Schlüsselwerte können nur einmal vorkommen
  - Schlüssel: Schlüsselwerte können nur einmal vorkommen
  - Fremdschlüssel: Nur Werte möglich, die als Primärschlüsselwerte in der referenzierten Tabelle vorkommen
  - CHECK-Klausel
  - … (Datentypen, NOT NULL, …)

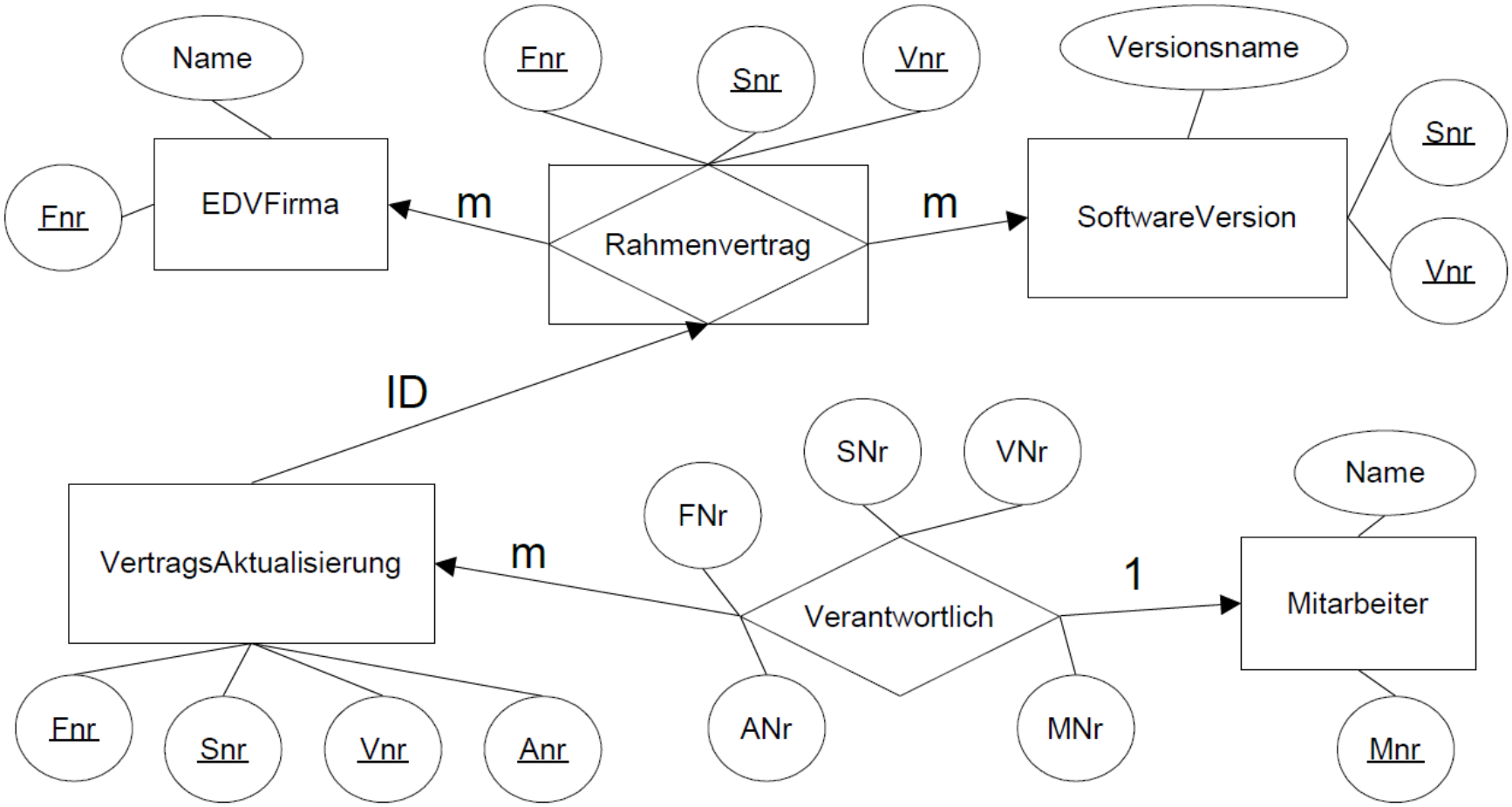
- CONSTRAINTS können einen Namen haben und sind dann als Datenbankobjekte verwaltbar (erzeugen, ändern, löschen)

ER-Schema → Relationenformat

- Jedes «Kästchen» (jeder Entitäts- und jeder Beziehungstyp) ergibt ein Relationenformat. Unabhängige Entitätstypen zuerst.
- Reihenfolge der Attribute beliebig (es lohnt sich aber trotzdem, darüber nachzudenken).
- Alle Fremdschlüssel-Attribute müssen im Relationenformat aufgeführt werden.
- Primärschlüssel-Attribute werden auch bei der Dokumentation des Relationenformats unterstrichen. Empfehlung: Jede Tabelle erhält einen Primärschlüssel.
- Reihenfolge ist gleich wie beim Erstellen eines korrekten ER-Diagramms.

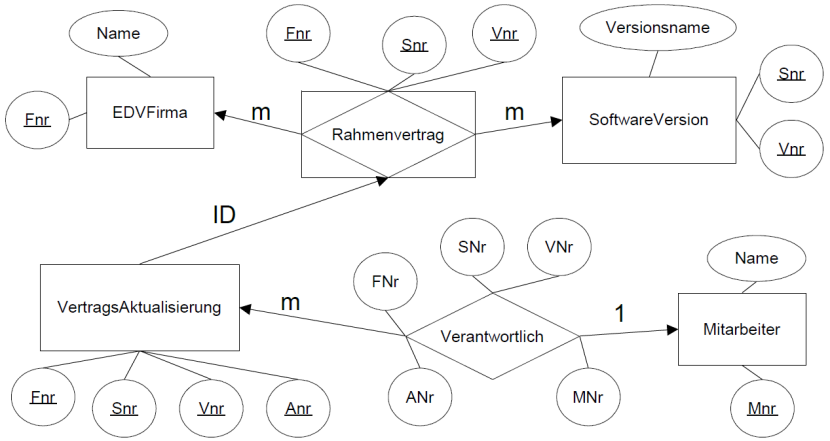
Bsp.: ER-Schema → Relationenformat

- Aufgabe aus Praktikum 8:



Bsp.: ER-Schema → Relationenformat

- Aufgabe aus Praktikum 8:



- Unabhängige Entitätstypen:
  - EDVFirma(FNr,Name)
  - SoftwareVersion(SNr,VNr, Versionsname)
  - Mitarbeiter(MNr,Name)

- Abhängige Entitäts-/Beziehungstypen:
  - Rahmenvertrag(FNr,SNr,VNr)
  - Vertragsaktualisierung(FNr,SNr,VNr,ANr)
  - Verantwortlich(FNr,SNr,VNr,ANr,MNr)

Bsp.: Relationenformat → Tabellen

- Jedes Relationenformat wird zu einer Datenbank-Tabelle.
- Die Tabellen bestehen aus Spalten = Attributen (Struktur-Information) und Zeilen = Tupel (Daten).
- Die Tupel in einer Tabelle sind NICHT geordnet (obschon es optisch natürlich eine „1. Zeile“ gibt), SQL folgt hier der Mengentheorie.
- Demzufolge gibt es in (Standard-)SQL auch keine Befehle wie z.B. „gib das zehnte Tupel zurück“.
- Die Reihenfolge der Attribute ist beliebig (aber in der Praxis wichtig!)
- Ist die Attributs-Reihenfolge einmal festgelegt, spielt sie jedoch eine Rolle: z.B. sind <A, B> und <B, A> verschiedene Schlüssel!
- Jedes Tupel muss eindeutig identifizierbar sein → Unique-Schlüssel ist nötig (besser: IMMER Primärschlüssel definieren).
- Abfragen können Tupel mit Duplikaten liefern = relationaler Bag.

Bsp.: Relationenformat → Tabellen

- Unabhängige Entitätstypen:
  - EDVFirma(FNr, Name)

```
CREATE TABLE EDVFirma(
  FNr  integer     NOT NULL,
  Name varchar(50) NOT NULL,

  CONSTRAINT PK_EDVFirma PRIMARY KEY(FNr)
);
```

Nicht nötig, aber empfohlen

  - Sehr wichtig: Wahl der Attributnamen und der Datentypen!!!

Bsp.: Relationenformat → Tabellen

- Unabhängige Entitätstypen:
  - SoftwareVersion(SNr, VNr, Versionsname)

```
CREATE TABLE SoftwareVersion(
  SNr integer NOT NULL,
  VNr integer NOT NULL,
  Versionsname varchar(20) NOT NULL,

 CONSTRAINT PK_SoftwareVersion PRIMARY KEY(SNr,VNr)
);
```

Bsp.: Relationenformat → Tabellen

- Unabhängige Entitätstypen:
  - Mitarbeiter(MNr, Name)

```
CREATE TABLE Mitarbeiter(
  MNr  integer     NOT NULL,
  Name varchar(50) NOT NULL,

  CONSTRAINT PK_Mitarbeiter PRIMARY KEY(MNr)
);
```

Bsp.: Relationenformat → Tabellen

- Abhängige Entitäts-/Beziehungstypen:
  - Rahmenvertrag(FNr, SNr, VNr)

```
CREATE TABLE Rahmenvertrag(
  FNr integer NOT NULL,
  SNr integer NOT NULL,
  VNr integer NOT NULL,

  CONSTRAINT PK_Rahmenvertrag PRIMARY KEY(FNr,SNr,VNr),
  CONSTRAINT FK_EDVFirma FOREIGN KEY(FNr)
    REFERENCES EDVFirma(FNr),
  CONSTRAINT FK_SoftwareVersion FOREIGN KEY(SNr,VNr)
    REFERENCES SoftwareVersion(SNr,VNr)
);
```

  - Fremdschlüssel aus dem Schema lesen

Bsp.: Relationenformat → Tabellen

- Abhängige Entitäts-/Beziehungstypen:
  - Vertragsaktualisierung(FNr, SNr, VNr, ANr)

```
CREATE TABLE Vertragsaktualisierung(
  FNr integer NOT NULL,
  SNr integer NOT NULL,
  VNr integer NOT NULL,
  ANr integer NOT NULL,

  CONSTRAINT PK_Vertragsaktualisierung PRIMARY KEY(FNr,SNr,VNr,ANr),
  CONSTRAINT FK_Rahmenvertrag FOREIGN KEY(FNr,SNr,VNr)
    REFERENCES Rahmenvertrag(FNr,SNr,VNr)
);
```

Bsp.: Relationenformat → Tabellen

- Abhängige Entitäts-/Beziehungstypen:
  - Verantwortlich(FNr, SNr, VNr, ANr, MNr)

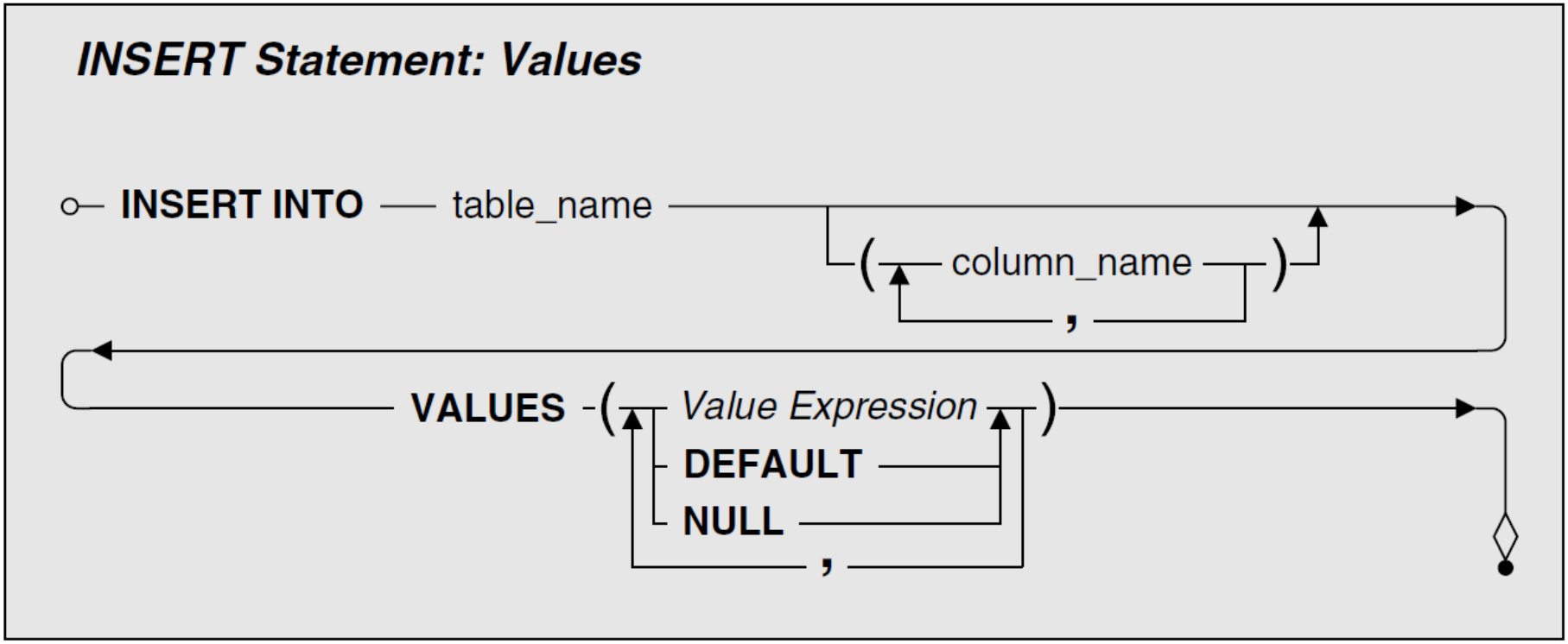
```
CREATE TABLE Verantwortlich(
  FNr integer NOT NULL,
  SNr integer NOT NULL,
  VNr integer NOT NULL,
  ANr integer NOT NULL,
  MNr integer NOT NULL,
  CONSTRAINT UK_Verantwortlich UNIQUE(FNr,SNr,VNr,ANr),
  CONSTRAINT FK_Vertragsaktualisierung FOREIGN KEY(FNr,SNr,VNr,ANr)
    REFERENCES Vertragsaktualisierung(FNr,SNr,VNr,ANr),
  CONSTRAINT FK_Mitarbeiter FOREIGN KEY(MNr)
    REFERENCES Mitarbeiter(MNr)
);
```

DML: Daten einfügen, ändern löschen

- DML = Data Manipulation Language
- Der Teil von SQL, der den Inhalt, nicht die Struktur, von Datenbanken verändert.
- SQL-Befehle: INSERT, UPDATE, DELETE (seit SQL:2003 auch MERGE)
- In der Praxis auch wichtig: Werkzeuge für «bulk loading»

	DDL	DML
Einfügen / erzeugen	CREATE	INSERT
Ändern	ALTER	UPDATE
Löschen	DROP	DELETE

DML: Daten einfügen



- Insert fügt immer ganze Tupel in eine Tabelle ein.

DML: Daten einfügen

- Einfügen von Daten in eine Tabelle:

```
INSERT INTO Student (SNo,SName,Adresse)
VALUES('87-604-1','Meier','Basel');
```

- `Student`: Name der Tabelle, in die Daten eingefügt werden sollen.
- `(SNo,SName,Adresse)`: Liste von Attributen, für die ein Wert eingefügt werden soll.
- `('87-604-1','Meier','Basel')`: Liste von Attributwerten, die eingefügt werden sollen.

- Anzahl und Datentypen müssen zueinander passen. Die Reihenfolge spielt aber keine Rolle.

DML: Daten einfügen

- Bemerkungen:
  - Die Attributnamen können weggelassen werden (→ schlechter Stil, warum?)
  - Bei fehlenden Attributwerten wird der Default-Wert (oder NULL) eingetragen, sofern definiert bzw. erlaubt.
  - Attribute und Attributwerte (explizit angegeben oder implizit über die Position festgelegt) müssen zusammenpassen.
  - Die Reihenfolge kann geändert werden, wenn die Attribute angegeben werden.

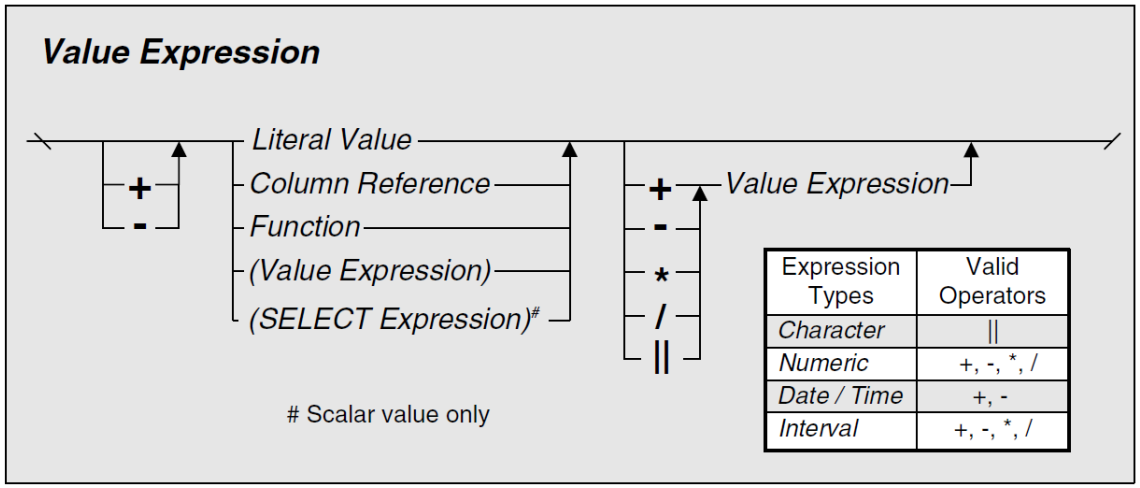
```
INSERT INTO Student VALUES('87-604-1','Meier','Basel');

INSERT INTO Student (Adresse,SNo,SName)
VALUES('Basel','87-604-1','Meier');
```

- Bei unzulässigen Daten wird nichts eingefügt. Schlüsselbedingungen etc. werden überprüft.

DML: Daten einfügen

- Nebst Konstanten können auch komplexere Ausdrücke benutzt werden:

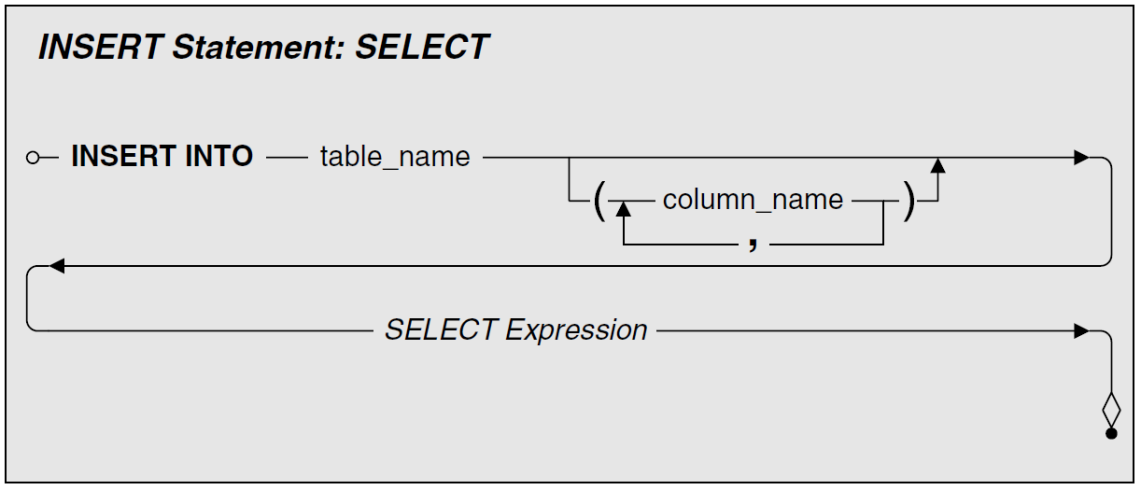


Expression Types	Valid Operators
Character	||
Numeric	+, -, *, /
Date / Time	+, -
Interval	+, -, *, /

```
INSERT INTO Employees (EmployeeID, EmpFirstName, EmpLastName)
VALUES (
   (SELECT MAX(EmployeeID) FROM Employees) + 1,
    'Peter',
    'Meier'
);
```

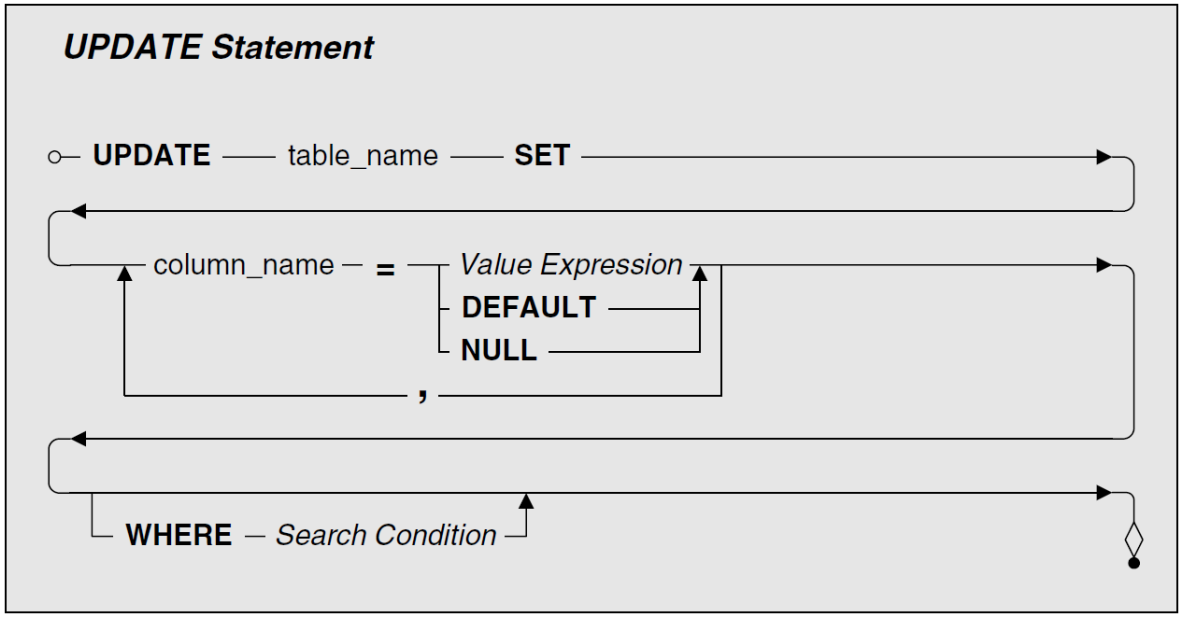
DML: Daten einfügen

- Es kann auch das Resultat einer Abfrage eingefügt werden (siehe später):




```
INSERT INTO Employees (EmpFirstName, EmpLastName)
SELECT Customers.CustFirstName,
       Customers.CustLastName
FROM Customers
WHERE Customers.CustomerID IN (1,4,9);
```

DML: Daten ändern



- UPDATE wird in der Regel mit einer Selektion verbunden.
- Beispiel:

```
UPDATE Salaer SET Betrag = Betrag * 1.05 WHERE ID = 3;
```

DML: Daten ändern

- Ändern von Daten in einer Tabelle:

```
UPDATE Student
SET Adresse = 'Zürich'
WHERE SNo = '87-604-1';
```

- `Student`: Name der Tabelle, in der Daten geändert werden sollen.
- `Adresse = 'Zürich'`: Zu änderndes Attribut und neuer Wert.
- `SNo = '87-604-1'`: Selektionskriterium, sagt aus, bei welchen Tupeln geändert werden soll.

- Anzahl und Datentypen müssen passen. Die Reihenfolge spielt keine Rolle.

DML: Daten ändern

- Bemerkungen:
  - Wenn die WHERE-Klausel weggelassen wird, werden ALLE Tupel geändert (also Vorsicht!)
  - Es können mehrere Attributwerte in einer Anweisung gleichzeitig geändert werden.
  - Attribute und Attributwerte müssen zusammenpassen.

```
UPDATE Student
SET Adresse = 'Zürich', SName = 'Maier'
WHERE SNo = '87-604-1';
```

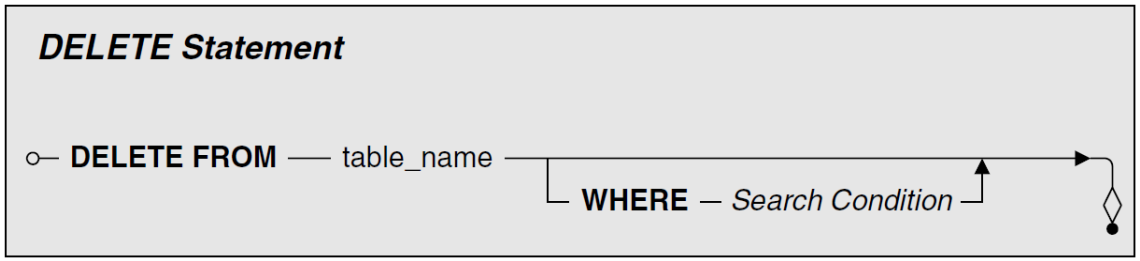
- Bei unzulässigen Daten wird nichts geändert. Geändert werden stets alle angegebenen Attribute. Schlüsselbedingungen etc. werden überprüft.

DML: Daten ändern

- Bemerkungen:
  - Für die neuen Attributwerte können auch Berechnungen (*, /, -, +) mit bereits bestehenden Attributwerten gemacht werden.

```
UPDATE Belegt
SET Note = Note + 0.5
WHERE SNo = '87-604-1';
```

DML: Daten löschen



- DELETE löscht immer ganze Tupel!
- DELETE wird in der Regel mit einer Selektion verbunden.
- Beispiel:

```
DELETE FROM Salaer WHERE Betrag > 100000;
```

- Ohne SearchCondition wird die ganze Tabelle «geleert»!

DML: Daten löschen

- Löschen von Daten in einer Tabelle:

```
DELETE FROM Student
WHERE SNo = '94-555-1';
```

- `Student`: Name der Tabelle, in der Daten gelöscht werden sollen.
- `SNo = '94-555-1'`: Kriterium, welche Tupel gelöscht werden sollen.

DML: Daten löschen

- Bemerkungen:
  - Wenn die WHERE-Klausel weggelassen wird, werden ALLE Tupel gelöscht (also noch mehr Vorsicht!)
  - Es werden immer ganze Tupel gelöscht.
  - «Löschen» von Attributwerten geht nicht (nur NULL eintragen, sofern erlaubt)

```
UPDATE Student SET Adresse = NULL WHERE SNo = '94-555-1';
```

- Bei Verletzung von Schlüsselbedingungen oder wenn die WHERE-Klausel keine Treffer ergibt, wird nichts gelöscht.

DML: Bemerkungen

- DML-Anweisungen werden üblicherweise von Anwendungsprogrammen generiert und nicht manuell eingegeben.
- Einzufügende oder zu ändernde Daten können auch aus einer Abfrage stammen.
- Bei allen DML-Anweisungen wird überprüft, ob die Anweisung ausgeführt werden kann:
  - Verletzung von Datentypen
  - Schlüsselbedingungen (auch Primär-/Fremdschlüssel)
  - CHECK-Klauseln
  - …
- Alle Anweisungen werden ganz oder gar nicht ausgeführt.

SQL – DML: Hörsaalübung (geleitet)

1. Starten Sie MySQL-Workbench und lassen Sie das Skript Lektion_10_DB.sql laufen (zu finden auf OLAT).

2. Formulieren Sie dann die notwendigen SQL-Anweisungen um alle Daten der Studenten Meier und Imboden einzugeben:

SNo	SName	Address	DNo	DName	Course	Grade
87–604–I	Meier	Basel	IIIC	Informatik	Informatik	6
87–604–I	Meier	Basel	IIIC	Informatik	Analysis	5
87–604–I	Meier	Basel	IIIC	Informatik	Physik	4
91–872–I	Schmid	Bern	IIIC	Informatik	Informatik	5
91–872–I	Schmid	Bern	IIIC	Informatik	Analysis	3
91–109–I	Anderegg	Zürich	IIIC	Informatik	Informatik	4
94–555–P	Imboden	Luzern	IX	Mathematik	Algebra	3

3. Meier zieht von Basel nach Zürich um. Führen Sie die Datenbank nach.

4. Imboden wird exmatrikuliert. Führen Sie die Datenbank nach.

DQL: Übersicht

- Häufigste Anwendung von SQL: Daten abfragen.
- Wenige Befehle, aber komplexe Verschachtelungen möglich.
- Ist an relationale Algebra angelehnt, setzt diese aber nicht exakt um:
  - gewisse Befehle eliminieren Duplikate nicht automatisch.
- → Widerspruch zur Definition von Mengen
  - Bei Duplikaten erhält man einen relationalen Bag (kompliziertere Algebra)
  - NULL's folgen einer dreiwertigen Logik (true, false, unknown)
    → unknown = «so falsch, dass auch das Gegenteil nicht stimmt»
- Schlussfolgerung: NULL's schon im DB-Design soweit sinnvoll vermeiden (was wir mit unserer Design-Methode auch tun).

DQL: Query

- Query ::= <subquery> {("**UNION**" | "**INTERSECT**"[2] | "**EXCEPT**"[2]) ["**ALL**" | "**DISTINCT**"[1]] <subquery>} ";"[3]

  subquery ::= vollständige SQL-Abfrage

- Abfragen können also verschachtelt sein. Egal, wie kompliziert eine Abfrage ist, liefert sie jedoch nur EIN Resultat als neue Tabelle

[1] Der Defaultwert ist unterstrichen. Wird die Angabe weggelassen, so wird automatisch der Defaultwert eingesetzt

[2] INTERSECT ist der SQL-Ausdruck für Durchschnitt (∩), EXCEPT der für Differenz (\)

[3] SQL-Statements werden in den meisten DBMS mit ";" abgeschlossen, gehört jedoch nicht zum Standard. Manche Systeme verlangen dies sogar explizit.

Einfache *Subquery*: SELECT-Klausel

```
subquery ::= "SELECT" ["ALL" | "DISTINCT"] <attributeList>
             "FROM" <tableName> ";"
```

- Das ist die einfachste Form einer Subquery. Wir werden später Verfeinerungen kennenlernen.
- Beispiel CDShop:

```
SELECT[1] titel, musikgruppe FROM CD;
```

[1] SQL ist nicht case-sensitive, "SELECT" bedeutet dasselbe wie "Select" oder "sElEct"

- «SELECT» ist historisch «gewachsen». Im folgenden wird nicht immer die gebräuchlichste Form einer Abfrage zuerst gezeigt, da sich das Vorgehen an den Erkenntnissen aus den früheren Vorlesungen orientiert.

SELECT und Projektion

- Diese Form des SELECT entspricht der Projektion in der Welt der Bags:

```
SELECT Name
FROM Besucher;
```

- entspricht π_{Name}(Besucher), während

```
SELECT DISTINCT Name
FROM Besucher;
```

Duplikatelimination

- äquivalent zu $\delta(\pi_{Name}(\text{Besucher}))$ ist.

Einfache *Subquery*: SELECT-Klausel

attributeList::= (<*columnSpec*> {"," <*columnSpec*>} | "*"[1])

columnSpec ::= <*scalarExpr*> ["**AS**" <neuer spaltenName> [2]]

scalarExpr ::= (<*columnRef*> | <literal>) {<*operator*> <*scalarExpr*>}

operator ::= ("**+**" | "**-**" | "*****" | "**/**")

columnRef ::= [[<schemaName> "."] <tableName> "."][3] <attributeName>

Achtung: Default ist KEINE Duplikatelimination (SELECT ALL)

1 * wählt alle Attribute der *Datenquelle* aus

2 Rename-Operation = Umbenennen von Attributen im Resultat

3 <attributeName> muss in der *Datenquelle* eindeutig sein → soviele Angaben, bis Eindeutigkeit gewährleistet ist

Einfache *Subquery*: SELECT-Klausel

- Beispiel CDShop: (Kommentare in SQL mit "--" starten, in MySQL "-- ")

```
1)
  SELECT * FROM CD   -- alle Information aus der Tabelle CD listen
2)
  SELECT DISTINCT betrag FROM Salaer;
  -- jetzt erscheint jeder Betrag nur noch einmal -> Resultat kann weniger
  -- Zeilen haben als es in der Salaer-Tabelle Tupel gibt
3)
  SELECT betrag AS Gehalt -- Attribut umbenennen
  FROM Salaer;
4)
  SELECT betrag * 1.025 -- oder Salaer.betrag oder CDShop.Salaer.betrag
  FROM Salaer;
  -- wenn im Beispiel 2 mehrere Mitarbeiter dasselbe Salär verdienen, gibt
  -- es im Resultat auch mehrere Zeilen mit demselben Betrag
5)
  SELECT 1  -- Literal
  FROM Hierarchie; -- listet so oft '1' auf, wie es Hierarchie-Tupel gibt
```

Einfache *Subquery*: Datenquelle

Datenquelle ::= <*tableExpr*> | <*joinExpr*>

tableExpr ::= [<schemaName>"."]<tableName> [["**AS**"] <aliasName>]

joinExpr ::= (<*tableExpr*> | "**(**" <*joinExpr*> "**)**") ["**NATURAL**" | ("**LEFT**" | "**RIGHT**" | "**FULL**") "**OUTER**" | "**CROSS**"] "**JOIN**" <*tableExpr*> ["**ON**" <*joinCondition*>][1]

joinCondition ::= <*attributeComparison*> {("**AND**" | "**OR**") <*attributeComparison*>

attributeComparison ::= ([<tableName>"."]<attributeName> | <literal>) <*theta-Operator*> ([<tableName>"."]<attributeName> | <literal>)

theta-Operator ::= "**<**" | "**<=**" | "**=**" | "**>=**" | "**>**" | "**<>**" [2]

[1] Alle Join-Typen mit Ausnahme des Natural Joins und des Kreuzprodukts (CROSS JOIN) erfordern eine *JoinCondition*

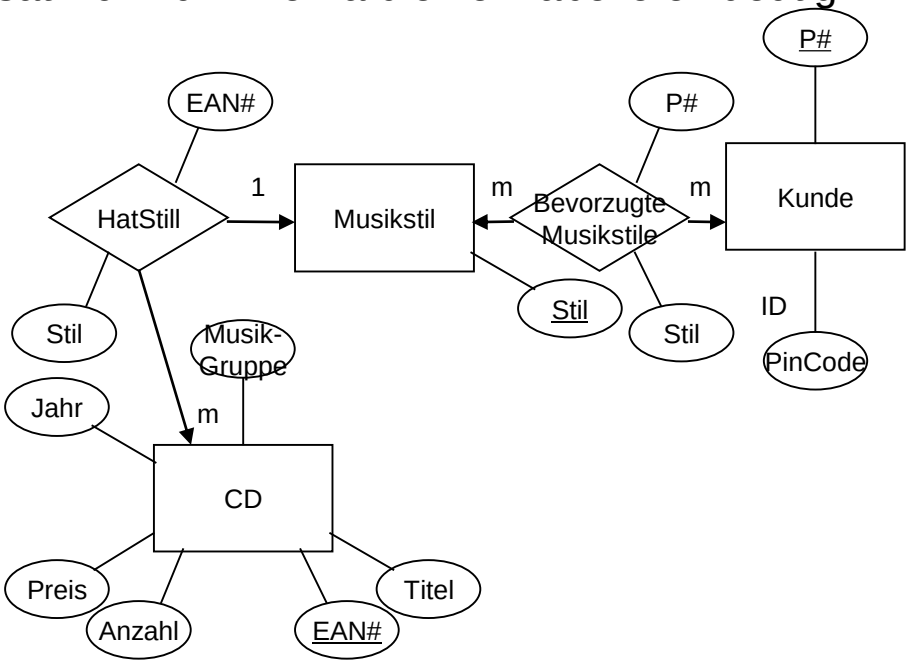
[2] in der Praxis fast ausschliesslich Equi-Join üblich ("=")

SELECT und JOIN

- Diese Form des SELECT entspricht in der Welt der Bags dem Join.
- ACHTUNG: In SQL sind Attributnamen grundsätzlich nur innerhalb einer Tabelle eindeutig.

```
SELECT *
FROM CD CROSS JOIN Musikstil
```

- entspricht CD ⋈ Musikstil, falls die beiden Bags keine gemeinsamen Attribute haben (= Kreuzprodukt)



Die Variante:

```
SELECT *
FROM CD NATURAL JOIN HatStil
```

entspricht aber dem Ausdruck CD ⋈ HatStil in der Bag-Welt (JOIN auf gleich benannten Attributen)

Einfache *Subquery*: Datenquelle

- Weitere Beispiele CDShop:

1)
```
SELECT bestDatum
FROM Bestellung AS B[1] JOIN KaufHistorie KH[1] ON bestNr =[2] bNr;
```
2)
```
SELECT name, stil
FROM (BevorzugteMusikstile NATURAL JOIN[3] Kunde) NATURAL JOIN Person;
```

[3] mit Equi-JOIN:
```
SELECT name, stil
FROM (BevorzugteMusikstile BM
JOIN Kunde K ON BM.pNr = K.pNr[4]) NATURAL JOIN Person;
```

[1] zwei verschiedene Arten, Tabellen umzubenennen

[2] Equi-JOIN, wählt aus beiden Tabellen die Tupel aus, deren Schlüsselwerte übereinstimmen

[4] Tabellenangabe für Eindeutigkeit nötig

Resultat des Equi-Joins hat mehr Spalten als Natural Join, Natural Join eliminiert jeweils eines der übereinstimmenden Attribute

Einfache *Subquery*: Datenquelle

- Obschon die Datenquelle erst an zweiter Stelle steht, ist sie das zentrale Element. Sie muss alle gewünschten Attribute und Tupel enthalten.

- Diese Rohdaten können anschliessend durch weitere Bearbeitung (Select = Attributs-Auswahl / Projektion, *searchCondition* = Tupelauswahl / Selektion) auf das gewünschte Format gebracht werden.

- Was nicht in der Datenquelle ist, kann weder angezeigt noch bearbeitet werden!

Und weiter...

- Das nächste Mal: SQL (Queries, Fortsetzung)