Zürcher Hochschule
für Angewandte Wissenschaften

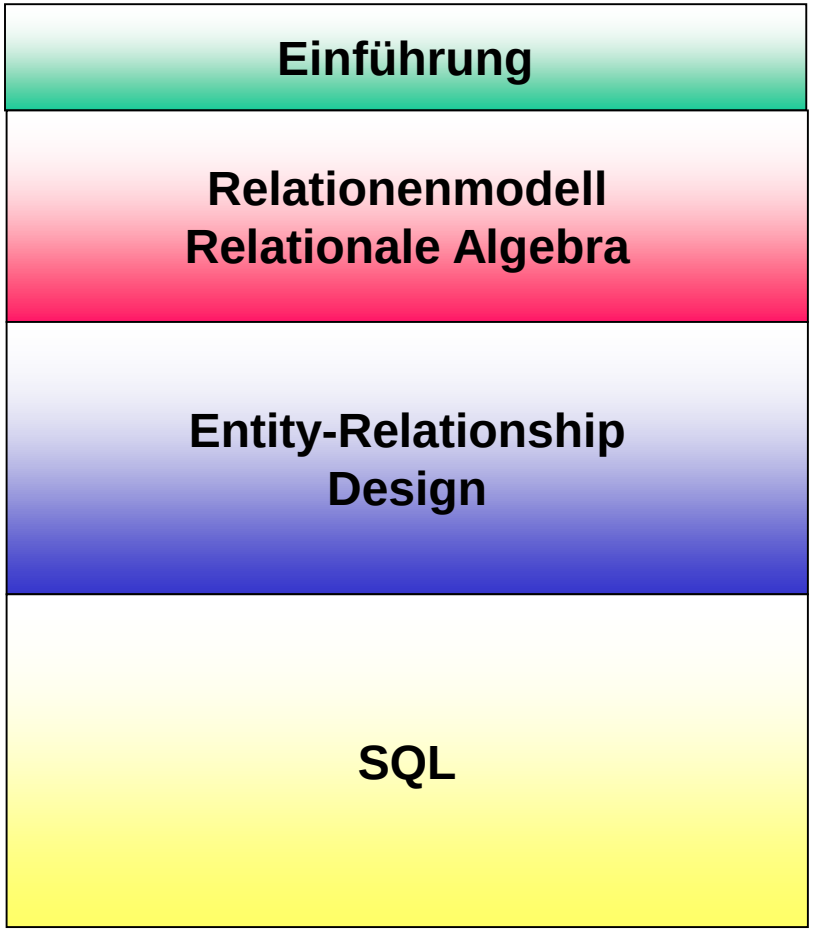
School of
Engineering
InIT Institut für angewandte
Informationstechnologie

DAB1 – Datenbanken 1

Dr. Daniel Aebi (aebd@zhaw.ch)

Lektion 11: SQL – DQL

Wo stehen wir?



"You are here"

Rückblick

- Abbildung ER-Schema → Relationenformate → SQL
- DML-Anweisungen: INSERT, UPDATE, DELETE
- Grundaufbau der SELECT-Anweisung:

SELECT … → welche Attribute (entspricht Projektion)
FROM … → aus welchen Tabellen (eine oder Verbund mehrerer)

Es fehlen noch:

WHERE … → gemäss welchen Kriterien (entspricht Selektion)
GROUP BY … → wie zusammengefasst
ORDER BY … → wie geordnet

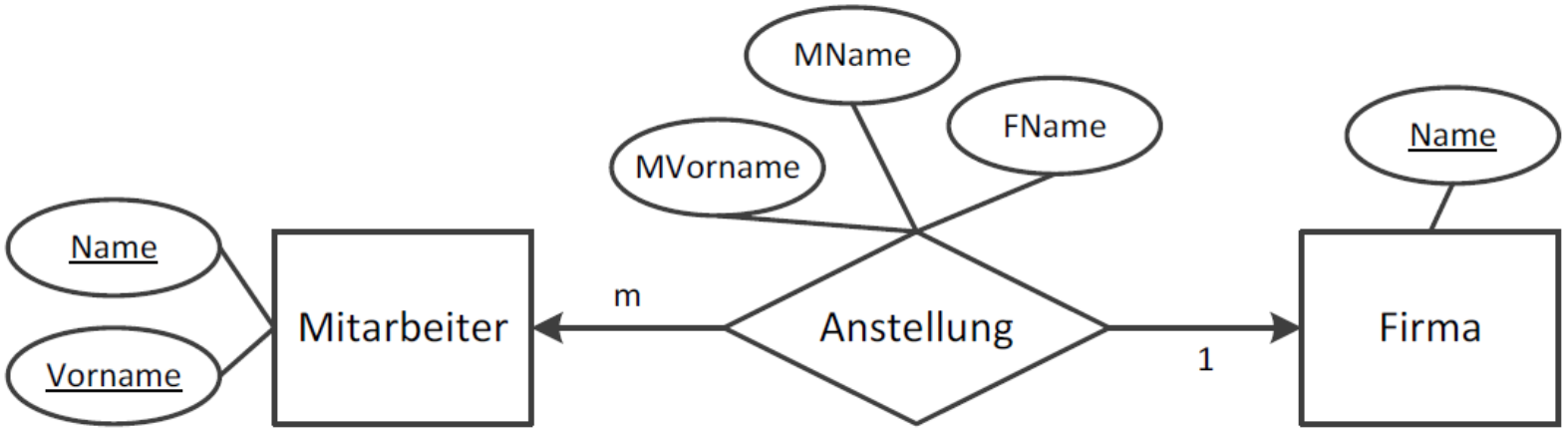
Lernziele Lektion 11

- Nochmals: Abbildung ER-Schema → SQL verstehen
- Weitere Möglichkeiten der SQL-SELECT-Anweisung kennen

ER-Schema → SQL revisited

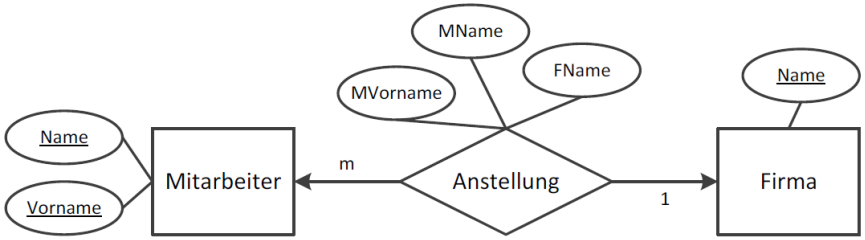
- Aufgabe 1 Praktikum 9:

- Mitarbeiter arbeiten bei höchstens einer Firma. Firmen stellen naturgemäss mehrere Mitarbeiter an. Mitarbeiter haben Vor- und Nachnamen, Firmen jeweils einen Namen.



ER-Schema → SQL revisited

- Aufgabe 1 Praktikum 9:



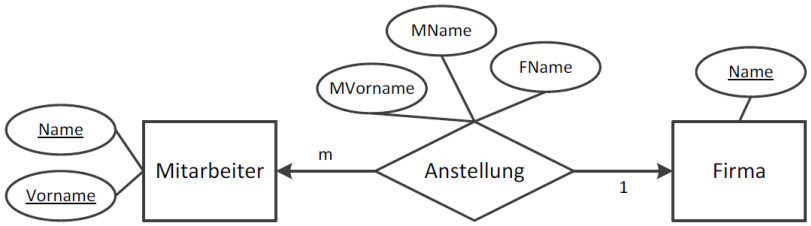
- Unabhängige Entitätstypen:
  - Mitarbeiter (Name, Vorname)
  - Firma(Name)

```
CREATE TABLE Mitarbeiter (
  Name    varchar(100) NOT NULL,
  Vorname varchar(100) NOT NULL,
  CONSTRAINT PK_Mitarbeiter PRIMARY KEY (Name, Vorname)
);

CREATE TABLE Firma (
  Name varchar(100) NOT NULL,            -- auch möglich: Name varchar(100) PRIMARY KEY
  CONSTRAINT PK_Firma PRIMARY KEY (Name) -- auch möglich: PRIMARY KEY (Name)
);
```

ER-Schema → SQL revisited

- Aufgabe 1 Praktikum 9:
- Was machen wir mit der Anstellung?



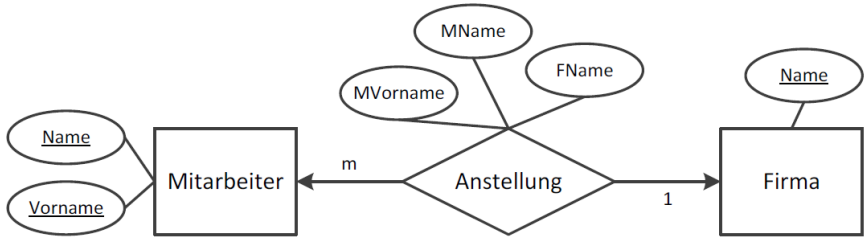
- Vorschlag 1 (so macht man das doch in der «Praxis»):

```
CREATE TABLE Firma (
  Name varchar(100) NOT NULL PRIMARY KEY
);
CREATE TABLE Mitarbeiter (
  Name        varchar(100) NOT NULL,
  Vorname     varchar(100) NOT NULL,
  Firmaname   varchar(100) NOT NULL,
  CONSTRAINT PK_Mitarbeiter PRIMARY KEY (Name, Vorname),
  CONSTRAINT FK_Firma FOREIGN KEY(Firmaname) REFERENCES Firma(Name)
);
```

- Wollten wir wirklich das? Ist das die Aussage des ER-Diagrammes?
- Was machen wir mit Kardinalitäten 1 und 1?

ER-Schema → SQL revisited

- Aufgabe 1 Praktikum 9:
- Was machen wir mit der Anstellung?



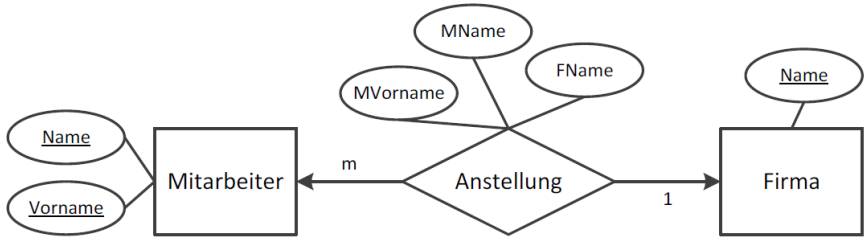
- Lösung (richtig gemäss «unserem» Vorgehen):

```
CREATE TABLE Anstellung (
  MName    varchar(100) NOT NULL,
  MVorname varchar(100) NOT NULL,
  FName    varchar(100) NOT NULL,
  CONSTRAINT UK_Anstellung UNIQUE (MName, MVorname), CONSTRAINT FK_Mitarbeiter FOREIGN KEY (MName, MVorname)
  REFERENCES Mitarbeiter(Name, Vorname), CONSTRAINT FK_Firma FOREIGN KEY (FName) REFERENCES Firma(Name)
);
```

- Vorteile: Entspricht Diagramm, Attribute die zur Anstellung gehören (z.B. der Lohn) können am richtigen Ort 'versorgt' werden. Keine NULL's!
- Nachteil: Eine Tabelle mehr.

ER-Schema → SQL revisited

- Aufgabe 1 Praktikum 9:



- Dumm gelaufen, Mitarbeiter sollten doch in mehreren Firmen arbeiten können (d.h. 1 → m)

- Einfache Lösung (das umgekehrte, d.h. m → 1 geht i.a. nicht, warum?):

```
ALTER TABLE Anstellung DROP CONSTRAINT UK_Anstellung;
ALTER TABLE Anstellung ADD CONSTRAINT UK_Anstellung
  UNIQUE (MitarbeiterName, MitarbeiterVorname, FirmaName);
```

Optimierung – Ablauf

- Reihenfolge der Abarbeitung einer Abfrage in SQL:

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. ORDER BY

Datenquelle

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

- Diese «Rohdaten» können anschliessend durch weitere Bearbeitung

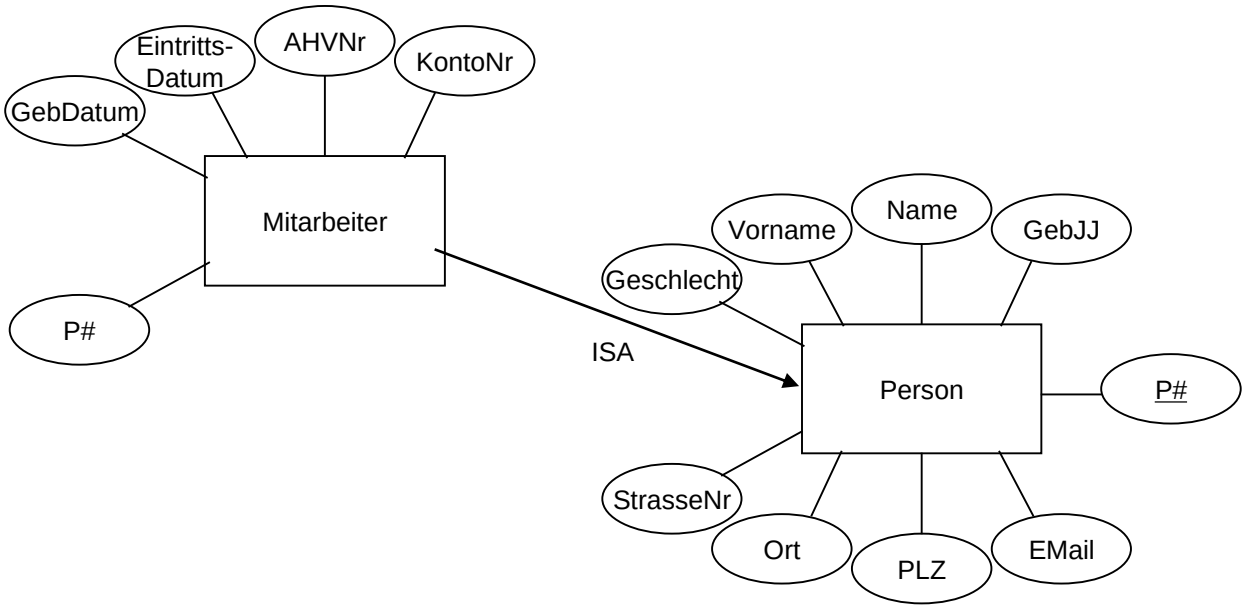
  SELECT = Attributsauswahl (entspricht Projektion),
  WHERE = Tupelauswahl (entspricht Selektion)

  auf das gewünschte Format gebracht werden.

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

Einfache *Subquery*: Beispiele CDShop

? Wo wohnen die Mitarbeiter? Gesucht ist eine Tabelle mit Namen und Ortschaft



Einfache *Subquery*: Beispiele CDShop

```
SELECT name, ort
FROM Person NATURAL JOIN[1] Mitarbeiter;
```

[1] ein Natural Join kann IMMER auch als Equi-Join formuliert werden:

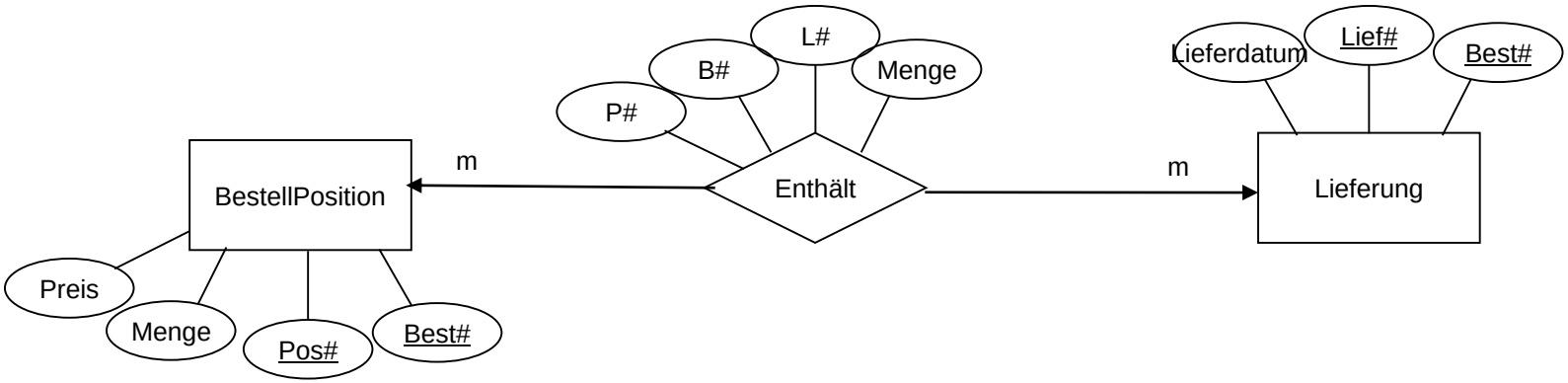
```
SELECT name, ort
FROM Person JOIN Mitarbeiter ON Person.pNr[2] = Mitarbeiter.pNr;
```

[2] hier ist für die Eindeutigkeit die Angabe des <tableName> zwingend

Warum nicht einfach SELECT name, ort FROM Person?

Einfache *Subquery*: Beispiele CDShop

? Welche Bestellpositionen wurden wann (ev. teilweise) geliefert? Gesucht ist eine Tabelle mit bestNr, posNr, bestellMenge, lieferMenge, lieferDatum



Einfache *Subquery*: Beispiele CDShop

```
SELECT
  BestellPosition.bestNr,
  posNr,
  BestellPosition.menge AS BestellMenge,
  Enthaelt.menge AS Liefermenge,
  lieferDatum
FROM
  BestellPosition JOIN Enthaelt ON
  BestellPosition.bestnr[1] = bNr AND
  posNr = pNr[2] JOIN Lieferung ON
  bNr = Lieferung.bestNr[1] AND lNr = LiefNr;
```

[1] auch hier ist für die Eindeutigkeit die Angabe des <tableName> zwingend

[2] Schlüssel besteht aus zwei Attributen → alle in der Join-Condition aufführen

Einfache *Subquery*: Rückblick

- Gesucht ist eine Liste aller Musikstile

```
SELECT stil
FROM Musikstil;
```

- Diese Abfrage ist gleichwertig zu folgender:

```
SELECT *[1]
FROM Musikstil;
```

[1] * wählt alle Attribute der Datenquelle (Musikstil) aus. Die Tabelle hat nur ein Attribut → liefert dasselbe

- Diese Abfrage ist gleichwertig auch zu folgender:

```
SELECT DISTINCT[2] stil
FROM Musikstil;
```

[2] DISTINCT eliminiert Duplikate. Da jedoch stil unique ist, gibt es *per definitionem* keine Duplikate → überflüssig. Tatsächlich erfordert DISTINCT ein internes Sortieren der Tabelle (~ n * log(n) Operationen auf n Tupel) und beeinträchtigt die Performance → nur wenn wirklich nötig!

- Mit der Duplikatelimination ist SPARSAM umzugehen. Es muss nach jedem Schritt überlegt werden, ob eine Elimination notwendig ist → Siehe auch Praktika (prüfungsrelevant!)

Einfache *Subquery*: searchCondition 1

```
SELECT  *
FROM  Datenquelle
WHERE logischerAusdruck;
```

- WHERE-Klausel ist ein Tupelfilter. Alle Tupel, für die der *logischeAusdruck* wahr ist, werden in das Resultat übernommen.
- *logischerAusdruck* kann beliebig komplex sein, z.B. auch verschachtelte Subqueries enthalten.
- ACHTUNG: Suchprädikate werden mit einer dreiwertigen Logik ausgewertet (false, true, unknown). Unknown steht für NULL.

a und b			
a \ b	f	u	w
f	f	f	f
u	f	u	u
w	f	u	w

a oder b			
a \ b	f	u	w
f	f	u	w
u	u	u	w
w	w	w	w

nicht a	
a	¬ *a*
f	w
u	u
w	f

SELECT und Selektion

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

```
SELECT *
FROM Person
WHERE Name = 'Müller';
```

- Entspricht: $\sigma_{Name='Müller'}(Person)$

SELECT und Projektion/Selektion

- Ein einzelner Select-Befehl kann Projektion und Selektion vereinigen:

```
SELECT Name
FROM   Person
WHERE  Name = 'Müller';
```

- Entspricht: $\pi_{Name}(\sigma_{Name=Müller}(Person))$

- Eigentlich sollte dann ja auch gehen:

```
SELECT Name
FROM (SELECT *
      FROM   Person
      WHERE  Name = 'Müller');
```

SELECT und Projektion/Selektion

- Diese Form (SELECT-Befehl als Ausdruck für eine Datenquelle) ist in SQL92 tatsächlich vorgesehen.
- In MySQL muss dem Ausdruck aber eine sog. Bereichsvariable zugeordnet werden:

```
SELECT Name
FROM (SELECT *
      FROM Person
      WHERE Name = 'Müller') AS x;
```

- Auch möglich:

```
SELECT *
FROM (SELECT Name
      FROM Person) AS x
WHERE x.Name = 'Müller';
```

Einfache *Subquery*: Beispiele CDShop

- Alle Mitarbeiter (name, betrag) mit weniger als 50‘000 Gehalt.

```
SELECT name, betrag
FROM salaer NATURAL JOIN person
WHERE betrag < 50000;
```

- Alle Mitarbeiter, die Meier, Maier, Mayer etc. in allen Varianten heissen

```
SELECT name
FROM Mitarbeiter NATURAL JOIN Person
WHERE name LIKE 'M__er';
```

'_' steht für ein einzelnes, beliebiges Zeichen
'%' steht für eine beliebige Anzahl Zeichen
LIKE kann nur für Strings verwendet werden

- Bem.: Text und Datum werden in SQL in einfache Anführungszeichen eingebettet (')

Einfache *Subquery*: Beispiele CDShop

- Alle Personen, deren Namen mit den Anfangsbuchstaben C, D, E beginnen

```
SELECT Name
FROM   Person
WHERE  Name BETWEEN 'C%' AND 'E%';
```

- Alle Lieferantenangebote, bei denen ein Preis fehlt.

```
SELECT eanNr, liefNr
FROM BezugsNachweis
WHERE preis IS NULL; -- Nicht etwa "= NULL" !!
```

- Alle Bestellpositionen, die in einem Mal ausgeliefert wurden, d.h.
  1) es existiert eine zugehörige Lieferposition,
  2) Liefermenge = Positionsmenge

```
SELECT bestNr, posNr
FROM Bestellposition JOIN Enthaelt ON bestNr = bNr AND posNr = pNr
WHERE Bestellposition.menge = Enthaelt.menge;
```

Einfache *Subquery*: searchCondition

searchCondition ::= **"WHERE"** [**"NOT"**] <logicalComparison> { (**"AND"** | **"OR"**) [**"NOT"**] <logicalComparison>}

logicalComparison ::= (<attributeName> | <literal>) { <compareOperator> <attributeName> | <literal> }

compareOperator ::= (**"<"** | **"<="** | **"="** | **">="** | **">"** | **"<>"** | **"LIKE"**[1])

[1] ausschliesslich für Text (String)

- Spezialfall Vergleichsoperator: **"BETWEEN"** attributeValue | literal **"AND"** attributeValue | literal
- Weitere Möglichkeit: **"WHERE"** <attributeName> [**"NOT"**] **"IN"** liste
  liste = (**"("**"<literal> {", " <literal>} **")"** | subquery)
- Beispiel CDShop: alle CD-Titel der Stile Pop oder Heavy Metal

```
SELECT Titel
FROM CD Natural Join HatStil
WHERE Stil IN ('Pop', 'Heavy Metal');
```

SELECT-FROM-WHERE

- Wir haben nun die Operationen Projektion (π), Selektion (σ) und Natural Join (⋈) mit SELECT in SQL nachgebildet

- Ein Select-Kommando deckt typischerweise eine Kombination dieser Operationen ab. Die häufigste Form ist:

```
SELECT Spaltenliste
FROM Datenquelle
WHERE Bedingung
```

- Ein Kommando wie $\pi_{Bname,\ Rname,\ Bsorte}(\sigma_{Name\ =\ 'Anderegg'}(Gast \bowtie Sortiment))$ entspricht ja eigentlich

SELECT-FROM-WHERE

```
SELECT bname, rname, bsorte
FROM (SELECT *
      FROM (SELECT *
            FROM gast NATURAL JOIN sortiment) as x
      WHERE bname='Anderegg') as y;
```

- Kann aber (und wird oft) in der folgenden Kurzform geschrieben werden:

```
SELECT g.bname, g.rname, s.bsorte
FROM gast as g, sortiment as s
WHERE g.rname = s.rname and g.bname = 'Anderegg';
```

SELECT-FROM-WHERE

```
SELECT Spaltenliste FROM a, b WHERE a.x = b.x;
```

- ist also eine Kurzform/Alternativform für

```
SELECT Spaltenliste
FROM a JOIN b ON a.x = b.x
```

- ACHTUNG: Join-Bedingungen und Select-Bedingungen werden gemischt!

```
SELECT Spaltenliste
FROM a, b
WHERE a.x = b.x AND Bedingung;
```

- Diese Form des JOIN wird nicht empfohlen!

Ordnung

- Resultate von SQL-Abfragen sind grundsätzlich in ihrer Ordnung nicht vorgegeben. D.h., dieselbe Abfrage kann (muss aber nicht) ein paar Minuten später ein anders geordnetes Resultat ergeben.
- SQL ist eben eine Mengensprache, Tupel in Relationen sind auch nicht geordnet.

- Man kann SQL-Resultate aber nach beliebigen Attributen ordnen ("sortieren") lassen:

```
SELECT *
FROM Besucher
ORDER BY Strasse, Name;
```

- Absteigend sortieren: "ORDER BY ... DESC"

Ordnung

? Ist durch Verwendung von "ORDER BY" gewährleistet, dass das Resultat bei mehreren gleichen Abfragen immer gleich geordnet ist?

Ordnung

- Wenn verschiedene Tupel gemäss Sortierkriterium gleichwertig sind, kann es weiterhin zu Umsortierungen kommen.
- Dieser Effekt kann ausgeschlossen werden, wenn das Sortierkriterium einen Schlüssel enthält.
- Merke: Spalten werden immer in der gleichen Reihenfolge ausgegeben (= Reihenfolge der Definition).

Lexikographische Ordnung

- Für manche Abfragen wollen wir eine lexikographische Ordnung à la Telefonbuch verwenden.
- Beispiel: Alle Besucher, die in der lexikographischen Reihenfolge zwischen <Meier, Hans> und <Schmid, Joseph> liegen.
- Kann nicht direkt ausgedrückt warden.
- Umschreiben:

`<a,b,c> < <x,y,z>` → `(a<x) OR (a=x AND b<y) OR (a=x AND b=y AND c<z)`

? Wie lautet also die obige Abfrage in SQL?

Lexikographische Ordnung

```
SELECT *
FROM Besucher
WHERE ((Name = 'Meier' AND Vorname >= 'Hans') OR
       (Name > 'Meier' AND Name < 'Schmid') OR
       (Name = 'Schmid' AND Vorname <= 'Joseph'));
```

EXISTS

- Gesucht sind alle Besucher, deren Vorname bei einem weiteren Besucher auch vorkommt.

- Jeder Besucher erfüllt die Bedingung, dass in der gleichen Tabelle ein anderer (= verschiedener) Besucher existiert mit gleichem Vornamen.

EXISTS

- Wir brauchen eine Art von «innerer Schleife»: für jedes Tupel wird die Tabelle nach passenden Gegenstücken «abgegrast»:

```
SELECT *
FROM Besucher
WHERE EXISTS (
    SELECT 1
    FROM Besucher
    WHERE Vorname = Vorname AND Name <> Name
)
```

Liefert nicht das gewünschte Ergebnis! Wir brauchen die gleiche Tabelle zweimal!

EXISTS

```
SELECT *
FROM Besucher AS x
WHERE EXISTS (
    SELECT 1
    FROM Besucher AS y
    WHERE x.Vorname = y.Vorname AND x.Name <> y.Name
)
```

- → Bereichsvariablen
- Neues Konstrukt «EXISTS»: erlaubt uns, solche Existenzbedingungen zu formulieren.

UNION/INTERSECT/EXCEPT

- SQL erlaubt Mengenoperationen (genauer Bag Operationen) auf kompatiblen Tabellenformaten (= gleiche Domänen, gleiche Anzahl Attribute).

- Die Bag Concatenation ⊔ entspricht UNION ALL
- Der Durchschnitt ∩ entspricht INTERSECT ALL
- Die Differenz \ entspricht EXCEPT ALL

- Jeweils auch mit Duplikatelimination kombinierbar:
  UNION DISTINCT/INTERSECT DISTINCT/EXCEPT DISTINCT

- Default ist "DISTINCT"!

Beispiel

- Beispiel: Wir haben 2 Tabellen mit Personendaten (gleiches Tabellenformat) und wollen eine Liste nur derjenigen Einträge, welche in der ersten, nicht aber in der zweiten Liste, vorkommen:

```
SELECT *
FROM Besucher1
EXCEPT ALL
SELECT *
FROM Besucher2;
```

- Achtung: geht nicht in MySQL (kann aber mit NOT EXISTS nachgebildet werden).

Subquery: Aggregatfunktionen ohne Gruppierung

- SQL kennt die 5 Aggregatfunktionen
  - COUNT
  - MAX
  - MIN
  - SUM[1]
  - AVG[1]

[1] Attribute müssen einen zählbaren Domain haben

- Beispiel CDShop: zähle die Anzahl verschiedener Titel der CDs

```
SELECT COUNT(DISTINCT titel) AS anzahlTitel[2]
FROM CD;
```

Alternative Schreibweise:

```
SELECT COUNT(*)
FROM CD;
```

[2] Umbennen den Attributs bietet sich an, weil einige DB-Systeme Phantasie-Namen vergeben → im Resultat nicht ersichtlich, was die Zahl bedeutet

Subquery: Aggregatfunktionen ohne Gruppierung

- Unterschied zwischen COUNT(*) und COUNT(<attributName>):
  - COUNT(<attributName>) zählt nur diejenigen Tupel, bei denen der Wert des Attributs nicht NULL ist
  - COUNT(*) zählt alle Tupel (es gibt kein Tupel, bei dem alle Attribute gleichzeitig NULL sein können)

- Weitere Option:

```
SELECT COUNT (DISTINCT TITEL)
FROM CD;
```

  zählt die Anzahl verschiedener Titel (falls es zufällig verschiedene CDs mit demselben Titel geben sollte).

- Aggregatfunktionen ohne Gruppierung liefern eine Tabelle mit einem Tupel und einem Attribut (also eine Zahl). Ist aber eine «richtige»Tabelle, die weiterverarbeitet werden kann.

Subquery: Aggregatfunktionen mit Gruppierung

- Beispiel CDShop: Anzahl Bestellungen pro Kunde

```
SELECT kdNr, COUNT(*) AS AnzahlBestellungen
FROM Kaufhistorie
GROUP BY kdNr;
```

Liefert eine Tabelle mit einem Tupel pro kdNr, d.h. ein Tupel pro Kunde

- Komplizierteres Beispiel: Bestellumsatz pro Kunde, unabhängig von der einzelnen Bestellung:

```
SELECT kdNr, SUM(menge * preis) AS Umsatz
FROM Bestellposition JOIN Kaufhistorie ON bestNr = bNr
GROUP BY kdNr;
```

Subquery: Aggregatfunktionen mit Gruppierung

- Noch komplizierteres Beispiel: Grösster Bestellumsatz

  Überlegung: Suche dasjenige Tupel der auf der vorigen Folie besprochenen Abfrage mit dem grössten Umsatz

```
SELECT MAX(Umsatz)[1]
FROM vorherige Abfrage;
```

  setzen wir ein:

```
SELECT MAX(Umsatz) AS GroessterUmsatz
FROM (SELECT kdNr, SUM(menge * preis) AS Umsatz
      FROM Bestellposition JOIN Kaufhistorie ON bestNr = bNr
      GROUP BY kdNr) AS x;
```

[1] äussere Subquery ohne Gruppierung !

(Das "AS x" ist nötig in vielen RDBMS, auch in MySQL)

SQL – DQL: Hörsaalübung (geleitet)

- Starten Sie MySQL-Workbench und formulieren Sie in der Datenbank `classicmodels` folgende Abfragen. Ziehen Sie ggf. die Dokumentation (`MySQLSampleDatabaseSchema.pdf`) der Datenbank bei:

1. Gesucht sind Kundennummer und Kundennamen von allen Kunden in Spanien.

2. Gesucht sind Kundenname und Bestelldatum von allen Bestellungen des Kunden mit der Kundennummer 103.

3. Gesucht sind Namen und Telefonnummern der Kunden, die von der Mitarbeiterin mit dem Nachnamen Firrelli betreut werden.

SQL – DQL: Lösungen

1.
```
SELECT customerNumber, customerName
FROM   customers
WHERE  country = 'Spain';
```

2.
```
SELECT customerName, orderDate
FROM   customers NATURAL JOIN orders
WHERE  customerNumber = 103;
```

3.
```
SELECT customerName,phone
FROM   employees
JOIN   customers
ON     employees.employeeNumber =
       customers.salesRepEmployeeNumber
WHERE  lastName = 'Firrelli';
```

Und weiter...

- Das nächste Mal: SQL (Queries, Fortsetzung)