universität wien

Kapitel 4

SQL – Structured Query Language

Inhalt

4.1 SQL Einführung

In der Vergangenheit eine Vielzahl von Sprachvorschlägen und –entwürfen für Manipulations- und Abfragesprachen

- fast jedes Datenbankprodukt verfolgte einen eigenen Ansatz
- große, (fast) unüberschaubare Menge von Sprachen

Probleme

- redundante Kosten für Schulungsaufwand
- kein standardisierter Datenaustausch

Diese Situation war der allgemeinen Verbreitung und Akzeptanz von relationalen Datenbanksystemen hinderlich

SQL: Structured Query Language

- strukturierte Sprache, die auf englischen Schlüsselwörtern basiert
- 1979 durch Oracle V2 (Relational Software Inc.) am Markt
- im anglikanischen Raum findet man auch die Aussprache /ˈsiːkwəl/

Mächtigkeit von SQL äquivalent zum Relationenkalkül und zur Relationenalgebra, relational vollständig

Ziele von SQL

Entwicklung einer **vereinheitlichten DB-Manipulationssprache** für alle Aufgaben der Datenbankverwaltung

Vereinfachter Zugang zur Sprache durch aufeinander aufbauende „Sprachebenen“ mit anwachsender Komplexität.

Dieser Ansatz führt zur gewünschten Eigenschaft der **Proportionalität**:

einfache Anfragemöglichkeiten für den gelegentlichen Benutzer

mächtige Sprachkonstrukte für den besser ausgebildeten Benutzer

Unterstützung der „**natürlich-sprachlichen**“ Formulierung einfacher Anfragen.

Vereinfachte bzw. **klar strukturierte Sprachkonstrukte** zur Erleichterung des Verständnisses und zur Reduktion von fehlerhaften Eingaben.

Sprachentwicklung von SQL

SQL wurde **„de facto“-Standard** in der relationalen Welt

- 1986 von ANSI, 1987 von ISO akzeptiert
- Spezielle Sprachkonstrukte für den DBA

SQL Standards werden stufenweise üblicherweise mit den Jahreszahlen der Veröffentlichung identifiziert

- SQL-86, SQL-89, SQL-92 (auch SQL2), SQL:1999 (auch SQL3), SQL:2003 (auch SQL4), SQL 2006, SQL 2008 und der aktuellen Standard SQL:2011

Jeder Standard erweiterte die Funktionalität der Sprache

- Führte zu einem **großem und sehr mächtigen Sprachumfang**, verminderte aber auch etwas die ursprüngliche Eleganz und Klarheit der Sprache
- Standard SQL:2006 erweitert die Sprache z.B. um **XML Daten** in der Datenbank und XQuery, die XML Abfragesprache
- Der aktuelle Standard SQL:2011 unterstützt **temporale** Datenbanken

Der SQL Standard ist frei erhältlich und sehr umfangreich, mehrere tausend Seiten Beschreibung

- Artikel + Präsentationen zu SQL http://www.wiscorp.com/SQLStandards.html

4.2 SQL Grundlagen

SQL basiert auf Mengen und relationalen Operationen mit speziellen Modifikationen und Erweiterungen

Eine typische SQL Abfrage hat die Form

SELECT A_1, A_2, ... , A_n
FROM r_1, r_2, ..., r_m
WHERE P

A_i *sind Attribute*

r_i sind Relationen

P ist ein Prädikat

Diese Abfrage ist äquivalent zum relationalen Algebra-Ausdruck

$$\Pi_{A1, A2, ..., An}(\sigma_P (r_1 \times r_2 \times ... \times r_m))$$

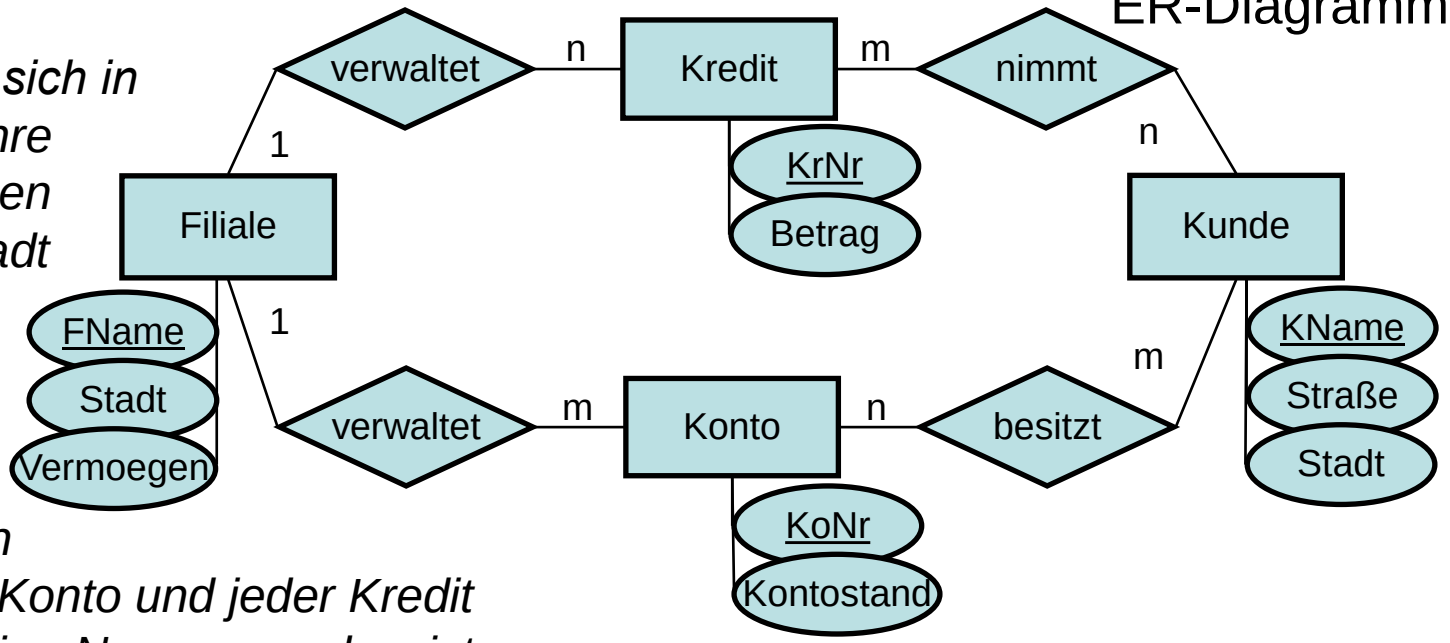
Das Ergebnis einer SQL Abfrage ist eine Relation

„Bank“-Schema für die Beispiele

Bank-Szenario:

„Eine Bank gliedert sich in Filialen, die durch ihre Namen unterschieden werden, in einer Stadt beheimatet sind und ein Vermögen ausweisen. Jede Filiale verwaltet ihre eigenen Konten und Kredite. Jedes Konto und jeder Kredit besitzt eine eindeutige Nummer und weist einen Kontostand (bei Konto) bzw. einen Betrag (bei Krediten) auf. Kunden besitzen Konten und nehmen Kredite, wobei Konten und Kredite einem oder mehreren Kunden zugeordnet sind. Kunden haben einen eindeutigen Namen und wohnen in einer Straße in einer Stadt.“

ER-Diagramm



DB-Schema „Bank“:

filiale (<u>fname</u>, stadt, vermoegen)

konto (<u>konr</u>, kontostand, *fname*)

kunde (<u>kname</u>, straße, stadt)

kredit (<u>krnr</u>, betrag, *fname*)

kontoinhaber (*<u>kname</u>*, *<u>konr</u>*)

kreditnehmer (*<u>kname</u>*, *<u>krnr</u>*)

SELECT Teil

Der ***SELECT*** Teil listet die in der Abfrage gewünschten Attribute auf

- Entspricht der Projektions-Operation der relationalen Algebra

„Finde die Namen aller Filialen in der *kredit* Relation“

```
SELECT fname
FROM kredit
```

In der reinen relationalen Algebra entspricht dies der Abfrage

$$\Pi_{fname}(kredit)$$

Anmerkungen

- SQL erlaubt kein ’-’ Zeichen in Namen
- Bei SQL Namen wird zwischen Groß- und Kleinschreibung nicht unterschieden

SELECT Teil (2)

SQL erlaubt doppelte Werte in Relationen (Tabellen) wie auch in Abfrageergebnissen

Um die Entfernung von doppelten Werte zu erzwingen muss das Schlüsselwort ***DISTINCT*** nach ***SELECT*** angegeben werden

"Finde die Namen aller Filialen in der *kredit* Relation und entferne alle Doppelten"

```
SELECT DISTINCT fname
FROM kredit
```

Das Schlüsselwort ***ALL*** gibt an, dass doppelte Werte nicht entfernt werden (Default wenn nichts angegeben ist)

```
SELECT ALL fname
FROM kredit
```

SELECT Teil (3)

Ein Stern im ***SELECT*** Teil spezifiziert “alle Attribute”

```
SELECT *
FROM kredit
```

Der ***SELECT*** Teil kann arithmetische Ausdrücke mit den Operatoren +, -, *, / und Konstanten und Tupelattributen enthalten

Die Abfrage

```
SELECT krnr, fname, betrag*100 FROM kredit
```

liefert eine Relation identisch zu *kredit*, außer dass das Attribut *betrag* mit 100 multipliziert wurde

WHERE Teil

Der ***WHERE*** Teil gibt Bedingungen an, die das Ergebnis erfüllen muss (entspricht dem Selektion Prädikat der relationalen Algebra)

"Finde alle Kreditnummern für Kredite der Perryridge Filiale deren Betrag größer als $1200 ist"

```
SELECT krnr
FROM kredit
WHERE fname = 'Perryridge' AND betrag > 1200
```

- Vergleichsbedingungen können mit den logischen Ausdrücke ***AND***, ***OR*** und ***NOT*** kombiniert werden
- Vergleichsbedingungen können auf Ergebnisse arithmetischer Ausdrücke angewendet werden

SQL kann auch einen ***BETWEEN*** Vergleichsoperator enthalten

"Finde die Kreditnummern aller Kredite deren Betrag zwischen $90,000 und $100,000 liegt (≥ $90,000 und ≤ $100,000)"

```
SELECT krnr
FROM kredit
WHERE betrag BETWEEN 90000 AND 100000
```

FROM Teil

Der ***FROM*** Teil listet alle Relationen auf, die in der Abfrage involviert sind

Entspricht dem Kartesischen Produkt der relationalen Algebra

Erzeuge das Kartesische Produkt *kreditnehmer* x *kredit*

```
SELECT *
FROM kreditnehmer, kredit
```

"Finde die Namen, Kreditnummern und Beträge aller Kunden mit einem Kredit bei der Perryridge Filiale"

```
SELECT kname, kreditnehmer.krnr, betrag
FROM kreditnehmer, kredit
WHERE kreditnehmer.krnr = kredit.krnr AND
      fname = 'Perryridge'
```

Umbenennen

SQL erlaubt das Umbenennen von Relationen und Attributen mit dem Schlüsselwort **AS** (**`alter-name AS neuer-name`**)

"Finde die Namen, Kreditnummern und Beträge aller Kunden; benenne den Spaltennamen *krnr* um in *kredit_id*"

```
SELECT kname, kreditnehmer.krnr AS kredit_id, betrag
FROM kreditnehmer, kredit
WHERE kreditnehmer.krnr = kredit.krnr
```

Tupelvariable werden im **FROM** Teil durch **AS** definiert

„Finde die Kundennamen, deren Kreditnummern und die Beträge für alle Kunden mit einem Kredit "

```
SELECT kname, T.krnr, S.betrag
FROM kreditnehmer AS T, kredit AS S
WHERE  T.krnr = S.krnr
```

„Finde alle Filialen, die größere Vermögen besitzen als Filialen in Brooklyn."

```
SELECT DISTINCT T.fname
FROM filiale AS T, filiale AS S
WHERE T.vermoegen > S.vermoegen AND
      S.stadt = 'Brooklyn'
```

String Operationen

SQL enthält Stringvergleichsoperatoren, Muster werden durch 2 spezielle Zeichen beschrieben

- Prozent (**%**). Das % Zeichen repräsentiert eine beliebige Zeichenkette
- Underscore (_). Das _ Zeichen steht für ein beliebiges Zeichen

LIKE erlaubt das Suchen nach Pattern in Strings

"Finde die Namen aller Kunden deren Straßenadresse das Wort ‚Haupt' enthält"

```
SELECT kname
FROM kunde
WHERE straße LIKE '%Haupt%'
```

Wie suche ich dann genau "Main%" ?

```
LIKE 'Main\%' ESCAPE '\'
```

SQL unterstützt eine Reihe von Stringoperationen wie

- Zusammenhängen (durch "||")
- Umwandeln von Groß- in Kleinbuchstaben und umgekehrt
- Bestimmen der Stringlänge, Teilstrings extrahieren, usw.

Reihenfolge der Ausgabetupeln

„Liste in alphabetischer Reihenfolge die Namen der Kunden mit einem Kredit in der Perryridge Filiale“

```
SELECT DISTINCT kname
FROM kreditnehmer, kredit
WHERE kreditnehmer.krnr = kredit.krnr AND
      fname = 'Perryridge'
ORDER BY kname
```

DESC spezifiziert absteigende und ***ASC*** aufsteigende (ist Default Wert) Reihenfolge

z.B. `ORDER BY kname DESC`

4.3 Data Definition and Manipulation Language

Die **Data Definition Language** (**DDL**) erlaubt nicht nur die Spezifikation einer Menge von Relationen(schemen), sondern liefert auch Informationen über jede einzelne Relation, d.h.

- Das Schema für jede Relation
- Den Wertebereich für jedes Attribut
- Integritätsbedingungen
- Die Indizes für jede Relation
- Sicherheits- und Autorisierungsinformation für jede Relation
- Die physische Speicherstruktur jeder Relation auf der Platte

Mit der Data Manipulation Language (DML) lassen sich die gespeicherten Daten in der Datenbank modifizieren

- Einfügen, Ändern, Löschen

Datentypen in SQL

`char(n)`	String mit Benutzer-spezifizierter fixer Länge
`varchar(n)`	String mit variabler Länge mit Benutzer-spezifizierter maximaler Länge
`int`	Integer (eine endliche maschinenabhängige Menge Ganzzahlen)
`smallint`	Small integer (eine maschinenspezifische Untermenge von int)
`numeric(p,d)`	Fix-Komma Zahlen, mit benutzerdefinierter Genauigkeit von p Ziffern und d Ziffern rechts vom Dezimalpunkt
`real, double precision`	Gleitkomma und doppeltgenaue Gleitkommazahlen (maschinenspezifische Genauigkeit)
`float(n)`	Gleitkommazahlen mit benutzerspez. Genauigkeit von zumindest n Ziffern
`date`	Datum, enthält ein (4 ziffriges) Jahr, Monat und Tag, **`to_char(d, fmt)`** wandelt Datum d gemäß Formatmodell in String um, **`to_date(c, fmt)`** wandelt String c gemäß Formatmodell in Datum um
`time`	Tageszeit in Stunden, Minuten und Sekunden
`timestamp`	Datum plus Tageszeit
`interval`	Zeitperiode

NULL Werte sind in allen Datentypen erlaubt. Deklaration eines Attributs ***NOT NULL*** verhindert NULL Werte für dieses Attribut

Relationen erzeugen

Eine SQL Relation wird durch **CREATE TABLE** erzeugt

CREATE TABLE r (A_1 D_1, A_2 D_2, ..., A_n D_n,
(Integritätsbedingung$_1$),
...,
(Integritätsbedingung$_k$))

r ist der Name der Relation

jedes A_i ist ein Attribut Name im Schema der Relation r

D_i ist der Datentyp der Werte von Attribut A_i

Integritätsbedingungen können auch direkt beim Attribut angegeben werden

```
CREATE TABLE filiale(
   fname      char(15) NOT NULL,
   stadt      char(30),
   vermoegen  integer
)
```

Integritätsbedingungen in CREATE TABLE

universität wien

Attributwert ungleich NULL	**NOT NULL**
Attribut ist Primärschlüssel	**PRIMARY KEY** (A1, ..., An)
Attribut muss Prädikat P erfüllen	**CHECK** (P)

"Deklariere *fname* als Primärschlüssel für *filiale* und gewährleiste, dass *vermoegen* nicht negativ werden kann"

```
CREATE TABLE filiale (
    fname      char(15),
    stadt      char(30),
    vermoegen  integer,
    PRIMARY KEY (fname),
    CHECK (vermoegen >= 0)
)
```

PRIMARY KEY Deklaration auf einem Attribut garantiert automatisch NOT NULL in SQL-92 und danach, muss aber in SQL-89 explizit angegeben werden

Relationen ändern und löschen

Die ***ALTER TABLE*** r ***ADD*** A D Anweisung wird verwendet, um Attribute zu existierenden Relationen hinzuzufügen

A ist der Name des hinzuzufügenden Attributs und D der Datentyp von A

```
ALTER TABLE konto ADD Eroeffnung date
```

Alle Tupel in der Relation bekommen NULL als Wert für das neue Attribut zugewiesen

Die ***ALTER TABLE*** r ***DROP*** A Anweisung kann auch zum Löschen eines Attributs verwendet werden

Dabei ist A der Name des Attributs der Relation r

```
ALTER TABLE konto DROP fname
```

Löschen von Attributen wird nicht von allen DBMS unterstützt

Die ***DROP TABLE*** Anweisung löscht alle Informationen über diese Relation aus dem DBMS

```
DROP TABLE konto
```

Modifikationen in der Datenbank

Daten können in Relationen eingefügt, Attributwerte von Tupeln gelöscht und geändert werden

Die grundlegenden Formen dieser Operationen sind für

Einfügen

INSERT INTO *r* $(A_1, \ldots, A_n)$ ***VALUES*** $(V_1, \ldots, V_2)$

Fügt ein neues Tupel $(V_1, \ldots, V_2)$ in eine Relation r ein

Ändern

UPDATE *r* ***SET*** *A*=<expr> ***WHERE*** <condition>

Der Wert des Attributes A wird für alle Tupel aus der Relation r, die die Bedingung <condition> erfüllen, auf den Wert des Ausdrucks <expr> gesetzt

Löschen

DELETE FROM *r* Löscht alle Tupel aus der Relation r

DELETE FROM *r* ***WHERE*** <condition> Löscht alle Tupel aus der Relation r, die die angegebene Bedingung <condition> erfüllen.

Modifikationen in der Datenbank – Einfügen

„Füge ein neues Tupel in *konto* ein“

```
INSERT INTO konto
VALUES ('A-9732', 'Perryridge',1200)
```

oder gleichbedeutend mit Attribut-Mapping

```
INSERT INTO konto (fname, kontostand, konr)
VALUES ('Perryridge', 1200, 'A-9732')
```

„Füge ein neues Tupel in *konto* ein mit *kontostand* NULL“

```
INSERT INTO konto
VALUES ('A-777','Perryridge', NULL)
```

Einfügen

„Alle Kreditkunden der Perryridge Filiale bekommen als Geschenk ein $200 Sparkonto. Die Kreditnummer dient als neue Kontonummer“

```
INSERT INTO konto
  SELECT krnr, 200, fname
  FROM kredit
  WHERE fname = 'Perryridge'
INSERT INTO kontoinhaber
  SELECT kname, krnr
  FROM kredit, kreditnehmer
  WHERE fname = 'Perryridge' AND
        kredit.krnr = kreditnehmer.krnr
```

Das ***SELECT FROM WHERE*** Statement wird vollständig ausgewertet, bevor eines seiner Ergebnisse in die Relation eingefügt wird (andernfalls würden Abfragen wie

```
INSERT INTO table1 SELECT * FROM table1
```

Probleme verursachen

Modifikationen in der Datenbank – Ändern

Erhöhe alle Konten mit Kontoständen über $10,000 um 6%, alle anderen um 5% (Schreibe 2 ***UPDATE*** Anweisungen)

```
UPDATE konto SET kontostand = kontostand * 1.06
WHERE kontostand > 10000

UPDATE konto SET kontostand = kontostand * 1.05
WHERE kontostand <= 10000
```

Die Reihenfolge ist wichtig!

Kann besser mit einer ***CASE*** Anweisung realisiert werden

"Erhöhe alle Konten mit Kontoständen über $10,000 um 6%, alle anderen erhalten 5%."

```
UPDATE konto SET kontostand = CASE
  WHEN kontostand <= 10000 THEN
    kontostand*1.05
  ELSE
    kontostand*1.06
END
```

Modifikationen in der Datenbank – Löschen

„Lösche all Kontodaten in der Perryridge Filiale“

```
DELETE FROM konto
WHERE fname = 'Perryridge'
```

„Lösche alle Konten in jeder Filiale in der Stadt Needham“

```
DELETE FROM konto
WHERE fname IN (
   SELECT fname
   FROM filiale
   WHERE stadt = 'Needham')
DELETE FROM kontoinhaber
WHERE konr IN (
   SELECT konr
   FROM filiale, konto
   WHERE stadt = 'Needham' AND
         filiale.fname = konto.fname)
```

Reihenfolge korrekt?

Kaskadierendes Delete!

4.4 Indexstrukturen

Indexstrukturen können die Datenbankzugriffe beschleunigen

- Werden vom DBMS üblicherweise automatisch erzeugt

Datenunabhängigkeit des Relationenmodells erlaubt ein relativ beliebiges Hinzufügen und Löschen von Indizes

- Jederzeit möglich, um z. B. bei veränderten Benutzerprofilen das Leistungsverhalten zu optimieren

Index Mechanismen sind nicht Teil der aktuellen SQL Standards

Die meisten SQL Implementierungen unterstützen aber Index Manipulation

Übliche Form der Vereinbarung

CREATE INDEX <index-name> ON <Relation> (<Attributliste>)

CREATE INDEX - Beispiele

Erzeugung eines Index auf der Spalte *stadt* von *kunde*

```
CREATE INDEX kundeind1 ON kunde(stadt)
```

Realisierung z. B. durch B+-Baum (oder Hashing, mit verminderter Funktionalität)

Erzeugung eines weiteren Index auf der Spalte *kname* auf *kunde*

```
CREATE UNIQUE INDEX kundeind2 ON kunde(kname)
```

Durch ***UNIQUE*** wird kname als Schlüsselkandidat definiert (keine doppelten Werte erlaubt)

Es können auch mehrere Attribute gemeinsam durch einen Index unterstützt werden

```
CREATE UNIQUE INDEX kkk ON kontoinhaber (kname, konr)
```

Löschen eines Index

```
DROP INDEX kundeind1
```

4.5 Mengen Operationen

Die Mengen Operationen ***UNION***, ***INTERSECT*** und ***EXCEPT*** entsprechen den Operatoren $\cup$, $\cap$, – der relationalen Algebra

Jede dieser Operationen eliminiert automatisch alle Doppelten

- Um diese zu erhalten müssen die Multiset Versionen ***UNION ALL***, ***INTERSECT ALL*** und ***EXCEPT ALL*** verwendet werden

Annahme ein Tupel kommt *m* fach in *r* und *n* fach in *s* vor, *dann gilt*

- *m* + *n* fach in r ***UNION ALL*** s
- min(*m,n)* fach in r ***INTERSECT ALL*** s
- max(0, *m* – *n)* fach in r ***EXCEPT ALL*** s

Mengen Operationen - Beispiele

„Finde alle Kunden, die einen Kredit, ein Konto oder beides haben“

```
(SELECT kname FROM kontoinhaber)
        UNION
        (SELECT kname FROM kreditnehmer)
```

„Finde alle Kunden, die sowohl einen Kredit als auch ein Konto besitzen“

```
(SELECT kname FROM kontoinhaber)
        INTERSECT
        (SELECT kname FROM kreditnehmer)
```

„Finde alle Kunden mit einem Konto aber ohne Kredit“

```
(SELECT kname FROM kontoinhaber)
        EXCEPT
        (SELECT kname FROM kreditnehmer)
```

4.6 Aggregatsfunktionen

Diese Funktionen werden auf Multisets von Werten einer Spalte einer Relation angewendet und liefern einen Wert zurück

AVG	Durchschnittswert
MIN	Minimum Wert
MAX	Maximum Wert
SUM	Summe der Werte
COUNT	Anzahl der Werte

„Finde den durchschnittlichen Kontostand der Perryridge Filiale“

```
SELECT AVG(kontostand)
FROM konto
WHERE fname = 'Perryridge'
```

„Bestimme die Anzahl der Tupel in der Kunden Relation“

```
SELECT COUNT(*) FROM kunde
```

„Bestimme die Anzahl der Kontoinhaber“

```
SELECT COUNT(DISTINCT kname) FROM kontoinhaber
```

Aggregatsfunktion – GROUP BY

Die ***GROUP-BY***-Klausel gruppiert die Tupel auf die eine Aggregat-Funktion angewendet wird.

Attribute der ***SELECT*** Klausel, die nicht in der Aggregat Funktion vorkommen, müssen in der ***GROUP BY*** Liste auftauchen

„Finde die Anzahl der Kontoinhaber für jede Filiale“

```
SELECT fname, COUNT(DISTINCT kname)
FROM kontoinhaber, konto
WHERE kontoinhaber.konr = konto.konr
GROUP BY fname
```

Aggregatsfunktion – HAVING

Da ***WHERE*** keine Aggregatsfunktionen erlaubt, benötigt man für Angabe von Prädikaten die ***HAVING*** Klausel

„Finde die Namen aller Filialen deren durchschnittlicher Kontostand höher ist als $1200“

```
SELECT fname, AVG(kontostand)
FROM konto
GROUP BY fname
HAVING AVG(kontostand) > 1200
```

Anmerkung

Prädikate im ***HAVING*** Teil werden nach der Bildung der Gruppen angewendet, wo hingegen Prädikate im ***WHERE*** Teil vor der Gruppenbildung angewendet werden

Abhängige Aggregatfunktionen

„Lösche alle Kontoeinträge mit Kontoständen unter dem Durchschnitt der Bank“

```
DELETE FROM konto
WHERE kontostand < (
  SELECT AVG(kontostand) FROM konto
)
```

Problem: Sobald wir Tupel aus *konto* löschen, ändert sich der Durchschnitt

Lösung in SQL

1. Berechne zuerst **`AVG(kontostand)`** und suche alle zu löschenden Tupel
2. Danach lösche alle oben gefundenen Tupel (ohne den Durchschnitt erneut zu berechnen oder die Tupels wiederholt zu testen)

4.7 NULL-Werte

Tupel können NULL-Werte enthalten, bezeichnet durch ***NULL***

NULL kennzeichnet einen unbekannten oder nicht existierenden Wert

Das Prädikat ***IS NULL*** kann verwendet werden um auf NULL-Werte abzufragen

„Finde alle Kredite in der *kredit* Relation mit einem NULL-Wert für *betrag*“

```
SELECT krnr
FROM kredit
WHERE betrag IS NULL
```

Das Ergebnis eines arithmetischen Ausdrucks der *NULL* beinhaltet ist *NULL*

z.B. 5 + NULL liefert NULL

Aggregatsfunktionen ignorieren einfach NULL-en

NULL-Werte und 3-wertige Logik

Beliebige Vergleiche mit *NULL* liefern *unknown*

z.B. *5 < NULL* oder *NULL <> NULL* oder *NULL = NULL*

3-wertige Logik verwendet den Wert *unknown*

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*

"*P* ***IS UNKNOWN***" ergibt true falls das Prädikat P *unknown* liefert

Das Ergebnis eines ***WHERE*** Prädikats wird als *false* gewertet, wenn es zu *unknown* evaluiert

NULL-Werte und Aggregatsfunktionen

„Liefere die Summe aller Kreditbeträge“

```
SELECT sum (betrag)
FROM kredit
```

Das o.a. Statement ignoriert NULL-Werte

Das Ergebnis ist *NULL*, falls kein Nicht-*null* Betrag existiert

Alle Aggregatsfunktionen außer **`COUNT(*)`** ignorieren Tupel mit *NULL* Werten in den aggregierten Attributen

4.8 Geschachtelte Abfragen

SQL stellt einen Mechanismus für geschachtelte Unter-Abfragen (nested Subquery) zur Verfügung

- Eine Unter-Abfrage ist ein ***SELECT-FROM-WHERE*** Ausdruck, der innerhalb einer anderen Abfrage (verschachtelt) auftritt

Die übliche Anwendung von Unter-Abfrage ist der Test auf Mengen-Mitgliedschaft, Mengen-Vergleich und Mengen-Kardinalität

„Suche alle Kunden, die sowohl ein Konto als auch einen Kredit bei der Bank haben“

```
SELECT DISTINCT kname
FROM kreditnehmer
WHERE kname IN (SELECT kname
                FROM kontoinhaber)
```

„Suche alle Kunden, die einen Kredit, aber kein Konto bei der Bank haben“

```
SELECT DISTINCT kname
FROM kreditnehmer
WHERE kname NOT IN (SELECT kname
                    FROM kontoinhaber)
```

4.9 Views oder Sichten

Views oder Sichten liefern einen Mechanismus, spezifische Information für einen bestimmten Zweck aufzubereiten

- Information vor Benutzer verstecken
- Information zur Analyse aggregieren
- Datenanpassung an andere Software
- …

Views werden folgendermaßen erstellt

CREATE VIEW *v* ***AS*** <query expression>

wobei

- <query expression> ist ein beliebiger Ausdruck
- Der Name der View ist *v*

Vergleiche mit externem Schema!

Views - Beispiel

Wir erzeugen eine View …

"Definiere View bestehend aus den Filialen und ihren Kunden“

```
CREATE VIEW alle_kunden AS
    (SELECT fname, kname
     FROM kontoinhaber, konto
     WHERE kontoinhaber.konr = konto.konr)
   UNION
   (SELECT fname, kname
     FROM kreditnehmer, kredit
     WHERE kreditnehmer. krnr = kredit.krnr)
```

… und verwenden sie im Anschluss

„Suche alle Kunden der Perryridge Filiale“

```
SELECT kname
FROM alle_kunden
WHERE fname = 'Perryridge'
```

Ändern in einer View

Erzeuge eine View aller Kredit Daten mit verstecktem Betrag

```
CREATE VIEW filiale_kredit AS
    SELECT fname, krnr
    FROM kredit
```

Füge ein neues Tupel zu *filiale_kredit* hinzu

```
INSERT INTO filiale_kredit VALUES
  ('Perryridge', 'L-307')
```

Diese Einfügung fügt eigentlich folgendes Tupel in die *kredit* Relation ein:

('L-307', 'Perryridge', NULL)

Updates auf komplexeren Views sind schwer oder überhaupt nicht zu übersetzen und daher nicht erlaubt

Die meisten SQL Implementationen erlauben Updates nur auf einfachen Views, die ohne Aggregate definiert sind

4.10 Join - Verknüpfen von Relationen

Die Join Operation verknüpft 2 Relationen und liefert diese Verknüpfung als Ergebnis zurück

- Diese Operationen werden üblicherweise als Ausdrücke für Teilabfragen im ***FROM*** Teil verwendet

Die *Join Bedingung* definiert, welche Tupel in den 2 Relationen verknüpft werden und welche Attribute im Ergebnis aufscheinen

Der *Join Typ* gibt an, wie Tupel verarbeitet werden, zu denen es keine zu verknüpfende Tupeln in der anderen Relation gibt

Join Bedingung
NATURAL
ON<Prädikat>
USING(A1, A2, ..., An)

Join Typ
INNER JOIN
LEFT OUTER JOIN
RIGHT OUTER JOIN
FULL OUTER JOIN

Joinen von Relationen – Daten für die Beispiele

Relation *kredit*

krnr	fname	betrag
K-170	Downtown	3000,00
K-230	Redwood	4000,00
K-260	Perryridge	1700,00

Relation *kreditnehmer*

kname	krnr
Jones	K-170
Smith	K-230
Hayes	K-155

Anmerkung: Es fehlt die *kreditnehmer* Information für K-260 und die *kredit* Information für K-155

INNER JOIN

Der einfache ***INNER* JOIN** entspricht einem simplen (Theta) Join mit einer gegebenen Join Bedingung

Die Attribute des Ergebnisses bestehen aus den Attributen der linken und der rechten Relation

```
SELECT * FROM
kredit INNER JOIN kreditnehmer ON
kredit.krnr = kreditnehmer.krnr
```

krnr	fname	betrag	kname	krnr
K-170	Downtown	3000,00	Jones	K-170
K-230	Redwood	4000,00	Smith	K-230

LEFT OUTER JOIN

Der ***LEFT OUTER JOIN*** wird folgendermaßen berechnet:

1. Zuerst bestimme den ***INNER JOIN***
2. Dann füge jedes Tupel der linken Relation, das zu keinem Tupel der rechten Relation passt, zum Ergebnis hinzu, wobei für Attribute der rechten Relation NULL Werte gesetzt werden

```
SELECT * FROM
kredit LEFT OUTER JOIN kreditnehmer ON
kredit.krnr = kreditnehmer.krnr
```

krnr	fname	betrag	kname	krnr
K-170	Downtown	3000,00	Jones	K-170
K-230	Redwood	4000,00	Smith	K-230
K-260	Perryridge	1700,00	NULL	NULL

RIGHT OUTER JOIN

Wird analog zum left outer Join berechnet, nur mit vertauschten Relationen

```
SELECT * FROM
kredit NATURAL RIGHT OUTER JOIN kreditnehmer
```

krnr	fname	betrag	kname
K-170	Downtown	3000,00	Jones
K-230	Redwood	4000,00	Smith
K-155	NULL	NULL	Hayes

Die Angabe von ***NATURAL*** verknüpft einen Equi-Join über alle Attribute, die in beiden Relationen gleich heißen

FULL OUTER JOIN

Der ***FULL OUTER JOIN*** ist eine Kombination des ***LEFT*** und ***RIGHT OUTER JOIN***

```
SELECT * FROM kredit FULL OUTER JOIN
  kreditnehmer USING (krnr)
```

krnr	fname	betrag	kname
K-170	Downtown	3000,00	Jones
K-230	Redwood	4000,00	Smith
K-260	Perryridge	1700,00	NULL
K-155	NULL	NULL	Hayes

Suche alle Kunden, die entweder ein Konto oder einen Kredit (aber nicht beides) besitzen

```
SELECT kname
FROM (kontoinhaber NATURAL FULL OUTER JOIN kreditnehmer)
WHERE konr IS NULL OR krnr IS NULL
```

4.11 Transaktionen

- Eine Transaktion ist eine Sequenz von Abfragen und Update Anweisungen, die als eine Einheit abgearbeitet werden
  - Transaktionen werden implizit gestartet und durch eine der folgenden Anweisungen beendet
    - **`COMMIT`** [**`WORK`**]: alle Updates werden permanent
    - **`ROLLBACK`** [**`WORK`**]: alle Updates der Transaktion werden zurückgesetzt
- Motivierendes Beispiel
  - Der Geldtransfer zwischen 2 Konten besteht aus 2 Schritten
  - (1) Buche Geld x von einem Konto ab und
  - (2) Füge x einem anderen Konto hinzu
  - Falls ein Schritt erfolgreich ist und der andere scheitert, ist das DBMS in einem inkonsistenten Zustand
  - Daher müssen beide Schritte erfolgreich sein, oder keiner!
- Transaktionsmechanismen oft abhängig von der spezifischen DBMS Implementation
  - In manchen DBMS wird jede erfolgreiche SQL Anweisung automatisch „committet“