# Datenbanken
## SQL

Nikolaus Augsten

nikolaus.augsten@sbg.ac.at

FB Computerwissenschaften
Universität Salzburg

Wintersemester 2013/14



## Inhalt





## Literatur und Quellen

Lektüre zum Thema "SQL":

- Kapitel 4 aus Kemper und Eickler: Datenbanksysteme: Eine Einführung.
  8. Auflage, Oldenbourg Verlag, 2011.
  7. Auflage: `http://www.oldenbourg-link.com/isbn/9783486592771`

Literaturquellen

- Elmasri and Navathe: Fundamentals of Database Systems. Fourth Edition, Pearson Addison Wesley, 2004.
- Silberschatz, Korth, and Sudarashan: Database System Concepts, McGraw Hill, 2006.

Danksagung Die Vorlage zu diesen Folien wurde entwickelt von:

- Michael Böhlen, Universität Zürich, Schweiz
- Johann Gamper, Freie Universität Bozen, Italien





## Inhalt

## Geschichte/1

- Die IBM Sprache Sequel wurde als Teil des System R Projekts am IBM San Jose Research Laboratory entwickelt.
- Umbenannt in Structured Query Language (SQL)
- ANSI und ISO standard SQL:
  - SQL-86, SQL-89: erste Standards, sehr ähnlich
  - SQL-92 (auch SQL2): größere Revision
    - entry level: entspricht in etwa SQL-89
    - weiters gibt es: intermediate level, full level
  - SQL:1999 (auch SQL3) – Rekursion, reguläre Ausdrücke, Trigger u.A.
  - SQL:2003 – Bug fixes zu SQL:1999, erste XML Unterstützung, WINDOW Funktionen, MERGE Befehl
  - SQL:2006 – Verbesserte XML Unterstützung, Einbindung von XQuery
  - SQL:2008 – viele kleinere Zusätze und Verbesserungen
  - SQL:2011 – Temporal Database Funktionalität
- Kommerzielle Systeme bieten:
  - einen Grossteil der Funktionen von SQL-92
  - eine Anzahl von Funktionen von späteren Standards
  - zusätzliche, proprietäre Funktionen





## Geschichte/2

- Don Chamberlin holds a Ph.D. from Stanford University.
- He worked at IBM Almaden Research Center doing research on database languages and systems.
- He was a member of the System R research team that developed much of today's relational database technology.
- He designed the original SQL database language (together with Ray Boyce, 1947–1974).

`http://researcher.watson.ibm.com/researcher/view.php?person=us-dchamber`





## Modell und Terminologie

- SQL verwendet die Begriffe Tabelle, Spalte und Zeile.
- Vergleich der Terminologie:

| SQL | Relationale Algebra |
|---|---|
| Tabelle | Relation |
| Spalte | Attribut |
| Zeile | Tupel |
| Anfrage | relationaler Algebra Ausdruck |

- In einer Tabelle kann die gleiche Zeile mehrmals vorkommen.
- Zwischen den Zeilen der Tabelle besteht keine Ordnung.





## DDL, DML und DCL

SQL besteht aus drei unterschiedlichen Teilen:

- DDL – Date Definition Language (Datendefinitionssprache): Schema erstellen, z.B. CREATE TABLE
- DML – Data Manipulation Language (Datenmaniuplationssprache), weiter unterteilt in
  - Anfragesprache: Anfragen, die keine Daten ändern, z.B. SELECT
  - Sonstige DML-Befehle: Anfragen, die Daten ändern können, z.B. UPDATE, INSERT, DELETE, COMMIT
- DCL – Data Control Language (Datenkontrollsprache): Zurgriffsrechte verwalten, z.B. GRANT

## Inhalt







## Datendefinitionssprache

Erlaubt die Spezifikation unterschiedlicher Eigenschaften einer Tabelle, zum Beispiel:

- Das Schema einer Tabelle.
- Die Domäne zu jeder Spalte.
- Integritätsbedingungen, welche alle Instanzen erfüllen müssen.
- Indexe (Schlagwortverzeichnisse), die für Tabellen aufgebaut werden sollen.
- Die physische Speicherstruktur jeder Tabelle.





## Vordefinierte Domänen in SQL

| | |
|---|---|
| char(n) | Zeichenkette von maximal $n$ Zeichen; nicht genutzte Zeichen werden mit Leerzeichen aufgefüllt. |
| varchar(n) | Zeichenkette von maximal $n$ Zeichen; variable Speicherlänge |
| integer | Eine ganze Zahl (maximale Grösse ist maschinenabhängig). |
| smallint | Eine kleine ganze Zahl (max. Grösse maschinenabhängig). |
| numeric(p,d) | Festkommazahl mit einer Präzision von $p$ Ziffern, wovon $d$ von diesen Ziffern rechts vom Komma stehen. |
| real, double precision | Gleitkommazahl mit einfacher bzw. doppelter Genauigkeit. Die Genauigkeit ist maschinenabhängig. |
| float(n) | Gleitkommazahl mit einer Genauigkeit von mindestens $n$ binären Ziffern. |





## Create Table

- Eine SQL Tabelle wird mit dem Befehl **create table** definiert:

  **create table** $R$(
  $A_1$ $D_1, A_2$ $D_2, \ldots, A_n$ $D_n$,
  (Integritätsbedingung$_1$),
  $\ldots$,
  (Integritätsbedingung$_k$))

  - $R$ ist der Name der Tabelle
  - $A_i, 1 \leq i \leq n$, ist eine Spalte der Tabelle
  - $D_i$ ist die Domäne von Spalte $A_i$
  - $D_i$ ist von **not null** gefolgt, falls Spalte $A_i$ keine *null*-Werte erlaubt

- Beispiel:

  **create table** *Filialen*(
  *FiName* varchar(15) **not null**,
  *TlfNr* varchar(10),
  *Umsatz* integer)

## Integritätsbedingungen

- Bedingungen auf Spalten: **not null**, **check** (Bedingung auf Spalte)
- Bedingungen auf Tabelle:
  - **unique** $(A_1, \ldots, A_n)$
  - **primary key** $(A_1, \ldots, A_n)$
  - **foreign key** $(A_1, \ldots, A_n)$ **references** $T(B_1, \ldots, B_n)$
  - **check** (Bedingung auf eine oder mehrere Spalten)
- Beispiel: *KoNr* als Primärschlüssel der Tabelle *Konten* definieren:

  **create table** *Konten*(
  *KoNr* integer, *FiName* varchar(30), *Guthaben* integer,
  **check** (*Guthaben* >= 0),
  **primary key** (*KoNr*))
- Beispiel: *KoNum* als Fremdschlüssel in der Tabelle *Kontoinhaber*:

  **create table** *Kontoinhaber*(
  *KuName* varchar(15), *KoNum* integer,
  **foreign key** (*KoNum*) **references** Konten(KoNr))





## Notation/1

- SQL ist eine umfangreiche Sprache und stellt verschiedene syntaktische Konstrukte zur Verfügung, um Tabellen und Integritätsbedingungen zu definieren.
- Oft gibt es mehrere Möglichkeiten, um etwas auszudrücken.
- Die genaue Syntax hängt auch vom Datenbanksystem und oft sogar von der verwendeten Version ab.
- Bei Syntaxproblemen ist die genaue Syntax nachzuschlagen (Manual, Web, Forum).
- Wir verwenden einen kleinen Kern von SQL, der allgemein ist und mehrheitlich unahängig vom Datenbanksystem und der Version ist.





## Notation/2

- Groß- und Kleinschreibung von reseriverten Wörtern:
  - In SQL ist Gross- und Kleinschreibung von reservierten Wörtern irrelevant (z.B. SELECT, select, SeLEct).
  - Im Programmcode werden reservierte Wörter meistens groß geschrieben (Bsp: SELECT).
  - In den Vorlesungsunterlagen verwenden wir Fettschrift für reservierte Wörter (Bsp: **select**).
- Groß- und Kleinschreibung von Bezeichnern:
  - In Bezeichnern kann Gross- und Kleinschreibung eine Rolle spielen (z.B. Tabellennamen in MySQL Linux).
  - Gross- und Kleinschreibung ist relevant, falls man den Bezeichner unter Anführungszeichen stellt (select "KundenName").
  - PostgreSQL verwenden doppelte Hochkommas für Bezeichner ("abcde"), MySQL erlaubt wahlweise Backticks (`abcde`) oder doppelte Hochkommas.
- Das Ende eines SQL Befehls wird oft durch einen Strichpunkt markiert

  **select * from** Konten;





## Drop und Alter Table

- Der **drop table** Befehl löscht alle Informationen einer Tabelle von der Datenbank, z.B. **drop table** *Filiale*
- Der **alter table** Befehl wird verwendet, um neue Spalten zu einer Tabelle hinzuzufügen. Die Werte für die neue Spalte sind:
  - $x$, falls **default** $x$ für die Spalte spezifiziert ist,
  - ansonsten *null*

  Beispiel: Spalte *AnzMitarbeiter* als neuen Spalte vom Typ **integer** in Tabelle *Filialen* einfügen (neue Werte sind *null*)

  **alter table** *Filialen* **add** *AnzMitarbeiter* **integer**
- Der **alter table** Befehl kann auch verwendet werden, um eine Spalte von einer Tabelle zu löschen:

  **alter table** *Filialen* **drop** *TlfNr*

  wobei *TlfNr* der Name einer Spalte von Tabelle *Filialen* ist.

## Zusammenfassung: DDL

- SQL DDL erlaubt
  - das Schema einer Tabellen zu definieren
  - jeder Spalte eine Domäne zuzuordnen
  - Integrationsbedingungen für Spalten anzugeben

  Viele weitere Möglichkeiten, z.B., Indices festlegen.
- Vordefinierte Domänen: **varchar**, **integer**, **float**, ...
- Integritätsbedingungen:
  **not null**, **unique**, **primary key**, **foreign key**, **check**
  SQL kennt noch viele weitere Integrationsbedingungen.
- Schema kann nachträglich mit **alter table** geändert werden.
- Tabellen können mit **drop table** gelöscht werden.





## Inhalt







## Ausdrücke und Prädikate

- Ausdrucksstarke Ausdrücke und Prädikate (Bedingungen) machen Computersprachen anwenderfreundlich.
- Datenbankfirmen messen sich anhand der angebotenen Ausdrücke und Prädikate (sowohl Funktionalität als auch Geschwindigkeit).
- Die effiziente Auswertung von Prädikaten ist ein wichtiger Aspekt on Datenbanksystemen.
- Beispiel: 1 Milliarde Tupel und die folgenden Prädikate:

  Nachname = 'Miller'
  Nachname like 'Ester%'
  Nachname like '%mann'
  length(Nachname) < 5

  - Eine alphabetische Ordnung unterstützt die effiziente Evaluierung des 1. und 2. Prädikats nicht aber des 3. und 4. Prädikats.
  - Das ist einer der Gründe warum die Definition von Prädikaten und Funktionen durch den Benutzer limitiert war/ist.





## Struktur von SQL Anfragen/1

- SQL basiert auf Relationen und relationalen Operatoren mit gewissen Änderungen und Erweiterungen (z.B. Duplikate).
- SQL ist sehr weit verbreitet in der Geschäftswelt.
- SQL ist weit mehr als einfache select-from-where Anfragen wie z.B.:

  ```
  select *
  from Kunden
  where KundenName = 'Bohr'
  ```

- Viele Benutzer/Programmierer...
  - unterschätzen SQL
  - verstehen nicht die Konzepte, die sich hinter der Syntax verbergen
  - verstehen nicht, wie mit einer deklarativen Sprache und mit Mengen zu arbeiten ist (dies braucht eine gewisse Übung)

## Struktur von SQL Anfragen/2

- Eine typische SQL Anfrage hat folgende Form:

**select** Teil
**from** Teil
**where** Teil
**group** Teil
**having** Teil
} *Anfrage–spezifikation*

**union**

**select** Teil
**from** Teil
**where** Teil
**group** Teil
**having** Teil
} *Anfrage–spezifikation*

**order** Teil

} *Anfrage–ausdruck*

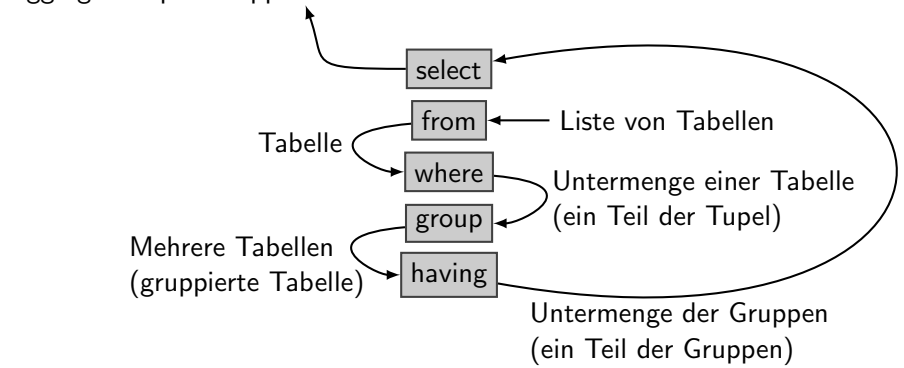
- Das Resultat einer SQL Anfrage ist eine (virtuelle) Tabelle.





## Illustration: Evaluierung einer Anfragespezifikation/1

Berechnet eine Zeile pro Gruppe; oft wird eine Aggregation pro Gruppe berechnet



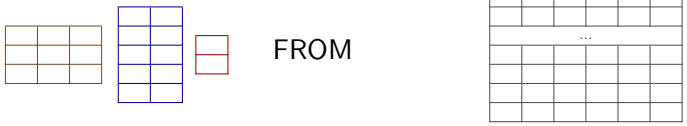

Aggregation: min, max, sum, count, avg einer Menge von Werten.



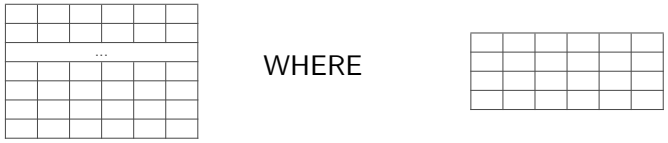


## Illustration: Evaluierung einer Anfragespezifikation/2

1. FROM: bilden des Kreuzprodukts aller Tabellen im **from** Teil




2. WHERE: eliminiert Tupel die die Bedingung im **where** Teil nicht erfüllen




3. GROUP BY: gruppiert Tupel gemäss den Spalten im **group** Teil






## Illustration: Evaluierung einer Anfragespezifikation/3

4. HAVING: eliminiert Gruppen welche die Bedingung des **having** Teil nicht erfüllen


5. SELECT: evaluiert die Ausdrücke im **select** Teil und produziert ein Ergebnistuple für jede Gruppe

## Konzeptionelle Evaluierung eines Anfrageausdrucks

1. Bilden des Kreuzprodukts aller Tabellen im **from** Teil.
2. Eliminierung aller Tupel die den **where** Teil nicht erfüllen.
3. Gruppierung der verbleibenden Tupel gemäss **group** Teil.
4. Eliminierung der Gruppen die den **having** Teil nicht erfüllen.
5. Evaluierung der Ausdrücke im **select** Teil.
6. Für jede Gruppe wird genau ein Resultattupel berechnet
7. Duplikate werden eliminiert falls **distinct** spezifiziert wurde.
8. Anfragespezifikationen werden unabhängig ausgewertet und anschliessend werden die Teilresultate durch die angegebenen **Mengenopertionen** (union, except, intersect) kombiniert.
9. Sortierung des Resultats gemäss **order** Teil.





## Der from Teil

- Der **from** Teil listet die Tabellen, die in der Anfrage involviert sind.
  - Entspricht dem kartesischen Produkt in der relationalen Algebra.
- Kartesisches Produkt von *Kreditnehmer* und *Kredite*
  **from** *Kreditnehmer*, *Kredite*
- Kartesisches Produkt von *Kreditnehmer* und *Kredite* mit Umbenennung:
  **from** *Kreditnehmer* **as** $T$, *Kredite* **as** $S$
- Umbenennung wird notwendig, wenn die gleiche Tabelle mehrmals im **from** Teil vorkommt.
  **from** *Kredite* **as** $K1$, *Kredite* **as** $K2$





## Der where Teil/1

- Der **where** Teil spezifiziert Bedingungen, die Ergebnistupel erfüllen müssen.
- Input: Der **where** Teil arbeitet mit der virtuellen Tabelle, die der **from** Teil produziert und behält alle Zeilen, welche die Bedingung erfüllen.
- Beispiel: Kredite der Brugg Filiale, die grösser als $1200 sind.

  **from** *Kredite*
  **where** *FiName* = 'Brugg' **and** *Betrag* > 1200

| *KrNr* | *FiName* | *Betrag* |
|---|---|---|
| L-260 | Brugg | 1700 |

- Der **where** Teil entspricht dem Selektionsprädikat.
- Prädikate können über die logischen Operatoren **and**, **or**, und **not** verknüpft werden.





## Der where Teil/2

- Der **where** Teil kann verwendet werden, um Join- oder Selektionsbedingungen zu spezifizieren.
- Selektionsbedingung: schränkt Attributwerte einer einzigen Tabelle aus dem **from** Teil ein.
  - **from** Filialen **where** FiName = 'Brugg'
  - **from** Filialen, Kredite **where** Betrag > 12000
- Joinbedingung: verknüpft Tupel von zwei Tabellen durch Prädikate, die jeweils Attribute beider Tabellen enthalten.
  - **from** Kreditnehmer, Kredite **where** KrNo = KrNr

## Integrierte Übung 4.1

Übersetzen Sie die folgenden Ausdrücke der relationalen Algebra in äquivalente SQL Fragmente:

1. $R \times S$
   **from** R, S
2. $(R \times S) \times T$
   **from** R, S, T
3. $\sigma_{A>5}(R)$
   **from** R
   **where** A $>$ 5
4. $\sigma_{A>5}(\sigma_{B=4}(R))$
   **from** R
   **where** A $>$ 5 **and** B $=$ 4
5. $\sigma_{A=X}(R \times S)$
   **from** R, S
   **where** A $=$ X
6. $\sigma_{A>5}(R) \times \sigma_{X=7}(S)$
   **from** R, S
   **where** A $>$ 5 **and** X $=$ 7





## Der group Teil

- Der **group** Teil partitioniert eine Tabelle in nicht-überlappende Teilmengen von Tupeln (=Gruppen).
- Input: Der **group** Teil nimmt die Tabelle, die der **where** Teil produziert hat und berechnet darauf die Gruppen.
- Konzeptionell gibt **group** mehrere Tabellen zurück.
- Beispiel: Konten gruppiert nach Filialen.

  **from** *Konten*
  **group by** *FiName*

Konten

| *KoNr* | *FiName* | *Guthaben* |
|---|---|---|
| A-101 | Chur | 500 |
| A-215 | Brugg | 700 |
| A-102 | Brugg | 400 |
| A-305 | Brugg | 350 |
| A-222 | Brugg | 700 |
| A-201 | Aarau | 900 |
| A-217 | Aarau | 750 |





## Der having Teil/1

- Input: Der **having** Teil nimmt eine gruppierte Tabelle und berechnet eine neue gruppierte Tabelle (mit ausgewählten Gruppen).
- Die **having** Bedingung wird auf jede Gruppe angewandt; nur jene Gruppen, welche die Bedingung erfüllen werden zurückgegeben.
- Die **having** Bedingung kann sich nur auf gruppierte oder aggregierte Attribute beziehen (weil die Bedingung entweder alle oder kein Tupel einer Gruppe auswählen muss).
- Alles oder nichts: Der **having** Teil gibt nie individuelle Tupel einer Gruppe zurück (entweder die gesamte Gruppe oder nichts).





## Der having Teil/2

- Filialen mit mehr als einem Konto:

  **from** *Konten*
  **group by** *FiName*
  **having count**(KoNr) $>$ 1

- Dieser **having** Teil gibt alle Gruppen mit mehr als einem Tupel zurück:

Konten

| *KoNr* | *FiName* | *Guthaben* |
|---|---|---|
| A-215 | Brugg | 700 |
| A-102 | Brugg | 400 |
| A-305 | Brugg | 350 |
| A-222 | Brugg | 700 |
| A-201 | Aarau | 900 |
| A-217 | Aarau | 750 |

## Integrierte Übung 4.2

- Welche der folgenden SQL Fragmente sind korrekt?

**from** Konten
**group by** FiName
**having** Guthaben $<$ 730
falsch

**from** Konten
**group by** FiName
**having** FiName = 'Chur'
**or** FiName = 'Aarau'
korrekt

**from** Konten
**group by** FiName
**having sum**(Guthaben) $<$ 1000
korrekt

Konten

| *KoNr* | *FiName* | *Guthaben* |
|---|---|---|
| A-101 | Chur | 500 |
| A-215 | Brugg | 700 |
| A-102 | Brugg | 400 |
| A-305 | Brugg | 350 |
| A-222 | Brugg | 700 |
| A-201 | Aarau | 900 |
| A-217 | Aarau | 750 |





## Der select Teil/1

- Der **select** Teil spezifiziert die Spalten, die im Resultat vorkommen sollen.
- Entspricht der Projektion in der relationalen Algebra.
- Beispiel: Namen aller Kunden:

**select** *KuName*
**from** *Kunden*

- Äquivalente Anfrage in relationaler Algebra (Beachte: *KuName* ist Primärschlüssel und hat keine Duplikate):

$\pi_{KuName}(Kunden)$





## Der select Teil/2

- SQL erlaubt Duplikate in Tabellen und im Resultat einer Anfrage.
- Duplikate können in SQL durch **distinct** eliminiert werden.
- Beispiel: Die Namen aller Filialen, die Kredite vergeben:
  - SQL:
    1. **select** *FiName* **from** *Kredite*
    2. **select distinct** *FiName* **from** *Kredite*
  - Relationale Algebra:

$\pi_{FiName}(Kredite)$

- SQL 1 ist *nicht* äquivalent zu $\pi_{FiName}(Kredite)$:
  - durch die Projektion entstehen Duplikate (mehrere Tupel von *Kredite* können denselben Wert für *FiName* haben)
  - relationale Algebra: die Duplikate im Ergebnis werden eliminiert
  - SQL: Duplikate werden nicht eliminiert
- SQL 2 ist äquivalent zu $\pi_{FiName}(Kredite)$:
  - **select distinct** eliminiert Duplikate im Ergebnis





## Der select Teil/3

- Im **select** Teil können Aggregationsfunktionen verwendet werden:
  - **avg**: Durchschnittswert
  - **min**: kleinster Wert
  - **max**: grösster Wert
  - **sum**: Summe aller Werte
  - **count**: Anzahl Werte
- Die Aggregatfunktionen verarbeiten alle Zeilen einer Gruppe und berechnen einen aggregierten Wert für diese Gruppe.
- Falls es einen **group** Teil gibt, dürfen im **select** Teil nur folgende Attribute vorkommen:
  - gruppierte Attribute: kommen im **group** Teil vor
  - aggregierte Attribute: beliebiges Attribut als Argument einer Aggregatfunktion
- Falls der **group** Teil fehlt, bildet die gesamte Tabelle die einzige Gruppe.

## Der select Teil/4

- Der Stern * im **select** Teil bedeutet "alle Spalten"

  **select** *
  **from** *Kredite*

- **count**(*) berechnet die Anzahl der Tupel pro Gruppe
  - **count**($*$) zählt Tupel, auch wenn diese nur *null*-Werten speichern
  - **count**($A$) zählt nur Attributwerte von $A$, die *nicht null* sind
- Beispiel:

*R*

| *A* |
|---|
| 3 |
| 3 |
| *null* |

**select ***
**from** *R*

| *A* |
|---|
| 3 |
| 3 |
| *null* |

**select count(*)**
**from** *R*

| *count* |
|---|
| 3 |

**select count(A)**
**from** *R*

| *count* |
|---|
| 2 |





## The select Teil/5

- Das durchschnittliche Guthaben auf den Konten der Brugg Filiale.

  **select avg**(*Guthaben*)
  **from** *Konten*
  **where** *FiName* = 'Brugg'

- Anzahl der Tupel in der Kunden Tabelle.

  **select count**($*$)
  **from** *Kunden*

- Die Anzahl der Konten pro Filiale.

  **select count**(KoNr), FiName
  **from** *Konten*
  **group by** *FiName*





## Integrierte Übung 4.3

Formulieren Sie die folgenden Anfragen in SQL:

Filialen[FiName, Stadt, Umsatz]
Kunden[KuName, Strasse, Ort]
Konten[KoNr, FiName, Guthaben]
Kredite[KrNr, FiName, Betrag]
Kontoinhaber[KuName, KoNr]
Kreditnehmer[KuName, KrNo]

1. Die Nummern jener Kredite, die grösser als 1200 sind

   **select** KrNr
   **from** Kredite
   **where** Betrag $>$ 1200

2. Die Namen aller Kunden, die einen Kredit bei der Brugg Filiale haben

   **select distinct** KuName
   **from** Kredite, Kreditnehmer
   **where** KrNo = KrNr **and** FiName = 'Brugg'





## Integrierte Übung 4.4

Formulieren Sie die folgenden Anfragen in SQL:

Filialen[FiName, Stadt, Umsatz]
Kunden[KuName, Strasse, Ort]
Konten[KoNr, FiName, Guthaben]
Kredite[KrNr, FiName, Betrag]
Kontoinhaber[KuName, KoNr]
Kreditnehmer[KuName, KrNo]

1. Von jeder Filiale das grösste Guthaben.

   **select** FiName, max(Guthaben)
   **from** Konten
   **group by** FiName

2. Von jeder Filiale das grösste und kleinste Guthaben.

   **select** FiName, max(Guthaben), min(Guthaben)
   **from** Konten
   **group by** FiName

Konten

| *KoNr* | *FiName* | *Guthaben* |
|---|---|---|
| A-101 | Chur | 500 |
| A-215 | Brugg | 700 |
| A-102 | Brugg | 400 |
| A-305 | Brugg | 350 |
| A-222 | Brugg | 700 |
| A-201 | Aarau | 900 |
| A-217 | Aarau | 750 |

## Anfrageausdruck/1

- Die Mengenoperationen **union**, **intersect**, und **except** entsprechen den relationalen Operatoren $\cup$, $\cap$, $-$
- Keine Duplikate: Jeder der Operatoren wird auf Tabellen ohne Duplikate angewandt und gibt ein Resultat ohne Duplikate zurück.
- Um Duplikate zu bewahren werden erweiterte Mengenoperationen verwendet: **union all**, **intersect all**, und **except all**.

  Annahme: ein Tupel kommt $m$ mal in $R$ und $n$ mal in $S$ vor. In diesem Fall kommt das Tupel:
  - $m + n$ mal in $R$ **union all** $S$ vor
  - $min(m, n)$ mal in $R$ **intersect all** $S$ vor
  - $max(0, m - n)$ mal in $R$ **except all** $S$ vor
- Union compatibility:
  - Im Unterschied zur relationalen Algebra müssen die Attributnamen in den Schemata nicht übereinstimmten.
  - Die Typen der entsprechenden Spalten müssen jedoch kompatibel sein.





## Anfrageausdruck/2

- Alle Kunden die Kredite oder Konten haben:

  (**select** *KuName* **from** *Kontoinhaber*)
  **union**
  (**select** *KuName* **from** *Kreditnehmer*)

- Kunden die sowohl einen Kredite wie auch ein Konto haben:

  (**select** *KuName* **from** *Kontoinhaber*)
  **intersect**
  (**select** *KuName* **from** *Kreditnehmer*)

- Kunden die ein Konto aber keinen Kredit haben:

  (**select** *KuName* **from** *Kontoinhaber*)
  **except**
  (**select** *KuName* **from** *Kreditnehmer*)





## Notation

- Um Namenskonflikte aufzulösen können qualifizierte Bezeichner verwendet werden:
  - T.C anstatt C
  - T.C bedeutet Spalte C aus Tabelle T
- Tabellen (und Spalten) können mit **as** umbenannt werden:
  - **from** Kunden **as** K
  - **select** max(Lohn) **as** GroessterLohn
- Eigenheiten realer Systeme:
  - In MySQL und PostgreSQL kann **as** in **from** und **select** Teil weggelassen werden
  - In Oracle muss **as** im **from** Teil weggelassen werden und kann im **select** weggelassen werden
  - Oracle verwendet MINUS statt EXCEPT für Mengendifferenz.
  - In MySQL existiert keine Mengendifferenz (EXCEPT) und kein Mengendurchschnitt (INTERSECT).





## Integrierte Übung 4.5

- Formulieren Sie folgende Anfrage in SQL:

  Bestimmen Sie das grösste Guthaben von Filialen, die ein Konto mit einem Guthaben von mehr als 500 haben.

```
select max(Guthaben), FiName
from Konten
group by FiName
having max(Guthaben) > 500
```

Filialen[FiName, Stadt, Umsatz]
Kunden[KuName, Strasse, Ort]
Konten[KoNr, FiName, Guthaben]
Kredite[KrNr, FiName, Betrag]
Kontoinhaber[KuName, KoNr]
Kreditnehmer[KuName, KrNo]

## Integrierte Übung 4.6

- Identifizieren Sie Probleme der folgenden SQL Anfrage:

  Bestimmen Sie für jede Filiale die Konten mit dem grössten Guthaben.

  **select max**(Guthaben), KoNr, FiName
  **from** Konten
  **group by** FiName

Konten

| *KoNr* | *FiName* | *Guthaben* |
|---|---|---|
| A-101 | Chur | 500 |
| A-215 | Brugg | 700 |
| A-102 | Brugg | 400 |
| A-305 | Brugg | 350 |
| A-222 | Brugg | 700 |
| A-201 | Aarau | 900 |
| A-217 | Aarau | 750 |

KoNr im select ist nicht erlaubt.

Im select können nur aggregierte oder gruppierte Spalten verwendet werden.

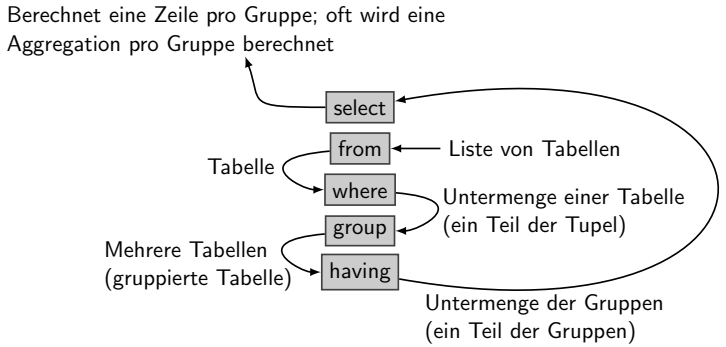
Der Grund ist, dass genau ein Resultattupel für jede Gruppe berechnet wird.





## Zusammenfassung: Grundstruktur von SQL

- Anfrageausdruck verbindet Anfragespezifikationen mit **union**,**except**,**intersect**
- Konzeptionelle Auswertung von Anfragespezifikation muss verstanden werden:








## Nullwerte/1

- Es ist möglich, dass Attribute einen Nullwert *null* haben.
- *null* steht für einen unbekannten Wert oder einen Wert der nicht existiert oder einen Wert der zurückgehalten wird oder ...
- Das Prädikat **is null** muss verwendet werden um auf Nullwerte zu prüfen.
  - Beispiel: Alle Kredite, für die der Betrag ein Nullwert ist.
    **select** *KrNr*
    **from** *Kredite*
    **where** *Betrag* **is null**
- Arithmetische Ausdrücke ergeben immer *null*, falls ein Teil *null* ist.
  - Beispiel: $5 + null$ ergibt *null*





## Nullwerte/2

- Intuition: Nullwerte sind Platzhalter für unterschiedliche Werte.

Konten (ohne Nullwerte)

| *KoNr* | *FiName* | *Guthaben* |
|---|---|---|
| A-101 | Chur | 500 |
| A-215 | Brugg | 700 |
| A-102 | Brugg | 400 |
| A-305 | Brugg | 350 |
| A-201 | Aarau | 900 |
| A-222 | Brugg | 700 |
| A-217 | Aarau | 750 |

Konten (mit Nullwerten)

| *KoNr* | *FiName* | *Guthaben* |
|---|---|---|
| A-101 | Chur | 500 |
| A-215 | null | 700 |
| A-102 | null | null |
| A-305 | Brugg | 350 |
| A-201 | null | 900 |
| A-222 | Brugg | 700 |
| A-217 | Aarau | 750 |

- Nullwerte sind also nicht als Variable mit Name *null* zu verstehen. Insbesondere ist $(null = null)$ nicht wahr.

## Nullwerte/3

- SQL verwendet dreiwertige Logik mit dem zusätzlichen Wahrheitswerte *unknown*.
- Jeder Vergleich mit *null* ergibt (den dritten logischen Wert) *unknown*
  Beispiele: $5 < null$ oder $null <> null$ oder $null = null$
- Wahrheitswerte logischer Verknüpfungen sind wie erwartet:
  - **OR** (*unknown* **or** *true*) = *true*,
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown*) = *unknown*
  - **AND** (*true* **and** *unknown*) = *unknown*,
    (*false* **and** *unknown*) = *false*
    (*unknown* **and** *unknown*) = *unknown*
  - **NOT** (**not** *unknown*) = *unknown*
- *unknown* als Ergebnis des Prädikates im **where** bzw. **having** Teil wird gleich behandelt wie *false* (d.h., Tupel bzw. Gruppe wird nicht zurückgegeben).





## Nullwerte/4

**Aggregatfunktionen:**

- Ignorieren Nullwerte in den aggregierten Attributen.
- Ausnahme: **count(*)** zählt die Anzahl der Zeilen in einer Tabelle.
- Beispiel: Die Anzahl vergebener Kredite?

  **select count**(*Betrag*)
  **from** *Kredite*

  - Die SQL Anfrage zählt keine Kredite mit einem Nullwert als Betrag.
  - Das Resultat ist 0 falls alle Kreditbeträge null sind.

**Gruppierung:**

- **group** betrachtet alle Nullwerte als wären sie identisch.
- Nullwerte in aggregierten Attributen werden als Gruppe zusammengefasst.
- Beispiel: $R[A, B, C] = \{[1, null, 100], [1, null, 200], [null, null, 100]\}$ gruppiert nach den Attributen $A$ und $B$ ergibt die Gruppen
  - $\{[1, null, 100], [1, null, 200]\}$
  - $\{[null, null, 100]\}$





## Duplikate/1

- Für Tabellen mit Duplikaten muss definiert werden, wie oft ein Tuple im Resultat einer Anfrage vorkommt (d.h. die reine Mengenlehre ist nicht mehr ausreichend).
- Beispiel:

R

| A | B |
|---|---|
| 1 | 4 |
| 1 | 2 |
| 1 | 3 |
| 1 | 3 |

S

| X |
|---|
| 1 |
| 1 |

**select** A
**from** R

| A |
|---|
| 1 |
| 1 |
| 1 |
| 1 |

**select** * **from** R, S

| A | B | X |
|---|---|---|
| 1 | 4 | 1 |
| 1 | 2 | 1 |
| 1 | 3 | 1 |
| 1 | 3 | 1 |
| 1 | 4 | 1 |
| 1 | 2 | 1 |
| 1 | 3 | 1 |
| 1 | 3 | 1 |

**select** A **from** R
**except all**
**select** X **from** S

| A |
|---|
| 1 |
| 1 |





## Duplikate/2

- Um SQL abbilden zu können, wird die relationale Algebra auf Multimengen (Mengen mit Duplikaten) erweitert.
- Beispiele:

  $\sigma_p(R)$ Für ein Tupel $t$ das $c$ mal in $R$ vorkommt gilt: Falls $t$ das Selektionsprädikat $p$ erfüllt, dann sind $c$ Kopien von $t$ in $\sigma_p(R)$, andernfalls keine.

  $\pi_A(R)$ Für jede Kopie eines Tupels $t$ von $R$ gibt es ein eine Kopie des Tupels $t.[A]$ in $\pi_A(R)$.

  $R_1 \times R_2$ Wenn es $c_1$ Kopien von $t_1$ in $R_1$ gibt und $c_2$ Kopien von $t_2$ in $R_2$, dann gibt es $c_1 * c_2$ Kopien des Tupels $t_1 \circ t_2$ in $R_1 \times R_2$.

## Duplikate/3

- SQL-Anfrage

  **select** $A_1, A_2, \ldots, A_n$
  **from** $R_1, R_2, \ldots, R_m$
  **where** $p$

  ist äquivalent zu Ausdruck der Relationalen Algebra mit Multimengen:

$$\pi_{A_1,A_2,\ldots,A_n}(\sigma_p(R_1 \times R_2 \times \ldots \times R_m))$$





## Ordnung der Tupel

- Die Zeilen einer Tabellen sind nicht geordnet.
- **order by** Teil: Das Ergebnis einer Anfrage lässt sich mit **order by** ordnen.
- Beispiel: Alphabetisch geordnete Liste aller Namen von Kunden die einen Kredit von der Brugg Filiale haben.

  **select distinct** *KuName*
  **from** *Kreditnehmer*, *Kredite*
  **where** *KrNo* = *KrNr* **and** *FiName* = 'Brugg'
  **order by** *KuName*

- Sortierung: Es ist möglich zwischen **desc** (absteigende Sortierung) oder **asc** (aufsteigende Sortierung, Default) auszuwählen.
  - Beispiel: **order by** *KuName* **desc**





## Integrierte Übung 4.7

- Erklären Sie das Resultat des folgenden SQL Befehls

  **select count**(*) **as** Cnt1,
  **count**(Umsatz) **as** Cnt2
  **from** Filiale

| Cnt1 | Cnt2 |
|---|---|
| 123 | 87 |

  Die Tabelle Filiale hat 123 Zeilen. Von 87 dieser Filialen ist der Umsatz bekannt. Die anderen Umsätze sind Nullwerte.





## Integrierte Übung 4.8

1. Was macht folgende Anfrage?
   **select** * **from** *PC* **where** *SpeedGHz* > 1 **or** *SpeedGHz* < 4
   Wie könnte eine äquivalente, bessere Anfrage lauten.

   Äquivalente Anfrage:
   **select** * **from** *PC* **where** *SpeedGHz* **is not null**
2. Was ergibt folgende Anfrage?
   **select** * **from** *R* **where** *X* <> *null*

   Eine leere Tabelle mit dem Schema von *R*.
3. Was ergibt folgende Anfrage für eine Tabelle $R[X]$?
   **select** * **from** *R* **group by** *X*

   Die Tabelle *R* ohne Duplikate.

## Zusammenfassung: Nullwerte, Duplikate, Ordnung

- Nullwerte: Wert nicht vorhanden.
  - Platzhalter für unterschiedliche Werte
  - dreiwertige Logik mit *unknown*
  - Aggregatfunktionen ignorieren Nullwerte (außer **count**(∗))
- Duplikate:
  - SQL erlaubt Duplikate
  - relationale Algebra für Multimengen erforderlich
- Ordnung:
  - Tupel in Tabelle sind nicht sortiert
  - Ergebnis einer Anfrage kann mit **order by** sortiert werden





## Inhalt







## Geschachtelte Anfragen/1

- In SQL können **select**-Anweisungen auf vielfältige Weisen verknüpft und geschachtelt werden.
- Eine Unteranfrage ist ein Anfrageausdruck der innerhalb einer anderen Anfrage geschachtelt ist.
- Eine typische Verwendung von Unteranfragen sind Tests auf Mengenzugehörigkeit, Mengenvergleiche und Kardinalitäten von Mengen.





## Geschachtelte Anfragen/3

- Unterabfragen im **where** Teil können folgende Konstrukte verwenden:
  - **exists**, **not exists**
  - **in**, **not in**
  - $=$ **some**, $<$ **some**, $<>$ **some** usw.
    **any** ist ein Synonym für **some**
  - $=$ **all**, $<$ **all**, $<>$ **all** usw.
- Beispiele:
  - **select** * **from** Kr **where** KrNr **in** (**select** KrNo **from** KrNe)
  - **select** * **from** Kr **where** KrNr $=$ **some** (**select** KrNo **from** KrNe)
  - **select** * **from** Kr **where** KrNr $<>$ **all** (**select** KrNo **from** KrNe)
- Außerdem können Unterabfragen nur über einen Operator knüpft sein.
  - In diesem Fall darf die Unterabfrage nur eine einige Zeile zurückliefern.
  - Typischerweise berechnen solche Unterabfragen eine Aggregatfunktion.
  - Beispiel:
    **select** * **from** Kr **where** Bertrag $=$ (**select** **avg**(Betrag) **from** KrNe)

## Anfragen mit EXISTS

- Die **exists** (und **not exists**) Unterabfragen werden oft verwendet. **exists** ist erfüllt falls die Unterabfrage nicht leer ist.
  - **exists** $(q) \Leftrightarrow q \neq \emptyset$
  - **not exists** $(q) \Leftrightarrow q = \emptyset$
- Bsp: Kontoinhaber die auch Kreditnehmer sind?
  - **select** KuName
    **from** Kontoinhaber **as** KI
    **where exists** (**select** *
    **from** Kreditnehmer **as** KN
    **where** KI.KuName = KN.KuName )
- Bsp: Kontoinhaber die nicht Kreditnehmer sind?
  - **select** KuName
    **from** Kontoinhaber **as** KI
    **where not exists** (**select** *
    **from** Kreditnehmer **as** KN
    **where** KI.KuName = KN.KuName)





## Anfragen mit IN

- $a$ **in** $(R)$
  - $a$ ist ein Ausdruck, z.B. ein Attributname oder eine Konstante
  - $R$ ist eine Anfrage und liefert gleich viele Spalten zurück wie der Ausdruck $a$ (eine Spalte, falls $a$ ein Attributname)
  - ist wahr, falls mindestens ein Ergebnistupel von $R$ gleich $a$ ist
- $a$ **not in** $(R)$
  - ist wahr, falls kein Ergebnistupel von $R$ gleich mit $a$ ist





## Beispiele: Anfragen mit IN

- Alle Kunden die sowohl ein Konto als auch einen Kredit haben.

  **select** *KuName*
  **from** *Kreditnehmer*
  **where** *KuName* **in** (**select** *KuName*
  **from** *Kontoinhaber*)

  Bestimmt alle Zeilen in der Tabelle *Kreditnehmer* deren Kundennanme auch in der Tabelle *Kontoinhaber* vorkommt

- Alle Kunden die einen Kredit aber kein Konto haben.

  **select** *KuName*
  **from** *Kreditnehmer*
  **where** *KuName* **not in** (**select** *KuName*
  **from** *Kontoinhaber*)





## Anfragen mit SOME

- $a$ <comp> **some** $(R) \Leftrightarrow \exists t \in R(a$ <comp> $t)$
  wobei <comp> eines der folgenden Prädikate sein kann:
  $<, \leq, \geq, >, =, \neq$
- Beispiele:

  $(5 <$ **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}$ $) =$ true (Bedeutung: $5 <$ ein Tupel in der Tabelle)

  $(5 <$ **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$ $) =$ false

  $(5 =$ **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$ $) =$ true

  $(5 \neq$ **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$ $) =$ true (weil $0 \neq 5$)

## Beispiel: Anfragen mit SOME

- Alle Filialen deren Umsatz grösser ist als derjenige der Filiale Aarau.

```
select T.FiName
from Filiale as T, Filiale as S
where T.Umsatz > S.Umsatz and S.FiName = 'Aarau'
```

- Die gleiche Anfrage wie oben aber mit $>$ **some** Konstrukt

```
select FiName
from Filiale
where Umsatz > some (select Umsatz
                     from Filiale
                     where FiName = 'Aarau')
```





## SOME vs. IN/1

- $=$ **some** und **in** sind äquivalent.
- Beispiel: Kontoinhaber die auch Kreditnehmer sind?

```
select KuName
from Kontoinhaber as KI
where KI.KuName in (select KN.KuName
                    from Kreditnehmer KN)
```

```
select KuName
from Kontoinhaber as KI
where KI.KuName = some (select KN.KuName
                        from Kreditnehmer KN)
```





## SOME vs. IN/2

- $\neq$ **some** und **not in** sind nicht äquivalent.
- Beispiel: Kontoinhaber die nicht Kreditnehmer sind?

**Richtig:**

```
select KuName
from Kontoinhaber as KI
where KI.KuName not in (select KN.KuName
                        from Kreditnehmer KN)
```

**Falsch:**

```
select KuName
from Kontoinhaber as KI
where KI.KuName <> some (select KN.KuName
                         from Kreditnehmer KN)
```





## Anfragen mit ALL

- $a$ <comp> **all** $(R) \Leftrightarrow \forall t \in R\ (a$ <comp> $t)$

$(5 < \textbf{all}\ \boxed{\begin{array}{c}0\\5\\6\end{array}}) = \text{false}$

$(5 < \textbf{all}\ \boxed{\begin{array}{c}6\\10\end{array}}) = \text{true}$

$(5 = \textbf{all}\ \boxed{\begin{array}{c}4\\5\end{array}}) = \text{false}$

$(5 \neq \textbf{all}\ \boxed{\begin{array}{c}4\\6\end{array}}) = \text{true}$ (since $5 \neq 4$ and $5 \neq 6$)

($\neq$ **all**) $\equiv$ **not in**
Aber: ($=$ **all**) $\not\equiv$ **in**

## Beispiel: Anfragen mit ALL

- Die Namen aller Filialen die ein grösseres Guthaben als alle Banken in Aarau haben.

```
select FiName
from Filiale
where Umsatz > all
                (select Umsatz
                 from Filiale
                 where FiOrt = 'Aarau')
```





## EXISTS statt SOME/ANY, IN, ALL

- Die Konstrukte **in**, **all**, **any**, **some** können unübersichtlich und schwer zu interpretieren werden.
- Beispiel: Nullwerte und leere Mengen müssen berücksichtigt werden.

$(5 < \textbf{all} \begin{array}{|c|}\hline 4 \\ \hline \text{null} \\ \hline \end{array}) = \text{false}$

$(5 > \textbf{all} \begin{array}{|c|}\hline 4 \\ \hline \text{null} \\ \hline \end{array}) = \text{unknown}$

$(5 <> \textbf{all}\ \emptyset) = \text{true}$

- Mithilfe von **exists** können alle Anfragen ausgedrückt werden, die **in**, **all**, **any**, **some** verwenden.
- Manche Implementierung schreiben alle geschachtelten Anfragen in **exists**-Anfragen um.





## Abgeleitete Tabellen

- SQL erlaubt eine Unterabfrage im **from** Teil (anstelle eines Tabellennamens kann eine SQL Anfrage verwendet werden).
- Das ist wichtig für die Geschlossenheit einer Sprache.
- Eine abgeleitete Tabelle wird durch einen Anfrageausdruck definiert.
- Den durchschnittlichen Kontostand von Filialen die einen durchschnittlichen Kontostand von mehr als $1200 haben.

```
select FiName, AvgGuthaben
from FilialeAvg
where AvgGuthaben > 1200
```

```
select FiName, AvgGuthaben
from (select FiName, avg(Guthaben) as AvgGuthaben
      from Konten
      group by FiName) as FilialeAvg
where AvgGuthaben > 1200
```





## Integrierte Übung 4.9

- Gegeben ist Tabelle $R$ wie folgt:

R

| A |
|---|
| 1 |
| 2 |
| 3 |

Geben Sie einen SQL Befehl der den grössten Wert in $R$ bestimmt. Der SQL Befehl soll ohne Aggregatfunktionen auskommen.

```
select A
from R as R1
where not exists ( select *
                   from R as R2
                   where R2.A > R1.A )
```

## Integrierte Übung 4.10

- Sind die folgenden SQL Befehle äquivalent?
  - **select** A **from** R, S **where** A = X
  - **select** A **from** R **where** A **in** ( **select** X **from** S )

Nein. Die Anzahl der Duplikate ist unterschiedlich. Beispiel:

R

| A |
|---|
| 1 |
| 1 |

S

| X |
|---|
| 1 |
| 1 |





## Zusammenfassung

- Geschachtelte Anfragen sind Anfragen mit Unteranfragen.
- Unterabfragen im **where** Teil können folgende Konstrukte verwenden:
  - (**not**) **exists**
  - (**not**) **in**
  - **some**/**any**
  - **all**
- Alle Unteranfragen können mit (**not**) **exists** ausgedrückt werden (empfohlen).
- Eine abgeleitete Tabellen ist eine Anfragen im **from** Teil.





## Inhalt







## Löschen von Tupeln

- Löschen aller Konten der Brugg Filiale.

```
delete from Konten
where FName = 'Brugg'
```

- Löschen aller Kredite zu denen kein Kreditnehmer erfasst ist.

```
delete from Kredite
where KrNr not in ( select KrNo
                    from Kreditnehmer )
```

- **delete** vs. **drop**:
  - "**delete from** Konten" löscht alle Zeilen der Tabelle *Konten*, das Schema bleibt jedoch erhalten
  - "**drop table** Konten" löscht alle Zeilen, Schemadefinition, Indexes usw. der Tabelle *Konten*

## Einfügen von Tupeln/1

- Neues Tupel zur Tabelle *Konten[KoNr, FiName, Guthaben]* hinzufügen:

  **insert into** *Konten*
  **values** ('A-9732', 'Brugg', 1200)

- Ein neues Tupel zur Tabelle Konten hinzufügen. Das Guthaben soll **null** sein.

  **insert into** *Konten*
  **values** ('A-9732', 'Brugg', null)





## Einfügen von Tupeln/2

- Nur die ersten zwei Werte werden gesetzt:

  **insert into** *Konten*
  **values** ('A-9732', 'Brugg')

- Nicht angegebene Werte sind *null* bzw. erhalten den Wert, der mit **default** festgelegt wurde.
- Ein Wert in der Mitte wird ausgelassen:

  **insert into** *Konten*(*KoNr*, *Guthaben*)
  **values** ('A-9732', 1200)

  Äquivalente Anfrage (falls *FiName* keinen Default-Wert hat):

  **insert into** *Konten*
  **values** ('A-9732', null, 1200)





## Einfügen von Tupeln/3

- Außer konstanten Tupeln kann auch das Ergebnis einer Anfrage eingefügt werde.
- Beispiel: Füge *Kredite* als Konten mit negativem Guthaben in die Tabelle *Konten* ein:

  **insert into** *Konten*
  **select** *KrNr*, *FiName*, $-$*Betrag* from *Kredite*





## Ändern von Tupeln

- Die Guthaben aller Konten mit Guthaben über $10,000 um 6% erhöhen. Die Guthaben aller anderen Konten um 5% erhöhen.
  - Kann mit Hilfe zweier **update** Befehle erreicht werden:

    **update** *Konten*
    **set** *Guthaben* $=$ *Guthaben* $*$ 1.06
    **where** *Guthaben* $>$ 10000

    **update** *Konten*
    **set** *Guthaben* $=$ *Guthaben* $*$ 1.05
    **where** *Guthaben* $\leq$ 10000

- Die Ordnung dieser Befehle ist wichtig. Die umgekehrte Reihenfolge der SQL Befehle ist falsch.

## Zusammenfassung

- Die Daten einer Tabelle können durch folgende Befehle geändert werden:
  - **delete**: Zeilen löschen
  - **insert**: neue Zeilen einfügen
  - **update**: Werte in einer oder mehrerer Zeilen ändern





## Inhalt







## Sichten (Views)

- Nutzen von Sichten
- Erstellung und Verwendung von Sichten
- Behandlung von Sichten durch das DBMS
- Temporäre Sichten





## Nutzen von Sichten

- In manchen Fällen ist es wünschenswert, dass
  - nicht alle Benutzer das gesamte logische Modell sehen (d.h. alle Tabellen der Datenbank)
  - Benutzer auf berechnete Tabellen zugreifen können (statt auf die tatsächlich gespeicherten Basistabellen)
- Beispiel: Ein Benutzer braucht Zugang zu Kundenname, Kreditnummer und Name der Filiale, soll aber den Kreditbetrag nicht sehen. Dieser Benutzer sollte eine Relation sehen, die in SQL so ausgedrückt wird:

  **select** *KuName*, *Kredite.KrNr*, *FiName*
  **from** *Kredite*, *Kreditnehmer*
  **where** *Kreditnehmer.KrNo* = *Kredite.KrNr*

- Eine Sicht (**view**) stellt einen Mechanismus zur Verfügung um:
  - Daten vor bestimmte Benutzergruppen zu verstecken
  - Benutzern Zugang zu Ergebnissen (komplexer) Anfragen zu geben

## Erstellen von Sichten

- Eine Sicht wird durch den Befehl **create view** erstellt:

  **create view** $v(A_1, A_2, \ldots, A_n)$ **as** <Anfrageausdruck>

  - wobei $v$ der Name der Sicht ist
  - <Anfrageausdruck> ein gültiger SQL Ausdruck, der $n$ Spalten liefert
  - $A_i$ den Namen der $i$-ten Spalte festlegt
- Spaltennamen optional: Falls die Spaltennamen im Anfrageausdruck eindeutig sind und keine Funktionen enthalten, müssen keine neuen Namen angegeben werden.
- Eine Sicht ist eine virtuelle Tabelle; der Name $v$ der Sicht kann in Anfragen wie eine Tabelle verwendet werden.
- Die Sichtdefinition (Name und Anfrageausdruck) wird als Metadaten in der Datenbank gespeichert.





## Beispiel: Erstellen von Sichten

- Eine Sicht auf Filialen und deren Kunden:

```
create view Alle_Kunden as
    (select FiName, KuName
    from Kontoinhaber, Konten
    where Kontoinhaber.KoNr = Konten.KoNr)
    union
    (select FiName, KuName
    from Kreditnehmer, Kredite
    where Kreditnehmer.KrNo = Kredite.KrNr)
```

- Finde alle Kunden der Filiale 'Brugg':

```
select KuName
from Alle_Kunden
where FiName = 'Brugg'
```





## Auswertung von Sichten (View Expansion)

- Die Definition einer Sicht ist in den Metadaten definiert.
- View Expansion: Bei der Auswertung einer Anfrage, wird der Name der Sicht durch den entsprechenden Anfrageausdruck ersetzt.
- View Expansion wird durch folgenden Algorithmus beschrieben:

  **repeat**
  finde alle Sichten $v_i$ im Anfrageausdruck $e_1$
  ersetzte $v_i$ in $e_1$ durch den Anfrageausdruck von $v_i$
  **until** $e_1$ enthält keine Sichten mehr
- Für Sichten die nicht rekursiv sind terminiert dieser Algorithmus.





## Änderbarkeit von Sichten

- Eine Sicht ist änderbar (update-fähig), wenn das Datenbanksystem die Umkehrabbildung von der Sicht zu den Basistabellen herstellen kann.
- In SQL-92 sind Sichten not updatable (nicht änderbar), wenn die Anfrage in der Sichtdefinition eine der folgenden Bedingungen erfüllt:
  1. das Schlüsselwort **distinct** wird benutzt
  2. ein **group by** Teil wird benutzt
  3. ein **having** Teil wird benutzt
  4. die **select** Liste enthält Ausdrücke, die verschieden von Spaltennamen sind, oder Spaltennamen kommen mehrfach vor
  5. der **from** Teil enthält mehr als eine Sicht/Tabelle oder eine nicht änderbare Sicht
- Theoretisch könnte die Umkehrabbildung auch für Sichten erstellt werden, die laut SQL nicht änderbar sind:

  SQL änderbare Sichten $\subset$ theoretisch änderbare Sichten $\subset$ alle Sichten

## Temporäre Sichten mit **with**

- Der **with** Teil ermöglicht die Definition von temporären Sichten, welche nur innerhalb desselben Anfrageausdrucks gültig sind.
- Beispiel: Finde alle Konten mit dem maximalen Kontostand:

```
with
Max_Kontostand (Wert) as (
    select max(Guthaben)
    from Konten
)
select KoNr
from Konten, Max_Kontostand
where Konten.Guthaben = Max_Kontostand.Wert
```





## Beispiel: Temporäre Sichten mit **with**

- Finde alle Filialen, in denen das Gesamtguthaben der Konten über dem durchschnittlichen Gesamtguthaben aller Filialen liegt.

```
with
Filiale_Guthaben (FiName, Wert) as (
    select FiName, sum(Guthaben)
    from Konten
    group by FiName
),
Filiale_Guthaben_Avg (Wert) as (
    select avg(Wert)
    from Filiale_Guthaben
)
select FiName
from Filiale_Guthaben, Filiale_Guthaben_Avg
where Filiale_Guthaben.Wert > Filiale_Guthaben_Avg.Wert
```





## Integrierte Übung 4.11

Betrachten Sie folgenden DDL Befehl:

```
create view v as
  select FiName, KuName
  from Konten ko, Kontoinhaber ki
  where ko.KoNr = ki.KoNr
```

Warum wird folgender DML Befehl abgewiesen?

```
update v
set FiName = 'Brugg'
where KuName = 'Tschurtschenthaler';
```





## Inhalt

# DCL: Data Control Language

- Autorisierung und Zugriffsrechte
- Befehl **grant**
- Befehl **revoke**

Literatur:
Kemper&Eikler. Datenbanksysteme – Eine Einführung. Kapitel 12.2.





# Authorisierung und Zugriffrechte

- Autorisierung schränkt den Zugriff und die Änderung von Daten durch Benutzer ein.
- Beschränkungen können sich beziehen auf:
  - Objekte wie z.B. Schemas, Tabellen, Spalten, Zeilen
  - Ressourcen wie z.B. Zeit (CPU, Verbindung, oder Wartezeiten).
- Es gibt Zugriffsrechte auf verschiedenen Ebenen:
  - System: tablespace, ...
  - Schema: Cluster, Index, Trigger, Datenbank, ...
  - Tabellen: create, alter, index, references, drop, select, delete, insert, update, ...
  - View: create, select, delete, insert, update
  - Prozeduren: create, alter, drop
  - Typen: create, alter, drop, execute
- Zugriffsrechte können an Benuzter oder Rollen (Role Based Access Control) vergeben werden





# Der GRANT Befehl

- Der **grant** Befehl überträgt Zugriffsrechte:
  **grant** <Liste von Zugriffsrechte>
  **on** <Tabelle oder View> **to** <Liste von Benutzern>
- <Liste von Benutzer> kann sein:
  - ein Benutzername
  - eine Rolle
  - **public**: alle gültigen Benuzter
- Zugriffsrechte auf Views bewirken keine Zugriffsrechte auf die dazugehörigen Basistabellen.
- Vergeber eines Zugriffsrechtes müssen dieses selber besitzen (oder Administrator sein).





# Einige Zugriffsrechte auf Tabellen

select: Direktes Leserecht über select-Anfragen oder indirektes Leserecht über Views.
- Beispiel: **select** Zugriffsrecht für Benuzter $U_1, U_2$, and $U3$ auf Relation *Filialen* vergeben:
  **grant select on** *Filialen* **to** $U_1, U_2, U_3$

insert: erlaubt Einfügen von Zeilen mit dem **insert** Befehl

update: erlaubt Ändern von Werten mit dem **update** Befehl

delete: erlaubt Löschen von Zeilen mit dem **delete** Befehl (**drop table** ist jedoch *nicht* erlaubt!)

## Der REVOKE Befehl

- Der **revoke** Befehl nimmt Zugriffsrechte zurück.

  **revoke** <Liste von Zugriffsrechte>
  **on** <Tabelle oder View> **to** <Liste von Benutzern>

  - Beispiel:

    **revoke select on** *Filialen* **from** $U_1, U_2, U_3$

- <Liste von Zugriffsrechte> kann **all** sein, um alle Zugriffsrechte zurückzunehmen
- Falls <Liste von Benutzern> **public** enthält, verlieren alle Benutzer die angegebenen Rechte, außer die Rechte wurden explizit gewährt.
- Falls dasselbe Zugriffsrecht von zwei verschiedenen Benutzern gewährt wurde (also doppelt), kann es auch nach dem **revoke** Befehl erhalten bleiben.





## Inhalt







## Zugriff auf die Datenbank

Zugriff auf die Datenbank über Programmiersprachen:

- Embedded SQL
- Dynamic SQL
- ODBC
- JDBC





## Datenbankzugriff

- API (application program interface) für die Interaktion mit einem Datenbankserver.
- API übernimmt:
  - Verbindung zu Datenbankserver herstellen (connection)
  - SQL Befehle an den Datenbankserver schicken
  - Ergebnistupel abrufen und in Programmvariablen speichern
- **Embedded SQL**: viele Sprachen erlauben die Einbettung von SQL in den Programm Code. Embedded SQL kann sein:
  - statisch (d.h. bekannt zum Zeitpunkt der Compilierung)
  - dynamisch (d.h. Code ist zum Zeitpunkt der Compilierung nicht bekannt und wird erst zur Laufzeit erzeugt)
- **ODBC** (Open Database Connectivity) ist ein Microsoft Standard und funktioniert mit C, C++, C#, und Visual Basic
- **JDBC** (Java Database Connectivity) ist von Sun Microsystems und funktioniert mit Java

## JDBC

- JDBC ist ein Java API zur Kommunikation mit SQL Datenbanken
- JDBC unterstützt eine Vielzahl von Funktionen um Daten anzufragen, zu ändern und die Ergebnistupel einzulesen.
- JDBC unterstützt auch Anfragen auf die Metadaten, z.B. Namen und Typen von Spalten.
- Ablauf der Kommunikation mit der Datenbank:
  - Netzwerkverbindung herstellen (*Connection* Objekt)
  - *Statement* Objekt erzeugen (ist einer Connection zugeordnet)
  - das *Statement* Objekt wird benutzt, um Anfragen auszuführen und Ergebnisse auszulesen
  - Exceptions werden zur Fehlerbehandlung verwendet





## Beispiel: JDBC/1

- Wir schreiben ein Java Programm, das sich über JDBC mit PostgreSQL Datenbank verbindet.
- Zugangsdaten:
  - Hostname: `dumbo.cosy.sbg.ac.at`
  - Port: `5432`
  - Datenbankname: `ss2013`
  - Benutzername: `augsten`
  - Passwort: `xxx`
- Aufruf des Programmes

  `java -cp .:postgresql_jdbc.jar PostgresqlJDBC`

  wobei folgende Dateien im aktuellen Pfad zu finden sein müssen:
  - `PostgresqlJDBC.class`
  - `postgresql_jdbc.jar`: ein JDBC Driver für Postgresql
- Das Programm gibt die Namen aller Tabellen zurück, deren Besitzter `augsten` ist.





## Beispiel: JDBC/2

```
import java.sql.*;

public class PostgresqlJDBC  {

    public static void main(String[] args) throws  Exception {

        Class.forName("org.postgresql.Driver");
        Connection conn =
            DriverManager.getConnection(
                "jdbc:postgresql://dumbo.cosy.sbg.ac.at:5432/ss2013",
                "augsten", "xxx");

        Statement stmt = conn.createStatement();

        ResultSet rset = stmt.executeQuery(
            "select tablename from pg_tables where tableowner='augsten'");

        while (rset.next())
            System.out.println(rset.getString(1));
    }

}
```