

# 4. SQL

## Einleitung

---

SQL ist eine **deklarative Anfragesprache** ("was" nicht "wie").

### Bestandteile von SQL

SQL setzt sich aus mehreren Teilen zusammen:

- **Datenbeschreibungssprache (Data Definition Language, DDL):**
    - Erstellt/ändert das Schema
    - `create`, `alter`, `drop`
  - **Datenmanipulationssprache (Data Manipulation Language, DML):**
    - Ändert Datensammlungen
    - `insert`, `update`, `delete`
  - **Anfragesprache (Data Query Language, DQL):**
    - Formuliert Anfragen auf den Ausprägungen
    - `select * from where ...`
  - **Ablaufsteuerungssprache (Transaction Control Language, TCL):**
    - Steuert Transaktionen
    - `commit`, `rollback`
  - **Datenaufsichtssprache (Data Control Language, DCL):**
    - Definiert/entzieht Zugriffsrechte
    - `grant`, `revoke`
-

# Relationen vs. Tabellen

---

## Mengen und Bags

- Bisher haben wir Relationen (Mengen) betrachtet.
- In SQL betrachten wir aber Tabellen (Bags, Multimengen).

## Was ist der Unterschied?

### Relationen (Mengen):

- Tabellen können keine Duplikate enthalten.
- Tabellen haben keine definierte Ordnung.
- Tabellen haben nicht unbedingt einen Schlüssel.

### Tabellen (Bags, Multimengen) in SQL:

- Können Duplikate enthalten.
  - Haben eine implizite Einfügeordnung (die aber bei Abfragen i.d.R. keine Rolle spielt, außer bei `ORDER BY` ).
  - Können Schlüssel definieren (Primärschlüssel, Fremdschlüssel), müssen es aber nicht.
-

# Grundlegende Datentypen in SQL

---

## Datentypen

- `character(n)`, `char(n)`
- `character varying(n)`, `varchar(n)`
- `integer`, `smallint`
- `numeric(p,s)`, `decimal(p,s)`
  - `p` = Präzision = max. Anzahl von Stellen (gesamt)
  - `s` = Scale = Anzahl von Stellen nach dem Komma
- `real`, `double`
- `blob` oder `raw` für sehr große binäre Daten
- `date` für Datumsangaben
- `xml` für XML-Dokumente
- ...

### `varchar(n)` vs. `char(n)`

- Beide sind auf Länge `n` beschränkt.
  - `char(n)` belegt immer `n` Bytes.
  - `varchar(n)` belegt nur den benötigten Platz, plus Längeninformation.
-

# Tabellen erstellen

## Tabelle erstellen

```
CREATE TABLE professor (
    empid    integer,
    name     varchar(10) NOT NULL,
    rank     char(2)
);
```

## Tabelle erstellen (mit UNIQUE Constraint)

```
CREATE TABLE professor (
    empid    integer UNIQUE NOT NULL,
    name     varchar(10) NOT NULL,
    rank     char(2)
);
```

Das `UNIQUE`-Constraint lässt in einer Spalte jeden Wert nur ein einziges Mal zu.

Tabellen kann man auch auf Basis von anderen Tabellen erstellen:

```
CREATE TABLE professor2 AS (SELECT * FROM professor);
```

## Primärschlüssel

`PRIMARY KEY` kennzeichnet ein Attribut als **Schlüsselattribut**.

### Primärschlüssel als Spalten-Constraint

```
CREATE TABLE professor (
    empid    integer PRIMARY KEY,
    name     varchar(10) NOT NULL,
    rank     char(2)
);
```

### Primärschlüssel als Tabellen-Constraint

```
CREATE TABLE professor (
    empid    integer,
    name     varchar(10) NOT NULL,
    rank     char(2),
    PRIMARY KEY (empid)
);
```

`PRIMARY KEY` kennzeichnet ein Attribut als **Schlüsselattribut**.

Das `Primary-Key`-Constraint beinhaltet `not-null`- und `unique`-Constraints.

## Fremdschlüssel

Gültige Werte des Fremdschlüssels müssen Werten des referenzierten Attributs entsprechen.

### Fremdschlüssel als Tabellen-Constraint

```
CREATE TABLE course (
    courseId      integer,
    title         varchar(30) NOT NULL,
    ects          integer,
    taughtby      integer,
    PRIMARY KEY (courseId),
    FOREIGN KEY (taughtby) REFERENCES professor(empid)
);
```

Fremdschlüssel können auch **Schlüsselkandidaten**(!) referenzieren (garantiert durch das `UNIQUE`-Constraint).

### Fremdschlüssel, der auf Primärschlüssel und UNIQUE-Constraint verweist

```
CREATE TABLE employee (
    cpr        integer PRIMARY KEY,
    empid     integer UNIQUE NOT NULL,
    name       varchar(20)
);

CREATE TABLE salary (
    cpr        integer REFERENCES employee(cpr),
    empid     integer REFERENCES employee(empid),
    salary     integer
);
```

## Defaultwerte

- Wird für ein Attribut beim Einfügen kein Wert angegeben, so wird dieser auf `NULL` gesetzt (Standard- bzw. Defaultwert).
- Beim Erstellen einer Tabelle können wir einen anderen Defaultwert definieren.

### Defaultwert für ein Attribut definieren

```
CREATE TABLE wine (
    wineID      integer NOT NULL,
    name        varchar(20) NOT NULL,
```

```

color      varchar(10) DEFAULT 'red',
year       integer,
vineyard   varchar(20)
);

```

Der Defaultwert für die Farbe eines Weins soll "red" sein.

## Zahlengeneratoren

**Zahlengeneratoren** erzeugen automatisch fortlaufende eindeutige IDs.

### Sequenz erstellen und als Defaultwert nutzen

```

CREATE SEQUENCE serial START 101;

CREATE TABLE wine (
    wineID    integer PRIMARY KEY DEFAULT nextval('serial'),
    name      varchar(20) NOT NULL,
    color     varchar(10),
    year      integer,
    vineyard  varchar(20)
);

```

Beim Hinzufügen von Weinen soll automatisch eine eindeutige `wineID` generiert werden.

---

# Tabellen verändern

## Attribute hinzufügen

```
ALTER TABLE professor  
ADD COLUMN office integer;
```

## Attribut löschen

```
ALTER TABLE professor  
DROP COLUMN name;
```

## Attributtyp ändern

```
ALTER TABLE professor  
ALTER COLUMN name type varchar(30);
```

```
CREATE TABLE professor (  
    empid    integer NOT NULL,  
    name     varchar(10) NOT NULL,  
    rank     char(2)  
);
```

## Tabelle löschen

---

```
DROP TABLE professor;
```

## Tabelle leeren

```
TRUNCATE TABLE professor;
```

Kann nicht verwendet werden, wenn der Fremdschlüssel einer anderen Tabelle auf die zu löschen Tabelle zeigt.

---

## Daten einfügen

```

INSERT INTO professor VALUES
(2136, 'Curie', 'C4', 36),
(2137, 'Kant', 'C4', 7);

INSERT INTO student (studid, name) VALUES
(28121, 'Archimedes');

INSERT INTO student (name) VALUES
('Meier');

INSERT INTO takes
SELECT studid, courseid
FROM student, course
WHERE title = 'Logics';

```

---

## Daten löschen und ändern

### Daten löschen

```

DELETE FROM student
WHERE semester > 13;

DELETE FROM student;

```

### Daten ändern

```

UPDATE student
SET semester = semester + 1;

UPDATE student
SET semester = semester + 1 WHERE ...;

```

# Rechteverwaltung

---

## Rechte gewähren

```
GRANT select (empid), update (office)
ON professor
TO some_user, another_user;
```

## Rechte entziehen

```
REVOKE ALL
ON professor
FROM some_user, another_user;
```

Rechte auf Tabellen, Spalten, ...:

```
select, insert, update, delete, rule, references, trigger
```

---

# Transaktionssteuerung

---

```
BEGIN;
```

kennzeichnet den **Beginn** einer Transaktion.

```
COMMIT;
```

schließt eine Transaktion ab.

Alle Änderungen, die in der Transaktion gemacht wurden, werden für andere sichtbar und sind garantiert dauerhaft (auch wenn sich ein Absturz ereignen sollte).

```
ROLLBACK;
```

setzt eine Transaktion zurück.

Alle Änderungen, die in der Transaktion gemacht wurden, werden verworfen.

---

# Anfragesprache

## Grundgerüst einer SQL-Anfrage (SFW-Block)

```
SELECT <Liste von Attributen>
FROM <Liste von Tabellen>
WHERE <Bedingung>;
```

- `SELECT <Liste von Attributen>` : Projektionsliste mit arithmetischen Operationen und Aggregatfunktionen
- `FROM <Liste von Tabellen>` : zu verwendende Tabellen, evtl. Umbenennungen
- `WHERE <Bedingung>` : Selektion und Join-Bedingungen, geschachtelte Anfragen

## Relationale Algebra → SQL

Mehr dazu → 1. Relationale Algebra

- Projektion  $\pi \rightarrow$  `SELECT`
- Kreuzprodukt  $\times \rightarrow$  `FROM`
- Selektion  $\sigma \rightarrow$  `WHERE`

## Auswahl von Tabellen

Man kann Tupelvariablen für Relationen definieren.

```
SELECT *
FROM wine AS W;
```

```
SELECT *
FROM wine W;
```

W	wineID	name	color	year	vineyard
	1042	La Rose Grand Cru	red	1998	Château La Rose
	2168	Creek Shiraz	red	2003	Creek
	3456	Zinfandel	red	2004	Helena
	2171	Pinot Noir	red	2001	Creek
	3478	Pinot Noir	red	1999	Helena
	4711	Riesling Reserve	white	1999	Müller
	4961	Chardonnay	white	2002	Bighorn

## Kreuzprodukt (Kartesisches Produkt)

---

Bei mehr als einer Tabelle in der `FROM`-Klausel wird das Kreuzprodukt gebildet.

```
SELECT *
FROM wine, producer;
```

Als Ergebnis werden **alle** Kombinationen generiert!

---

## Tupelvariablen für mehrfachen Zugriff

---

Die Definition von Tupelvariablen erlaubt mehrfachen Zugriff auf eine Tabelle (z.B. Self-Join).

```
SELECT *
FROM wine w1, wine w2;
```

Die Spalten des Ergebnisses lauten:

```
w1.wineID, w1.name, w1.color, w1.year, w1.vineyard, w2.wineID, w2.name, w2.color,
w2.year, w2.vineyard
```

---

# Natural Join

- Frühe SQL-Versionen:
  - Kein eigener Join-Operator.
  - Joinbedingung durch Prädikat in der WHERE -Klausel definiert.
- Neuere SQL-Versionen:
  - Kennen explizite Join-Operatoren.
  - NATURAL JOIN als Abkürzung für die ausführliche Anfrage mit Kreuzprodukt und anschließender Filterung.

## Natürlicher Join als expliziter Operator

- Syntax:

```
SELECT *
FROM tabelle1 NATURAL JOIN tabelle2;
```

- Implizite Joinbedingung: Spalten mit gleichem Namen in beiden Tabellen werden für den Join verwendet.
- Ergebnisspalten:
  - Spalten mit eindeutigen Namen aus beiden Tabellen.
  - Joinspalten erscheinen nur einmal im Ergebnis.
- Ergebnistupel: Kombination von Tupeln aus beiden Tabellen, die in den Joinspalten übereinstimmende Werte haben.

## Beispiel (aus den Slides)

Angenommen, wir haben die Tabelle `wine` mit den Spalten `wineID`, `name`, `color`, `year`, `vineyard` und die Tabelle `producer` mit den Spalten `producer`, `vineyard`, `region`.

```
SELECT *
FROM wine NATURAL JOIN producer;
```

- Die Joinbedingung basiert auf der Spalte `vineyard`, da sie in beiden Tabellen existiert.
- Die Spalte `vineyard` erscheint nur einmal im Ergebnis.

**Wichtig:** Ein NATURAL JOIN führt implizit einen Gleichheitsvergleich aller Spalten mit identischem Namen durch. Es ist wichtig, sich der Struktur der Tabellen bewusst zu sein, um unerwartete Join-Ergebnisse zu vermeiden, falls ungewollt Spalten mit gleichen Namen existieren.

# Weitere Join-Varianten

Es gibt weitere Möglichkeiten, Joins auszudrücken:

## USING-Klausel

- Syntax:

```
SELECT *
FROM tabelle1 JOIN tabelle2
USING (spalte1, spalte2, ...);
```

- Nach `USING` steht eine Liste von Spalten, über die der Join berechnet wird (Gleichheit).
- Funktioniert ähnlich wie `NATURAL JOIN`, erlaubt aber die explizite Angabe der Joinspalten, falls mehrere Spalten mit gleichen Namen existieren, aber nicht alle für den Join relevant sind.
- Joinspalten erscheinen nur einmal im Ergebnis.

## ON-Klausel

- Syntax:

```
SELECT *
FROM tabelle1 JOIN tabelle2
ON bedingung;
```

- Nach `ON` kann ein beliebiger boolescher Ausdruck stehen (wie in der `WHERE`-Klausel).
- Ermöglicht komplexe Joinbedingungen, die nicht nur auf Gleichheit von Spalten basieren (Theta-Join).

---

# Kreuzprodukt als expliziter Operator

## Kreuzprodukt (Kartesisches Produkt)

- Kombiniert jedes Tupel der ersten Tabelle mit jedem Tupel der zweiten Tabelle.
- Syntax:

```
SELECT *
FROM tabelle1, tabelle2;
```

oder explizit:

```
SELECT *
FROM tabelle1 CROSS JOIN tabelle2;
```

- Ergebnisspalten: Alle Spalten aus beiden Tabellen.
  - Anzahl der Ergebnistupel: Produkt der Anzahl der Tupel in den beiden Tabellen ( $|Tabelle1| \times |Tabelle2|$ ).
  - In den meisten Fällen ist ein reines Kreuzprodukt ohne anschließende Filterung ( WHERE - Klausel oder Join-Bedingung) wenig sinnvoll, da es zu sehr großen Ergebnismengen führen kann. Es bildet aber die Grundlage für andere Join-Operationen.
-

# Tupelvariable

## In Zwischenergebnissen

- "Zwischenergebnistabellen" aus SQL-Operationen oder einem SFW-Block können durch Tupelvariablen mit einem Namen versehen werden (Aliasing).
- Syntax:

```
SELECT attribute
  FROM relation AS alias;
```

Das Schlüsselwort `AS` ist optional.

## Verwendung

- Ermöglicht es, auf Attribute der resultierenden Tabelle über den Aliasnamen zuzugreifen.
- Besonders nützlich bei Joins, um die Herkunft von Attributen zu kennzeichnen oder um mehrdeutige Attributnamen aufzulösen.

## Beispiel

```
SELECT result.vineyard
  FROM (wine NATURAL JOIN producer) AS result;
```

In diesem Beispiel erhält das Ergebnis des `NATURAL JOIN` zwischen `wine` und `producer` den Aliasnamen `result`. Anschließend wird das Attribut `vineyard` über den Alias `result.vineyard` ausgewählt.

## Wichtiger Hinweis

Wenn eine Tupelvariable (Alias) definiert wird, dann können Spalten nur noch mit dem Variablenamen referenziert werden und nicht mehr mit dem Tabellennamen!

## Beispiel für eine ungültige Anfrage ohne Alias

```
SELECT name, year, vineyard
  FROM wine, producer
 WHERE wine.vineyard = producer.vineyard;
```

- Ist dies eine gültige SQL-Anfrage? Nein.
- Die Referenz ist nicht eindeutig: Das Resultat des Joins hat zwei Spalten mit dem Namen `vineyard`.

## Richtiger Ausdruck mit Alias

```
SELECT w.name, w.year, w.vineyard  
FROM wine AS w, producer AS p  
WHERE w.vineyard = p.vineyard;
```

Hier werden die Tabellen `wine` und `producer` mit den Aliassen `w` bzw. `p` versehen, um eindeutig auf die Spalte `vineyard` zugreifen zu können.

---

# Die SELECT-Klausel

## Festlegung der Projektionsattribute

- Syntax:

```
SELECT [DISTINCT] projektionsliste
      FROM ...
```

## Projektionsliste

- Liste von Spalten.
- Spezialfall: `*` steht für alle Spalten der Tabellen in der `FROM`-Klausel.
- Arithmetische Ausdrücke bestehend aus Konstanten und Spalten der Tabellen (z.B., `price * 1.19`).
- Aggregatfunktionen über Spalten der Tabellen (mehr dazu später).

## Duplikateliminierung

- Die Standardeinstellung von `SELECT` ist, alle ausgewählten Tupel im Ergebnis beizubehalten, auch wenn sie Duplikate sind. Das Ergebnis ist somit ein Multimenge (Bag).

## Beispiel ohne `DISTINCT`

```
SELECT name
      FROM wine;
```

name
La Rose Grand Cru
Creek Shiraz
Zinfandel
Pinot Noir
Pinot Noir
Riesling Reserve
Chardonnay

## Verwendung von `DISTINCT`

- Das Schlüsselwort `DISTINCT` vor der Projektionsliste bewirkt, dass Duplikate im Ergebnis eliminiert werden. Das Ergebnis ist eine Menge (Set) von eindeutigen Tupeln.

## Beispiel mit `DISTINCT`

```
SELECT DISTINCT name  
FROM wine;
```

- Entspricht der Projektion der Relationenalgebra.

name
La Rose Grand Cru
Creek Shiraz
Zinfandel
Pinot Noir
Riesling Reserve
Chardonnay

# Sortierung

---

- Die `ORDER BY`-Klausel dient zur Sortierung der Ergebnismenge nach einer oder mehreren Spalten.
- Syntax:

```
SELECT spalte1, spalte2, ...
FROM tabelle
ORDER BY spalte_sortier1 [ASC|DESC], spalte_sortier2 [ASC|DESC], ...;
```

- Reihenfolge:
  - `ASC` : aufsteigend (ascending) - Standard.
  - `DESC` : absteigend (descending).
- Es können mehrere Spalten für die Sortierung angegeben werden. Die Sortierung erfolgt zuerst nach der ersten Spalte, dann innerhalb gleicher Werte der ersten Spalte nach der zweiten Spalte usw.

## Beispiel

```
SELECT empid, name, rank
FROM professor
ORDER BY rank DESC, name ASC;
```

Das Ergebnis wird zuerst absteigend nach dem `rank` sortiert. Innerhalb der gleichen Ränge werden die Professoren aufsteigend nach ihrem `name` sortiert.

---

## Die WHERE-Klausel

- Die WHERE -Klausel filtert die Tupel der in der FROM -Klausel angegebenen Tabellen basierend auf einer Bedingung.
- Syntax:

```
SELECT ...
FROM tabelle
WHERE bedingung;
```

- Die bedingung beschreibt Bedingungen, die für alle Ergebnistupel gelten müssen.
- Mögliche Operatoren und Ausdrücke in der WHERE -Klausel:
  - Vergleiche von Attributen mit anderen Attributen und/oder Konstanten (=, >, <, >=, <=, <>).
  - Bereichsoperatoren (BETWEEN, NOT BETWEEN).
  - Mengenoperatoren (IN, NOT IN).
  - Mustervergleich (LIKE, NOT LIKE).
  - Nullwertprüfung (IS NULL, IS NOT NULL).
- Bedingungen in der WHERE -Klausel können logisch mit AND, OR, NOT kombiniert werden.
- Jede Bedingung, die für einen Theta-Join definiert werden kann, kann in der WHERE -Klausel verwendet werden.

### Anfrage mit einer Tabelle

```
SELECT empid, name
FROM professor
WHERE rank = 'C4';
```

professor	
<b>empid</b>	<b>name</b>
2125	Socrates
2126	Russel
2136	Curie
2137	Kant

professor			
<b>empid</b>	<b>name</b>	<b>rank</b>	<b>office</b>
2125	Socrates	C4	226
2126	Russel	C4	232
2127	Kopernikus	C3	310
2133	Popper	C3	52
2134	Augustinus	C3	309
2136	Curie	C4	36
2137	Kant	C4	7

**Wichtiger Hinweis:** SQL legt nicht fest, in welcher Reihenfolge Selektion, Projektion und Join ausgeführt werden. Der Anfrageoptimierer des Datenbankmanagementsystems (DBMS) bestimmt die effizienteste Ausführungsreihenfolge.

## Anfrage mit mehreren Tabellen

```
SELECT name, title
FROM professor, course
WHERE professor.empid = course.taughtby AND title = 'Bioethik';
```

- Welcher Professor liest „Bioethik“ (SQL und relationale Algebra)?

### Ausdruck in relationaler Algebra

$$\pi_{name,title}(\sigma_{empid=taughtby \wedge title='Bioethik'}(professor \times course))$$

## Übersetzung von SQL-Anfragen in relationale Algebra

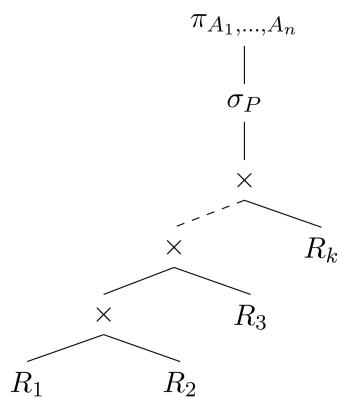
### Allgemeine Form einer (geschachtelten) SQL-Anfrage

Allgemeine Form einer  
(ungeschachtelten) SQL-Anfrage

```
SELECT A1, ..., An
FROM R1, ..., Rk
WHERE P;
```

Übersetzung in relationale Algebra

$$\pi_{A_1, \dots, A_n}(\sigma_P(R_1 \times \dots \times R_k))$$



# Mengenoperationen

- Mengenoperationen erfordern **Vereinigungskompatibilität**:
  - Gleiche Anzahl von Attributen mit kompatiblen Domänen.
  - Beide Relationen sind gleich (beide sind auf charakterbasierende Wertebereiche (unabhängig von der Länge des Strings))
  - Beide sind numerische Wertebereiche (unabhängig vom genauen Typ), z.B. integer oder float.
- Ergebnisschema: Spaltennamen der **ersten** Tabelle.

## UNION

- Vereinigung zweier Mengen. Entfernt Duplikate im Endergebnis.
- Beispiel:

```
(SELECT A, B, C FROM R1)
UNION
(SELECT A, C, D FROM R2);
```

## Duplikate und Mengenoperationen

- Bei Mengenoperationen ist Duplikateliminierung (`DISTINCT`) default!

## UNION vs. UNION ALL

- `UNION` : Entfernt Duplikate aus dem Ergebnis.

```
(SELECT name FROM assistant)
UNION
(SELECT name FROM assistant);
```

- `UNION ALL` : Behält alle Duplikate im Ergebnis bei.

```
(SELECT name FROM assistant)
UNION ALL
(SELECT name FROM assistant);
```

# Schachtelung von Anfragen

- Für Vergleiche mit Wertemengen sind Unteranfragen notwendig.
- Standardvergleiche in Verbindung mit den Quantoren `ALL` ( $\forall$ ) und `ANY` ( $\exists$ ).
- Spezielle Prädikate für den Zugriff auf Mengen: `IN` und `EXISTS`.

## Unkorrelierte Unteranfragen

- Notation: `attribut IN (SFW-Block)`

### Beispiel 1: Vergleich von Wert mit Menge von Werten

```
SELECT name
FROM professor
WHERE empid IN (SELECT taughtby
                  FROM course);
```

### Beispiel 2: Welche Weine wurden in der Region "Bordeaux" produziert (mit Hilfe von `IN`)?

Gegeben seien die Tabellen:

- `wine (wineID, name, color, year, vineyard)`
- `producer (vineyard, area, region)`

```
SELECT name
FROM wine
WHERE vineyard IN (SELECT vineyard
                     FROM producer
                     WHERE region = 'Bordeaux');
```

Können wir die gleiche Anfrage formulieren, ohne `IN` zu verwenden?

Ja, mit einem Join:

```
SELECT name
FROM wine
NATURAL JOIN producer
WHERE region = 'Bordeaux';
```

# Negation des IN-Prädikats

## Simulation des Differenzoperators

Der relationale Algebra Ausdruck:

$$\pi_{vineyard}(producer) - \pi_{vineyard}(wine)$$

entspricht der folgenden SQL-Anfrage:

```
SELECT vineyard
FROM producer
WHERE vineyard NOT IN (
    SELECT vineyard
    FROM wine
);
```

- Das `NOT IN`-Prädikat überprüft, ob ein Wert *nicht* in der Menge der Werte ist, die von der Unterabfrage zurückgegeben werden.
- In diesem Fall werden alle `vineyard`-Werte aus der `producer`-Tabelle ausgewählt, die *nicht* in der Menge der `vineyard`-Werte aus der `wine`-Tabelle enthalten sind.

## Korrelierte Unteranfragen

- Eine korrelierte Unteranfrage ist eine Unteranfrage, die sich auf eine Spalte aus der äußeren Abfrage bezieht.
- Die Unteranfrage wird für jede Zeile der äußeren Abfrage neu ausgewertet.

### Beispiel 1: Verwendung von `EXISTS`

```
SELECT name
FROM professor p
WHERE EXISTS (SELECT *
               FROM course v
               WHERE v.taughtby = p.empid);
```

- **Was wird hier berechnet?** Die Namen jener Professoren, die irgendeine Lehrveranstaltung (LVA) abhalten.
- Für jeden Professor in der äußeren Abfrage prüft die Unterabfrage, ob es mindestens einen Eintrag in der Tabelle `course` gibt, dessen `taughtby`-Wert mit dem `empid` des aktuellen Professors übereinstimmt.

### Beispiel 2: Auswirkung der `SELECT`-Liste in der Unteranfrage

```
SELECT name
FROM professor p
WHERE EXISTS (SELECT 42
               FROM course v
               WHERE v.taughtby = p.empid);
```

- **Was passiert, wenn wir das `SELECT *` in der Unteranfrage ändern?**
- **Erhalten wir nun andere Ergebnisse?** Nein.
- Das `EXISTS`-Prädikat prüft lediglich auf die Existenz von Zeilen, die die Bedingung erfüllen. Die tatsächlich selektierten Spalten in der Unteranfrage sind irrelevant.
- **Ist das Ergebnis der inneren Anfrage leer?** Nicht notwendigerweise.
- **Für jedes Ergebnistupel in der inneren Anfrage wird 42 zurückgegeben.** Entscheidend ist, ob *mindestens ein* Tupel die `WHERE`-Bedingung der inneren Anfrage erfüllt. Wenn ja, ist die `EXISTS`-Bedingung für die aktuelle Zeile der äußeren Abfrage wahr.

# Quantoren: IN vs. EXISTS

---

## IN

- **Bedeutung:** Ist das "linke Tupel" in der "rechten Menge" enthalten?
- **Funktionsweise:** Vergleicht einen Wert (oder ein Tupel von Werten) mit der Menge der Ergebnisse, die von der Unterabfrage zurückgegeben werden. Die Bedingung ist wahr, wenn der Wert (oder das Tupel) in dieser Menge enthalten ist.
- **Beispiel:**

```
SELECT ...
WHERE (studid, courseid) IN (SELECT studid, courseid FROM takes);
```

## EXISTS

- **Bedeutung:** Ist die "rechte Seite" leer? (Gibt die Unterabfrage mindestens eine Zeile zurück?)
- **Funktionsweise:** Prüft, ob die Unterabfrage mindestens eine Zeile zurückgibt, die die WHERE -Bedingung innerhalb der Unterabfrage erfüllt. Die Bedingung ist wahr, wenn die Unterabfrage mindestens eine Zeile zurückgibt, unabhängig von den selektierten Spalten.
- **Beispiel:**

```
SELECT ...
WHERE EXISTS (SELECT * FROM takes WHERE ...);
```

**Wichtiger Unterschied:** IN vergleicht Werte, während EXISTS auf die Existenz von Zeilen prüft. EXISTS ist oft effizienter, besonders bei großen Unterabfragen, da es stoppt, sobald die erste passende Zeile gefunden wurde.

## Quantor: ALL und ANY

- Vergleich eines Wertes mit einer Menge von Werten.
- Das Prädikat mit `ALL` ist wahr, wenn die Bedingung für *alle* Werte in der Menge der Unterabfrage erfüllt ist.

### Beispiel All:

```
SELECT name
FROM student
WHERE semester >= ALL (SELECT semester
                        FROM student);
```

- Dann gibts noch `ANY`
- Das ist wahr, wenn mindestens eins davon erfüllt ist.

### Beispiel Any:

```
SELECT name
FROM student
WHERE semester >= ANY (SELECT semester
                        FROM student);
```

## Vergleich All und Any

### Tabellen:

- `wine(wineID, name, color, year, vineyard)`
- `producer(vineyard, area, region)`

### Beispiel 1: Finde den ältesten Wein (verwende ALL).

```
SELECT *
FROM wine
WHERE year <= ALL (
    SELECT year FROM wine
);
```

- `ALL` : Die Bedingung ist wahr, wenn sie für *alle* Werte in der Unterabfrage zutrifft.
- Hier: Wählt alle Weine aus, deren `year`-Wert kleiner oder gleich dem `year`-Wert *jedes* anderen Weins ist (also der älteste Wein bzw. die ältesten Weine).

### Beispiel 2: Finde alle Weingüter, die Rotweine herstellen (verwende ANY).

```
SELECT *
FROM producer
WHERE vineyard = ANY (
    SELECT vineyard FROM wine
    WHERE color = 'red'
);
```

- ANY : Die Bedingung ist wahr, wenn sie für *mindestens einen* Wert in der Unterabfrage zutrifft.
  - Hier: Wählt alle Weingüter aus, deren `vineyard`-Wert mit dem `vineyard`-Wert *mindestens eines* Rotweins übereinstimmt.
-

## Unteranfragen in der WHERE-Klausel ohne Quantoren

**Beispiel: Vergleich eines Wertes mit einem anderen Wert**

```
SELECT *
FROM grades
WHERE grade < (SELECT AVG(grade) FROM grades);
```

- Hier: Wählt alle Einträge aus der Tabelle `grades` aus, deren `grade`-Wert kleiner ist als der Durchschnitt aller `grade`-Werte in derselben Tabelle.
- Die Unterabfrage `(SELECT AVG(grade) FROM grades)` wird einmalig ausgeführt und liefert einen einzelnen Wert (den Durchschnitt).
- Dieser einzelne Wert wird dann mit dem `grade`-Wert jeder Zeile der äußeren Abfrage verglichen.

## Unteranfragen in der SELECT-Klausel

- Die Unteranfrage wird für jedes Lösungstupel der äußeren Abfrage evaluiert.
- Die Unteranfrage ist korreliert.

**Beispiel: Berechne einen Wert für jedes äußere Tupel.**

```
SELECT empid, name, (SELECT SUM(ects)
                      FROM course
                      WHERE taughtby = empid) AS teachingLoad
FROM professor;
```

- Hier: Wählt die `empid` und den `name` aus der Tabelle `professor` aus.
- Für jede Zeile in `professor` wird die Unterabfrage ausgeführt.
- Die Unterabfrage `(SELECT SUM(ects) FROM course WHERE taughtby = empid)` berechnet die Summe der `ects` aller Kurse, die von dem aktuellen Professor (`empid` der äußeren Abfrage) unterrichtet werden.
- Das Ergebnis dieser Summe wird als Alias `teachingLoad` in die Ergebnismenge aufgenommen.
- Die Unterabfrage ist **korreliert**, da sie sich auf den Wert `empid` der äußeren Abfrage bezieht.

## COALESCE()

**Aufgabe:** Gib eine Liste aller Studierenden aus inkl. deren Noten für abgeschlossene Kurse. Für Studierende, die keinen Kurs besucht haben, sollen `courseID` und `grade` den Wert `0` annehmen.

```
SELECT s.studID, s.name,
       COALESCE(g.courseID, 0),
       COALESCE(g.grade, 0)
  FROM student s LEFT OUTER JOIN grades g
    ON s.studID = g.studID;
```

- `COALESCE(Ausdruck1, Ausdruck2, ...)` : Gibt den ersten Ausdruck zurück, der nicht `NULL` ist. Sind alle Ausdrücke `NULL`, wird `NULL` zurückgegeben.
- Im Beispiel:
  - `COALESCE(g.courseID, 0)` : Wenn `g.courseID` für einen Studenten `NULL` ist (weil er keinen Kurs belegt hat), wird `0` zurückgegeben. Andernfalls wird der tatsächliche `g.courseID`-Wert verwendet.
  - `COALESCE(g.grade, 0)` : Wenn `g.grade` für einen Studenten `NULL` ist, wird `0` zurückgegeben. Andernfalls wird die tatsächliche `g.grade` verwendet.
- `LEFT OUTER JOIN` : Stellt sicher, dass *alle* Zeilen aus der linken Tabelle (`student`) im Ergebnis enthalten sind. Wenn es keine übereinstimmende Zeile in der rechten Tabelle (`grades`) gibt, werden die Spalten aus `grades` als `NULL` behandelt.

**Achtung:** Die Datentypen der Ausdrücke in `COALESCE()` müssen kompatibel sein. Im Beispiel wird davon ausgegangen, dass `courseID` und `grade` numerische Typen sind oder implizit in numerische Typen konvertiert werden können.

# Allquantifizierte Anfragen

**Aufgabe:** Welche Studierenden haben *alle* 4-ECTS Lehrveranstaltungen gehört?

**Tabellen:**

- `student(studID, name, semester)`
- `takes(studID, courseID)`
- `course(courseID, title, ects, taughtBy)`

**Relationale Algebra:**

$$takes \div \pi_{courseID}(\sigma_{ects=4}(course))$$

- $\sigma_{ects=4}(course)$ : Selektiert alle Kurse mit 4 ECTS-Punkten.
- $\pi_{courseID}(\sigma_{ects=4}(course))$ : Projiziert die `courseID`s der 4-ECTS-Kurse.
- $\div$ : Divisionsoperator.  $A \div B$  liefert alle Tupel  $t$  aus  $A$ , sodass für *jedes* Tupel  $u$  in  $B$  das Tupel  $(t, u)$  in  $A$  existiert.
- Im Kontext: Liefert alle `studID`s aus `takes`, sodass für *jede* `courseID` eines 4-ECTS-Kurses ein Eintrag mit dieser `studID` und der `courseID` in `takes` existiert.

**Allquantifizierte Anfragen - Nicht direkt in SQL**

SQL stellt **keinen Allquantor** direkt zur Verfügung.

**Realisierung durch Logische Äquivalenz (mittels 2 x NOT EXISTS):**

Die Anfrage "Finde alle Studierenden, die alle 4-ECTS Lehrveranstaltungen gehört haben" ist logisch äquivalent zu "Finde alle Studierenden, für die es *keine* 4-ECTS Lehrveranstaltung gibt, die sie *nicht* gehört haben."

Dies kann in SQL mit zwei `NOT EXISTS`-Klauseln umgesetzt werden

Umformulierung in äquivalente Anfrage mit Negation und Existenzquantor (Prädikatenlogik)

$$\forall x F(x) \Leftrightarrow \neg \exists x (\neg F(x))$$

**Allquantifizierte Anfragen - "Logische Umformung"**

**Umformulierung in äquivalente Anfrage mit Negation und Existenzquantor (Prädikatenlogik):**

$$\forall x F(x) \Leftrightarrow \neg \exists x (\neg F(x))$$

- $\forall$ : Allquantor ("für alle")
- $\exists$ : Existenzquantor ("es existiert mindestens ein")

- $\neg$ : Negation ("nicht")
- $F(x)$ : Eine Aussage über  $x$

**Beispiel:** Welche Studierenden haben alle 4-ECTS Lehrveranstaltungen gehört?

$\iff$  Suchen Sie jene Studierenden, für die gilt: sie haben alle 4-ECTS Lehrveranstaltungen gehört.

$\iff$  Suchen Sie jene Studierenden, für die **nicht** gilt: es gibt eine 4-ECTS Lehrveranstaltung, die der/die Studierende **nicht** gehört hat.

Diese logische Umformung ist die Grundlage für die Implementierung allquantifizierter Anfragen in SQL mithilfe von `NOT EXISTS`.

**SQL-Umsetzung folgt nun direkt aus:**

Suchen Sie jene Studierende, für die **nicht** gilt: es gibt eine 4-ECTS Lehrveranstaltung, für die **nicht** gilt: die/der Studierende hat diese Lehrveranstaltung gehört.

```
SELECT s.*  
FROM student s  
WHERE NOT EXISTS (SELECT *  
                   FROM course c  
                   WHERE c.ects = 4  
                   AND c.courseID NOT IN (SELECT t.courseID  
                                         FROM takes t  
                                         WHERE t.studID = s.studID));
```

- Die äußere `NOT EXISTS`-Klausel prüft für jeden Studenten `s` aus der Tabelle `student`, ob die innere `SELECT`-Anweisung **keine** Zeilen zurückliefert.
- Die innere `SELECT`-Anweisung findet alle 4-ECTS-Kurse (`WHERE c.ects = 4`), deren `courseID` **nicht** in der Menge der `courseID` s enthalten ist, die der aktuelle Student `s` in der Tabelle `takes` belegt hat (`WHERE t.studID = s.studID`).
- Wenn die innere `SELECT`-Anweisung **keine** Zeilen zurückliefert, bedeutet das, dass der Student für jeden 4-ECTS-Kurs einen Eintrag in der Tabelle `takes` hat, also alle 4-ECTS-Kurse gehört hat. In diesem Fall ist die Bedingung der äußeren `NOT EXISTS`-Klausel wahr, und der Student wird ausgewählt.

## Mächtigkeit des SQL-Kerns

---

relationale Algebra	SQL
projection	<code>SELECT DISTINCT</code>
selection	<code>WHERE</code> ohne Schachtelung
join	<code>FROM</code> , <code>WHERE</code> mit Join oder <code>NATURAL JOIN</code>
renaming	<code>FROM</code> mit Tupelvariable; <code>AS</code>
difference	<code>WHERE</code> mit Schachtelung; <code>EXCEPT</code>
intersection	<code>WHERE</code> mit Schachtelung; <code>INTERSECT</code>
union	<code>UNION</code>

---

# Joins

## Join Varianten:

- CROSS JOIN
- NATURAL JOIN
- JOIN oder INNER JOIN
- LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN

## Standard Formulierung (für INNER JOIN):

```
SELECT *
FROM R1, R2
WHERE R1.A = R2.B;
```

- Kombiniert Zeilen aus `R1` und `R2`, bei denen der Wert der Spalte `A` in `R1` gleich dem Wert der Spalte `B` in `R2` ist.
- Nur übereinstimmende Zeilen aus beiden Tabellen werden im Ergebnis berücksichtigt.

## Alternative Formulierung (expliziter INNER JOIN):

```
SELECT *
FROM R1 JOIN R2 ON R1.A = R2.B;
```

- Semantisch identisch zur Standardformulierung des Inner Joins.
- Die `JOIN`-Klausel in Verbindung mit der `ON`-Klausel spezifiziert explizit die Join-Bedingung.

## Right outer Join

```
SELECT *
FROM grades g RIGHT OUTER JOIN student s ON g.studid = s.studid;
```

g.studid	g.courseid	g.empid	g.grade	s.studid	s.name	s.semester
⊥	⊥	⊥	⊥	24002	Xenokrates	18
25403	5041	2125	2.0	25403	Jonas	12
⊥	⊥	⊥	⊥	26120	Fichte	10
⊥	⊥	⊥	⊥	26830	Aristoxenos	8
27550	4630	2137	2.0	27550	Schopenhauer	6
28106	5001	2126	1.0	28106	Carnap	3
⊥	⊥	⊥	⊥	29120	Theophrastos	2
⊥	⊥	⊥	⊥	29555	Feuerbach	2

Für jedes Tupel welches auf der rechten Seite ist, haben wir eine Ausgabe auch wenn sie keine Noten haben.

Das gibt es auch als **Left outer Join** und **Full outer Join**

---

# Aggregatfunktionen

Wie können wir die folgende Anfrage in SQL formulieren?

- Den durchschnittlichen Preis aller Artikel im Angebot
- Gesamtumsatz aller verkauften Produkte

Aggregatfunktionen berechnen neue Werte für eine Spalte.

Verfügbare Aggregatfunktionen:

`AVG`, `MAX`, `MIN`, `COUNT`, `SUM`

## Duplikate

Der Argumentspalte (außer im Falle `COUNT(*)`) kann optional ein `DISTINCT` oder ein `ALL` hinzugefügt werden.

- `DISTINCT`: Bevor die Aggregatfunktion ausgewertet wird, werden Duplikate entfernt.
- `ALL`: Duplikate werden für die Auswertung herangezogen (Standard/Default).

Null-Werte werden vor der Auswertung entfernt (außer im Falle `COUNT(*)`).

## Beispiel

### Anzahl von Weinen

```
SELECT COUNT(*) AS number
FROM wine;
```

### Die Anzahl von unterschiedlichen Regionen, wo Weine produziert werden

```
SELECT COUNT(DISTINCT region)
FROM producer;
```

### Die Namen und Jahre der Weine, die älter sind als der Durchschnitt

```
SELECT name, year
FROM wine
WHERE year < (SELECT AVG(year) FROM wine);
```

## Aggregatfunktionen in der WHERE-Klausel

⇒ Einsatz in Konstanten-Selektionen der `WHERE`-Klausel möglich.

Beispiel: Alle Weingüter, die nur einen Wein produzieren.

```
SELECT * FROM producer e
WHERE 1 = (SELECT COUNT(*) FROM wine w
            WHERE w.vineyard = e.vineyard);
```

- Die Unterabfrage `(SELECT COUNT(*) FROM wine w WHERE w.vineyard = e.vineyard)` zählt für jedes Weingut `e.vineyard` die Anzahl der Weine in der Tabelle `wine`, die von diesem Weingut produziert werden.
- Das Ergebnis dieser Unterabfrage ist ein einzelner numerischer Wert (die Anzahl der Weine).
- Die äußere `WHERE`-Klausel vergleicht diesen einzelnen Wert mit der Konstanten `1`. Nur Weingüter, für die die Anzahl der produzierten Weine gleich 1 ist, werden im Ergebnis ausgewählt.

## Verschachtelung

Nicht erlaubt!, daher:

Falsch:

```
SELECT f1(f2(A)) AS result
FROM R ...;
```

- Direkte Schachtelung von Aggregatfunktionen (`f1` angewendet auf das Ergebnis von `f2(A)`) ist in SQL nicht zulässig.

Möglich (durch Verwendung einer Subquery):

```
SELECT f1(temp) AS result
FROM ( SELECT f2(A) AS temp FROM R ... ) AS subquery;
```

- Um eine ähnliche Funktionalität zu erreichen, kann eine Subquery verwendet werden.
- Zuerst wird die innere Aggregatfunktion `f2(A)` in der Subquery berechnet und das Ergebnis unter dem Alias `temp` gespeichert.
- Anschließend kann die äußere Aggregatfunktion `f1` auf diese temporäre Spalte `temp` angewendet werden.

# Gruppierung

## Notation:

```
SELECT ...
FROM ...
(WHERE ... GROUP BY attributliste )
(HAVING bedingung );
```

## Aggregatsfunktion in Kombination mit Gruppierung

### Berechnung der Funktionen pro Gruppe:

```
SELECT taughtBy, SUM(ects)
FROM course
GROUP BY taughtBy;
```

- GROUP BY taughtBy : Gruppiert die Tupel der Tabelle course nach dem Wert des Attributs taughtBy .
- Alle Tupel, die den gleichen Wert für das Attribut taughtBy haben, werden zu einer Gruppe zusammengefasst.
- Die Aggregatfunktion SUM(ects) wird dann für jede dieser Gruppen separat berechnet.
- Das Ergebnis enthält für jeden taughtBy -Wert die Summe der ects der von dieser Person unterrichteten Kurse.

## Beispiel

### Was ist das Problem mit folgendem Ausdruck?

```
SELECT rank, COUNT(empId), name
FROM professor
GROUP BY rank;
```

- SQL erzeugt ein Lösungstupel pro Gruppe.
- Alle Spalten in der SELECT -Klausel müssen entweder in der GROUP BY -Klausel aufgeführt werden oder in einer Aggregatfunktion involviert sein.
- Im obigen Beispiel ist die Spalte name weder in der GROUP BY -Klausel ( rank ) enthalten noch wird sie in einer Aggregatfunktion verwendet. Da mehrere Professoren denselben rank haben können, ist unklar, welcher name in der Ergebnismenge für diese Gruppe angezeigt werden soll. Dies führt zu einem Fehler in den meisten SQL-Implementierungen.

Jeder Professor hat einen eigenen Namen und wir können nur einen Wert pro Gruppe haben, daher hat das Programm keine Ahnung was es ausgeben soll.

Man kann die Anfrage **retten** indem man den **Namen weglässt**.

### Weitere Beispiele:

```
SELECT COUNT(*)
FROM course
GROUP BY taughtBy
```

**richtig**

Das geht, weil wir hier nachschauen, wie viele Individuelle taughtBy Werte es gibt.

```
SELECT taughtBy, COUNT(*)
FROM course
GROUP BY ects
```

**falsch**

Das geht nicht, weil die `GROUP BY` -Klausel nicht mit den ausgewählten Spalten (`taughtBy` und `COUNT(*)`) übereinstimmt

```
SELECT taughtBy, COUNT(*)
FROM course
GROUP BY ects, taughtBy
```

**richtig**

Das stimmt, weil die `GROUP BY` -Klausel jetzt alle nicht-aggregierten Spalten enthält, die im `SELECT` -Teil der Abfrage stehen. Lass uns das genauer anschauen:

- `SELECT taughtBy, COUNT(*)` : Du wählst den Namen des Dozenten (`taughtBy`) und die Anzahl der Zeilen (`COUNT(*)`) aus der Tabelle `course` aus.
- `GROUP BY ects, taughtBy` : Du gruppierst die Zeilen nach *zwei* Spalten: `ects` und `taughtBy` .

## Having-Klausel

```
SELECT taughtBy, name, SUM(ects)
FROM course, professor
WHERE taughtBy = empID AND rank = 'C4'
GROUP BY taughtBy, name
HAVING AVG(ects) >= 3;
```

Die HAVING-Klausel drückt eine weitere Bedingung aus, die Gruppen erfüllen müssen, um im Resultat vorzukommen.

### Was wird hier berechnet? Was sind die Schritte?

#### Ausführen einer Anfrage mit Group by und Having

FROM course, professor

course × professor							
courseid	title	ects	taughtBy	empID	name	rank	office
5001	Basics	4	2137	2125	Socrates	C4	226
5041	Ethics	4	2125	2125	Socrates	C4	226
...	...	...	...	...	...	...	...
4630	Constructive Criticism	4	2137	2137	Kant	C4	7



WHERE taughtBy = empID AND rank = 'C4'

Dann bekommen wir das hier:

course × professor							
courseid	title	ects	taughtBy	empID	name	rank	office
5001	Basics	4	2137	2137	Kant	C4	7
5041	Ethics	4	2125	2125	Socrates	C4	226
5043	Theory of Cognition	3	2126	2126	Russel	C4	232
5049	DBS	2	2125	2125	Socrates	C4	226
4052	Logics	4	2125	2125	Socrates	C4	226
5052	Theory of Science	3	2126	2126	Russel	C4	232
5216	Bioethics	2	2126	2126	Russel	C4	232
4630	Constructive Criticism	4	2137	2137	Kant	C4	7



GROUP BY taughtBy, name

und führen `GROUP BY` aus:

course × professor							
taughtBy	name	courseid	title	ects	emplID	rank	office
2125	Socrates	5041	Ethics	4	2125	C4	226
		5049	DBS	2	2125	C4	226
		4052	Logics	4	2125	C4	226
2126	Russel	5043	Theory of Cognition	3	2126	C4	232
		5052	Theory of Science	3	2126	C4	232
		5216	Bioethics	2	2126	C4	232
2137	Kant	5001	Basics	4	2137	C4	7
		4630	Constructive Criticism	4	2137	C4	7



`HAVING AVG(ects) >= 3`

und jetzt noch `HAVING` dann haben wir das:

course × professor							
taughtBy	name	courseid	title	ects	emplID	rank	office
2125	Socrates	5041	Ethics	4	2125	C4	226
		5049	DBS	2	2125	C4	226
		4052	Logics	4	2125	C4	226
2137	Kant	5001	Basics	4	2137	C4	7
		4630	Constructive Criticism	4	2137	C4	7



`SUM(ects)`

Und jetzt berechnen wir noch die Summe:

"Pasted image 20250428144757.png" could not be found.

Und am Ende geben wir dann nur das aus, was in unserem `Select` steht:

taughtBy	name	SUM(ects)
2125	Socrates	10
2137	Kant	8

# Null-Werte

Manchmal sehr überraschende Anfrageergebnisse, wenn Null-Werte vorkommen.

```
SELECT COUNT(semester)
FROM student
WHERE semester < 13 OR semester >= 13;
```

```
SELECT COUNT(semester) FROM student;
```

Beide Anfragen liefern das gleiche Ergebnis, weil Tupel mit Null-Werten in der Spalte `semester` nicht gezählt werden.

## Erklärung:

- `COUNT(Spalte)` zählt nur die Zeilen, in denen die angegebene `Spalte` einen **nicht-NULL** Wert hat.
- In der ersten Anfrage filtert die `WHERE`-Klausel nach `semester`-Werten, die kleiner als 13 oder größer oder gleich 13 sind. Diese Bedingung schließt `NULL`-Werte nicht explizit aus oder ein. Da `NULL` weder kleiner als 13 noch größer oder gleich 13 ist, werden Zeilen mit `NULL` in der Spalte `semester` durch die `WHERE`-Klausel nicht gefiltert, aber von `COUNT(semester)` dennoch nicht gezählt.
- Die zweite Anfrage zählt alle nicht-NULL Werte in der Spalte `semester` über die gesamte Tabelle.
- Da `COUNT(semester)` in beiden Fällen nur nicht-NULL Werte in der Spalte `semester` berücksichtigt, liefern beide Anfragen dasselbe Ergebnis (die Anzahl der Studierenden, deren Semester-Wert nicht `NULL` ist).

## Auswertung von Null-Values

### Arithmetische Ausdrücke:

- Null-Werte werden "propagiert".
- `null + 1`  $\Rightarrow$  `null`
- `null * 0`  $\Rightarrow$  `null`

$\Rightarrow$  Sobald irgendwo `null` steht kommt `null` raus

### Vergleichsoperatoren:

- SQL hat eine dreiwertige Logik: `true`, `false` und `unknown`.
- Wenn mindestens ein Argument `null` ist, dann ist das Ergebnis `unknown`.
- `studid = 5`  $\Rightarrow$  `unknown` immer, wenn `studid null` ist.

⇒ Sobald null in einem Vergleich vorkommt kommt unknown raus und nicht mehr true oder false

### Test, ob ein Attribut den Wert null hat:

- Attribut IS NULL

nur so kann man testen ob gleich null ist.

### Zusammenfassend:

Logische Ausdrücke werden gemäß folgender Tabellen berechnet

NOT		
true	false	
unknown	unknown	
false	true	

AND	true	unknown	false
true	true	unknown	false
unknown	unknown	unknown	false
false	false	false	false

OR	true	unknown	false
true	true	true	true
unknown	true	unknown	unknown
false	true	unknown	false

## Null-Werte in der WHERE-Klausel und Gruppierung

### WHERE-Klausel:

- Die WHERE -Klausel reicht nur Tupel weiter, die zu true evaluieren.
- Tupel, die zu unknown evaluieren (z.B. bei Vergleichen mit NULL ), sind nicht Teil des Resultats.

### Gruppierung ( GROUP BY ):

- null wird als eigener, distinkter Wert interpretiert.
- Tupel mit null in der Gruppierungsspalte werden in einer eigenen Gruppe zusammengefasst.

# Berechnung der Vorgänger

**Aufgabe:** Welche Vorlesungen müssen besucht werden, um die Vorlesung "Theory of Science" verstehen zu können?

## Tabellen:

- `requires: { predecessor, successor }`
- `course: { courseid, title, ects, taughtBy }`

```
SELECT predecessor
FROM requires, course
WHERE successor = courseid
AND title = 'Theory of Science';
```

## Erläuterung:

- Die Anfrage verbindet die Tabellen `requires` und `course` über die Bedingung `successor = courseid`.
- Anschließend filtert sie nach den Einträgen, bei denen der `title` der Vorlesung in der Tabelle `course` gleich 'Theory of Science' ist.
- Die `SELECT`-Klausel gibt dann den `predecessor` (die vorausgesetzte Vorlesung) für diese Vorlesung aus.

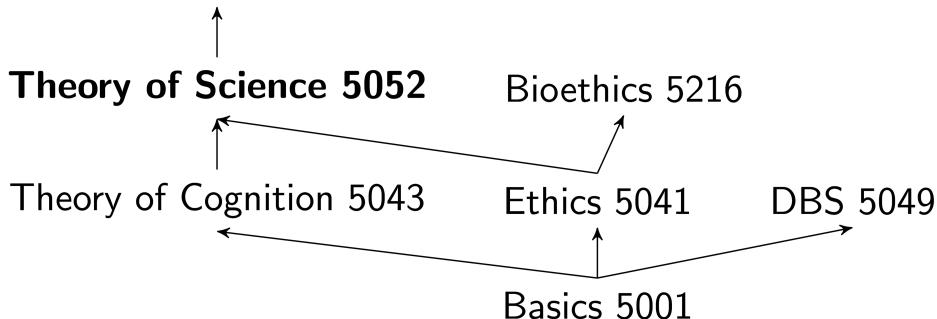
**Wichtiger Hinweis:** Hier werden allerdings nur die **direkten** Vorgänger berechnet. Um alle transitiven Vorgänger (Vorgänger von Vorgängern usw.) zu finden, wären rekursive Anfragen notwendig, die im Standard-SQL nicht direkt unterstützt werden (können aber in einigen Datenbankmanagementsystemen mit speziellen Erweiterungen realisiert werden).

## Beispiel

```
SELECT r2.predecessor
FROM course c, requires r1, requires r2
WHERE c.title='Theory of Science' AND
      c.courseid = r1.successor AND
      r1.predecessor = r2.successor;
```

Aus welchen Vorlesungen besteht das Ergebnis dieser Anfrage?

Advanced Algorithms 5259

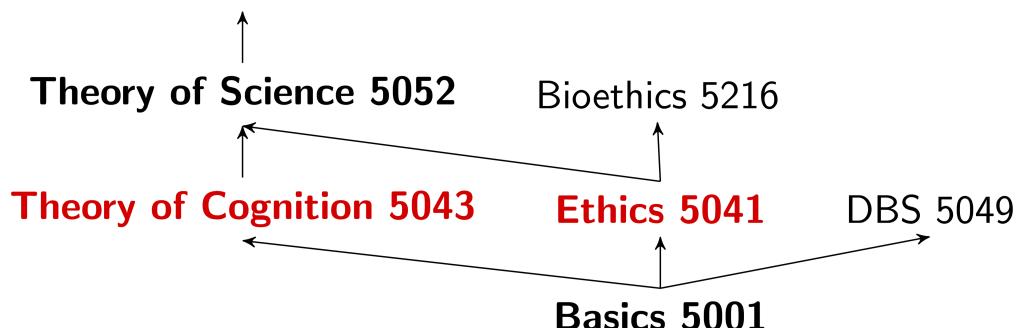


SELECT r2.predecessor

```
FROM course c, requires r1, requires r2
WHERE c.title='Theory of Science' AND
      c.courseid = r1.successor AND
      r1.predecessor = r2.successor;
```

Aus welchen Vorlesungen besteht das Ergebnis dieser Anfrage?

Advanced Algorithms 5259



## Vorgänger der Tiefe n

```

SELECT r1.predecessor
FROM requires r1,
     ...,
     requires rn_minus_1,
     requires rn,
     course c
WHERE r1.successor = r2.predecessor AND
      ... AND
      rn_minus_1.successor = rn.predecessor AND
      rn.successor = c.courseid AND
      c.title = 'Theory of Science';
  
```

Wie berechnet man die ganze Liste von Vorgängern beliebiger Tiefe?

Tiefe 1 UNION Tiefe 2 UNION Tiefe 3 UNION ...

---

# Rekursion in SQL

## Beispiel 1: Summe der Zahlen von 1 bis 100

```
WITH RECURSIVE mytable(number) AS (
    VALUES(1)          -- nicht rekursiver Teil (Basisfall)
    UNION ALL
    SELECT number + 1  -- rekursiver Teil
    FROM mytable
    WHERE number < 100   -- Abbruchbedingung
)
SELECT sum(number)      -- eigentliche Anfrage
FROM mytable;

-- Result: 5050 (Summe der Zahlen von 1 bis 100)
```

- `WITH RECURSIVE mytable(number) AS (...)`: Definiert eine Common Table Expression (CTE) namens `mytable` mit einer Spalte `number`, die rekursiv aufgebaut wird.
- `VALUES(1)`: Der nicht-rekursive Teil initialisiert die CTE mit dem Startwert 1.
- `UNION ALL`: Kombiniert den nicht-rekursiven und den rekursiven Teil.
- `SELECT number + 1 FROM mytable WHERE number < 100`: Der rekursive Teil greift auf die vorherige Iteration von `mytable` zu und erzeugt den nächsten Wert (`number + 1`), solange die Bedingung `number < 100` erfüllt ist. Die CTE referenziert sich hier selbst.
- `SELECT sum(number) FROM mytable`: Die eigentliche Anfrage summiert alle Werte, die in der rekursiv erzeugten CTE `mytable` enthalten sind.

## Anfrage für Voraussetzungen von Kursen

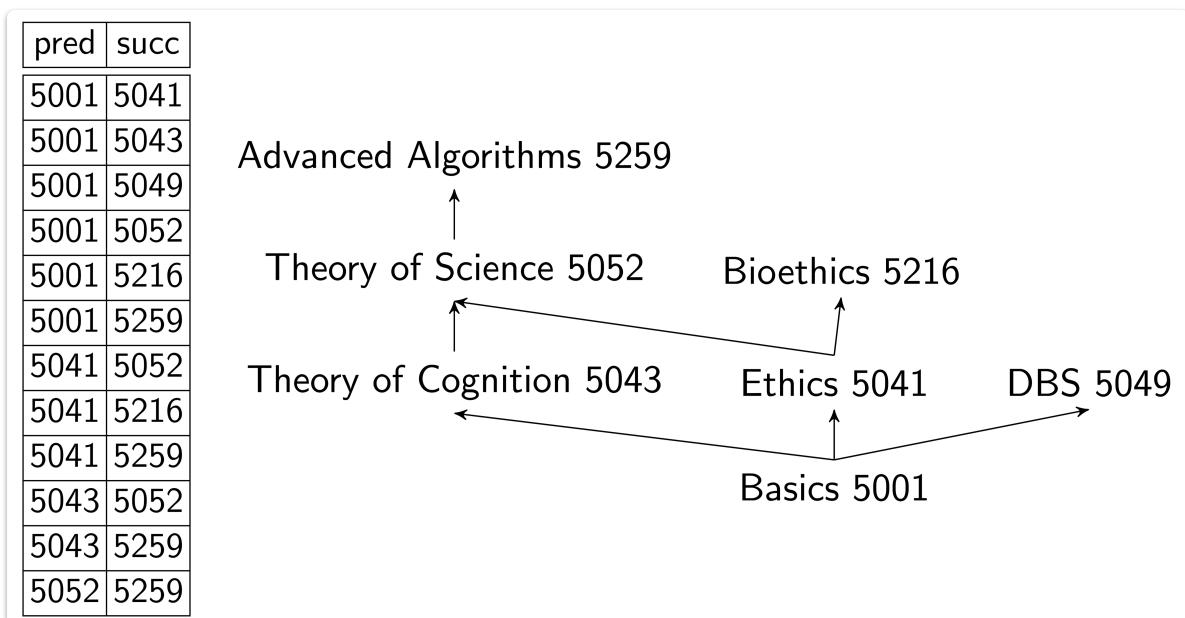
### Beispiel 2: Transitive Voraussetzungen von Kursen

```
WITH RECURSIVE transitiveCourse (pred, succ) AS (
    SELECT predecessor, successor
    FROM requires
    UNION ALL
    SELECT DISTINCT tc.pred, r.successor
    FROM transitiveCourse tc, requires r
    WHERE tc.succ = r.predecessor
)
SELECT *
FROM transitiveCourse
ORDER BY (pred, succ) ASC;
```

- `WITH RECURSIVE transitiveCourse (pred, succ) AS (...)`: Definiert eine rekursive CTE namens `transitiveCourse` mit den Spalten `pred` (Voraussetzung) und `succ` (Nachfolger).

- `SELECT predecessor, successor FROM requires` : Der nicht-rekursive Teil wählt die direkten Voraussetzungen aus der Tabelle `requires`.
- `UNION ALL` : Kombiniert die direkten und die transitiven Voraussetzungen.
- `SELECT DISTINCT tc.pred, r.successor FROM transitiveCourse tc, requires r WHERE tc.succ = r.predecessor` : Der rekursive Teil verbindet die bereits gefundenen transitiven Voraussetzungen (`tc`) mit den direkten Voraussetzungen (`r`), um weitere transitive Beziehungen zu finden. Wenn der Nachfolger einer bereits bekannten Voraussetzung (`tc.succ`) der Vorgänger einer anderen Vorlesung (`r.predecessor`) ist, dann ist die ursprüngliche Voraussetzung (`tc.pred`) auch eine transitive Voraussetzung für den Nachfolger (`r.successor`). `DISTINCT` wird verwendet, um Duplikate zu vermeiden.
- `SELECT * FROM transitiveCourse ORDER BY (pred, succ) ASC` : Die eigentliche Anfrage wählt alle transitiven Voraussetzungen aus der CTE aus und sortiert das Ergebnis.

Dieses Beispiel zeigt, wie Rekursion verwendet werden kann, um transitive Beziehungen in hierarchischen Daten (wie Kursvoraussetzungen) zu ermitteln.



## Unendliche Rekursion verhindern

1. Die meisten DBMS beschränken die Rekursionstiefe mit einem Parameter.
2. Direkt in der Anfrage kodieren.

Beispiel zur Verhinderung unendlicher Rekursion durch Tiefenbegrenzung:

```

WITH RECURSIVE transitiveCourse (pred, succ, depth) AS (
    SELECT predecessor, successor, 0
    FROM requires
    UNION ALL
    SELECT DISTINCT tc.pred, r.successor, tc.depth + 1
    FROM transitiveCourse tc, requires r
    WHERE tc.succ = r.predecessor
    AND tc.depth < 1 -- Begrenzung der Rekursionstiefe
  )
  
```

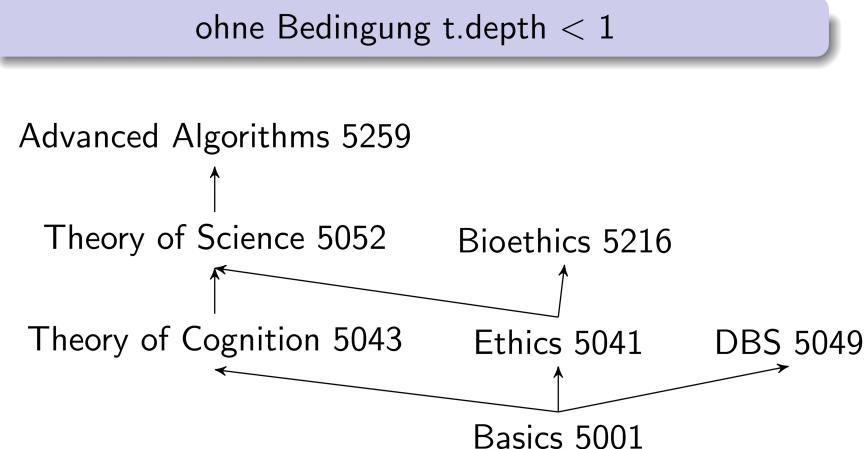
```
)
SELECT *
FROM transitiveCourse
ORDER BY (pred, succ) ASC;
```

- Im Vergleich zum vorherigen Beispiel wurde eine Spalte `depth` zur CTE `transitiveCourse` hinzugefügt, um die aktuelle Rekursionstiefe zu verfolgen.
- Im nicht-rekursiven Teil wird die Tiefe auf `0` initialisiert.
- Im rekursiven Teil wird die Tiefe bei jedem Schritt um `1` erhöht (`tc.depth + 1`).
- Die `WHERE`-Klausel im rekursiven Teil enthält nun die Bedingung `AND tc.depth < 1`. Dies begrenzt die Rekursionstiefe auf maximal 1. Sobald `tc.depth` den Wert 1 erreicht, wird der rekursive Schritt nicht mehr ausgeführt, wodurch eine unendliche Schleife verhindert wird.

**Hinweis:** Der Wert `< 1` in diesem Beispiel ist eine sehr strenge Begrenzung und würde nur direkte Voraussetzungen und deren unmittelbare Voraussetzungen berücksichtigen. In realen Szenarien wäre eine höhere Tiefenbegrenzung oder eine andere Abbruchlogik erforderlich, abhängig von der Struktur der Daten. Die meisten DBMS erlauben auch eine Konfiguration der maximalen Rekursionstiefe auf Systemebene.

### Ohne der Bedingung:

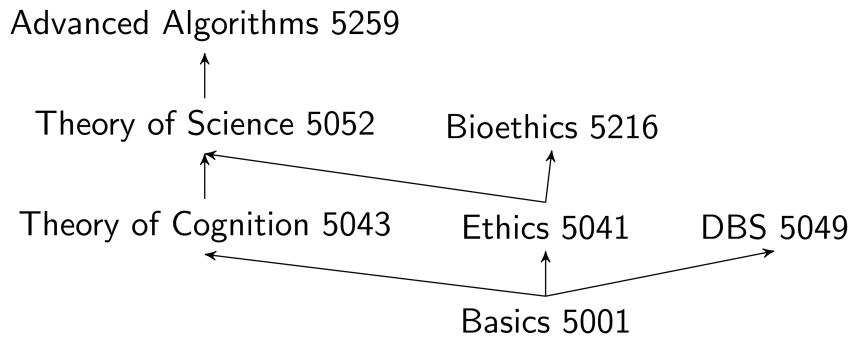
pred	succ	depth
5001	5041	0
5001	5043	0
5001	5049	0
5001	5052	1
5001	5216	1
5001	5259	2
5041	5052	0
5041	5216	0
5041	5259	1
5043	5052	0
5043	5259	1
5052	5259	0



### Mit der Bedingung:

pred	succ	depth
5001	5041	0
5001	5043	0
5001	5049	0
5001	5052	1
5001	5216	1
5001	5259	2
5041	5052	0
5041	5216	0
5041	5259	1
5043	5052	0
5043	5259	1
5052	5259	0

mit Bedingung  $t.\text{depth} < 1$



## Noch ein Beispiel von Rekursion

Alle Voraussetzungen einer bestimmten Vorlesung

```

WITH RECURSIVE transitiveCourse (pred, succ) AS (
    SELECT predecessor, successor
    FROM requires
    UNION ALL
    SELECT DISTINCT tc.pred, r.successor
    FROM transitiveCourse tc, requires r
    WHERE tc.succ = r.predecessor
)
SELECT c2.title
FROM transitiveCourse tv, course c1, course c2
WHERE tv.succ = c1.courseid
AND c1.title = 'Theory of Science'
AND tv.pred = c2.courseid;
  
```

Was genau ist das Resultat dieser Anfrage?

Das Resultat dieser Anfrage sind die **Titel aller Vorlesungen, die (direkte oder indirekte) Voraussetzungen für die Vorlesung "Theory of Science" sind.**

Erläuterung der Anfrage:

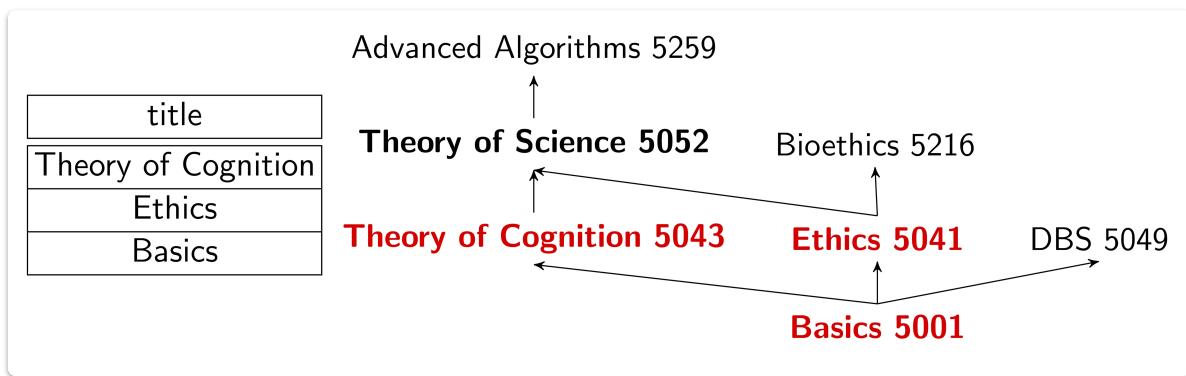
### 1. Rekursive CTE `transitiveCourse`:

- Berechnet alle transitiven Beziehungen zwischen Voraussetzungen (`pred`) und Nachfolgern (`succ`).
- Der nicht-rekursive Teil wählt die direkten Voraussetzungen aus der Tabelle `requires`.
- Der rekursive Teil findet weitere Voraussetzungen, indem er bereits gefundene (`tc`) mit direkten Voraussetzungen (`r`) verknüpft.

### 2. Hauptabfrage:

- Verbindet die rekursive CTE `transitiveCourse` (alias `tv`) mit der Tabelle `course` zweimal (alias `c1` und `c2`).
- `tv.succ = c1.courseid`: Verbindet den Nachfolger (`succ`) aus der transitiven Voraussetzungsliste mit der `courseid` der Vorlesung, für die wir die Voraussetzungen suchen (`c1`).
- `c1.title = 'Theory of Science'`: Filtert das Ergebnis, sodass wir nur die Voraussetzungen für die Vorlesung mit dem Titel 'Theory of Science' betrachten.
- `tv.pred = c2.courseid`: Verbindet die Voraussetzung (`pred`) aus der transitiven Voraussetzungsliste mit der `courseid` einer anderen Vorlesung (`c2`).
- `SELECT c2.title`: Gibt den `title` (`c2.title`) dieser vorausgesetzten Vorlesung aus.

Zusammenfassend liefert die Anfrage eine Liste der Titel aller Kurse, die in der Kette der Voraussetzungen irgendwann vor der Vorlesung "Theory of Science" stehen.



## Anzahl der Ergebnisse begrenzen

Um die Anzahl der Ausgaben zu begrenzen gibt es **verschiedene Möglichkeiten**

### SQL 2003: Window Function

```
SELECT *
FROM (SELECT
    ROW_NUMBER() OVER (ORDER BY semester DESC) AS number,
    studid, name, semester
    FROM student
) AS studTmp
WHERE number <= 5;
```

```
SELECT *
FROM (SELECT
    RANK() OVER (ORDER BY semester DESC) AS number,
    studid, name, semester
    FROM student
) AS studTmp
WHERE number <= 5;
```

### SQL 2008: Fetch-First-Klausel

```
SELECT *
FROM student
ORDER BY semester DESC
FETCH FIRST 5 ROWS ONLY;
```

Unterstützt von IBM DB2, Sybase SQL Anywhere, PostgreSQL,...

### Nicht-Standard-Syntax

```
SELECT * FROM student LIMIT 5;
```

Unterstützt von MySQL, Sybase SQL Anywhere, PostgreSQL,...

```
SELECT * FROM student WHERE ROWNUM <= 5;
```

Unterstützt von Oracle

```
SELECT TOP 5 FROM student;
```

Unterstützt von MS SQL server

**LIMIT** wird in der Prüfung akzeptiert.



# Abstraktionsebenen eines Datenbanksystems

Änderungen auf der logischen Ebene haben keine Auswirkungen auf externe Schemata und Anwendungsprogramme.



Datenunabhängigkeit:

- **Physische Unabhängigkeit**: Änderungen auf der physischen Ebene haben keinen Einfluss auf die logische Ebene.
- **Logische Datenunabhängigkeit**: Änderungen auf der logische Ebene haben keinen Einfluss auf die Sichten (Views).

Mehr zu **Abstraktion** siehe: [EP2](#)

## Beispiele für Views:

### Beispiel 1: View `profsAndtheirCourses`

```

CREATE VIEW profsAndtheirCourses AS
SELECT c.title, p.name
FROM professor p, course c
WHERE p.empid = c.taughtBy;

SELECT * FROM profsAndtheirCourses;
  
```

- `CREATE VIEW profsAndtheirCourses AS ...`: Erstellt eine virtuelle Tabelle (View) namens `profsAndtheirCourses`.
- Die Definition der View (`SELECT c.title, p.name ...`) wählt den Titel der Kurse (`c.title`) und die Namen der Professoren (`p.name`) aus, indem die Tabellen `professor` und `course` über die `empid` des Professors und `taughtBy` des Kurses verbunden werden.
- Die zweite `SELECT`-Anweisung greift auf die erstellte View zu, als wäre sie eine reguläre Tabelle, und gibt alle Spalten (Titel des Kurses und Name des Professors) aus.

### Beispiel 2: View `ectsPerStud`

```

CREATE VIEW ectsPerStud AS
SELECT s.name, s.studid, SUM(c.ects) AS sum
  
```

```
FROM student s, takes t, course c
WHERE t.courseid = c.courseid AND s.studid = t.studid
GROUP BY s.name, s.studid;

SELECT sum FROM ectsPerStud;
```

- `CREATE VIEW ectsPerStud AS ...`: Erstellt eine View namens `ectsPerStud`.
- Die Definition der View berechnet die Summe der ECTS-Punkte (`SUM(c.ects)`) für jeden Studenten. Dazu werden die Tabellen `student`, `takes` und `course` verbunden und nach dem Namen und der `studid` der Studenten gruppiert. Das Ergebnis der Summe wird als `sum` aliasiert.
- Die zweite `SELECT`-Anweisung greift auf die View zu und wählt die Spalte `sum` (die Summe der ECTS-Punkte pro Student) aus.

**Views/Sichten können verwendet werden, um abgeleitete Attribute darzustellen (ER Diagramm).**

- Views ermöglichen es, komplexe Abfragen zu kapseln und als einfache virtuelle Tabellen darzustellen.
- Dies kann die Datenmodellierung vereinfachen, indem abgeleitete Informationen (wie die Summe der ECTS pro Student oder die Liste der Kurse pro Professor) als scheinbar eigenständige Attribute oder Relationen dargestellt werden.
- Dies kann auch die Wiederverwendbarkeit von Abfragen und die Lesbarkeit von SQL-Code verbessern.

## Views ändern

`CREATE OR REPLACE VIEW :`

```
CREATE OR REPLACE VIEW profsAndtheirCourses AS
SELECT c.title, p.name
FROM professor p, course c
WHERE p.empid = c.taughtBy;
```

- Ermöglicht das Ändern einer bestehenden View oder das Erstellen einer neuen View, falls noch keine mit dem angegebenen Namen existiert.
- **Wichtig:** `REPLACE VIEW` erwartet dieselben Spalten in derselben Reihenfolge mit denselben Datentypen wie die ursprüngliche View. Andernfalls kann es zu Fehlern oder unerwartetem Verhalten kommen.

`DROP VIEW :`

```
DROP VIEW ectsPerStud;
CREATE VIEW ectsPerStud AS
SELECT s.name, s.studid, SUM(c.ects) AS sum
```

```
FROM student s, takes t, course c
WHERE t.courseid = c.courseid
  AND s.studid = t.studid
GROUP BY s.name, s.studid;
```

- `DROP VIEW ectsPerStud;` : Entfernt die existierende View namens `ectsPerStud`.
- Anschließend wird die View mit einer neuen Definition neu erstellt. Dies ist eine Möglichkeit, eine View zu ändern, wenn die Struktur sich signifikant ändert.

### Alternative:

- Die Views erst löschen (`DROP VIEW`), dann neu erstellen (`CREATE VIEW`).
  - SQL-Erweiterungen, z.B. `ALTER VIEW` von PostgreSQL, bieten eine direktere Möglichkeit, die Definition einer View zu ändern, ohne sie vorher löschen zu müssen. Allerdings ist `ALTER VIEW` nicht Teil des Standard-SQL und wird nicht von allen Datenbankmanagementsystemen unterstützt.
-

# Sichten vs. materialisierte Sichten

---

## (Dynamische) Sicht (View)

- Entspricht einem Makro für eine Anfrage.
- Ergebnis der Anfrage wird nicht vorberechnet, sondern erst dann, wenn die Sicht benutzt wird.
- **Vorteil:** Die Daten in der Sicht sind immer aktuell, da die zugrundeliegende Abfrage bei jedem Zugriff neu ausgeführt wird.
- **Nachteil:** Die Performance kann leiden, insbesondere bei komplexen Sichten, da die Berechnung bei jeder Anfrage wiederholt wird.

## Materialisierte Sicht (Materialized View)

- Ergebnis der Sicht wird vorberechnet und persistent gespeichert (ähnlich einer regulären Tabelle).
- Rechenaufwand bevor irgendeine Anfrage ausgeführt wird (bei der Erstellung und Aktualisierung der materialisierten Sicht).
- **Vorteil:** Deutlich schnellere Abfragezeiten, da das Ergebnis bereits vorliegt.
- **Nachteil:** Die Daten in der materialisierten Sicht sind nicht immer sofort aktuell. Sie müssen explizit aktualisiert werden (manuell oder periodisch), was zusätzlichen Aufwand verursacht.

## Was ist die bessere Wahl? Laufzeit vs. Updates

Die Wahl zwischen einer dynamischen und einer materialisierten Sicht hängt von den spezifischen Anforderungen ab:

- **Dynamische Sicht ist besser, wenn:**
  - Daten häufig geändert werden und die Aktualität der Daten in der Sicht entscheidend ist.
  - Die zugrundeliegende Abfrage relativ einfach ist und die Performance keine kritische Rolle spielt.
  - Speicherplatz begrenzt ist, da keine zusätzlichen Daten gespeichert werden müssen.
- **Materialisierte Sicht ist besser, wenn:**
  - Die Performance von Abfragen auf die Sicht kritisch ist (z.B. bei häufig genutzten, komplexen Berichten).
  - Die Daten nicht sehr häufig geändert werden oder eine gewisse Latenz bei der Aktualisierung akzeptabel ist.
  - Ausreichend Speicherplatz für die Speicherung der vorberechneten Daten vorhanden ist.

Oft ist es ein Trade-off zwischen der Aktualität der Daten (dynamische Sicht) und der Performance der Abfragen (materialisierte Sicht).

## Updates und Views

```
CREATE VIEW howTough AS
SELECT empid, AVG(grade) AS avgGrade
FROM grades
GROUP BY empid;

UPDATE howTough
SET avgGrade = 1.0
WHERE empid = ( SELECT empid
                 FROM professor
                 WHERE name = 'Socrates');
```

Was ist hier das Problem?

- Problem: Wo speichere ich das?
- Wie sollen die Noten in der Ursprungstabelle verändert werden?

## Noch ein Beispiel:

```
CREATE VIEW courseView AS
SELECT title, ects, name
FROM course, professor
WHERE taughtBy = empid;

INSERT INTO courseView
VALUES ('Nihilism', '2', 'Nobody');
```

Tabellen:

- course: { courseid, title, ects, taughtBy }
- professor: { empid, name, rank, office }

Was ist das Problem? Welche Tupel sollen in die Ursprungstabellen eingefügt werden?

Das Problem bei diesem `INSERT`-Befehl auf die View `courseView` ist, dass die View auf einer Verknüpfung zweier Tabellen (`course` und `professor`) basiert und nicht alle Spalten der Basistabellen in der View enthalten sind. Insbesondere fehlt die Information, wie die eingefügten Werte den Primär- und Fremdschlüsseln der Basistabellen zugeordnet werden sollen.

## Änderbarkeit von Sichten in SQL

Daten in einer View sind veränderbar (update/insert/delete), wenn ...

in SQL-92

- nur eine Tabelle
- nur Selektion und Projektion
- keine Aggregatfunktionen, Gruppierung und Duplikateliminierung

in SQL-99

- wie SQL-92
- mehrere Tabellen und Attribute, wenn diese mit Hilfe des Primärschlüssels eindeutig zugeordnet werden können
- D.h. auch Views mit Joins können manchmal geändert werden

## Änderbarkeit von Views in SQL

Views in Postgres sind nicht materialisiert!

```
CREATE VIEW stud AS
  SELECT *
  FROM student;
```

```
INSERT INTO stud VALUES (42, 'mueller', 11);
```

**ERROR: cannot insert into a view.**

**HINT: You need an unconditional ON INSERT DO INSTEAD rule.**

- Views in Postgres sind nicht änderbar!
- Aber: Jede View kann einzeln mit Hilfe von Regeln (rules) „freigeschaltet“ werden.

# Integritätsbedingungen

- Zusätzliches Instrument, um Inkonsistenzen zu verhindern.
- Ziel: Das Einfügen inkonsistenter Daten verhindern.

Welche Integritätsbedingungen haben wir in der Vorlesung schon kennengelernt?

- Keine zwei Tupel haben dieselben Werte in den Schlüsselattributen (Primärschlüsselbedingung).
- Kardinalitäten/Stelligkeit von Beziehungstypen (definiert die Anzahl der Beziehungen, die eine Entität eingehen kann).
- Generalisierung: jedes Entity vom Subtyp muss im Obertyp enthalten sein (Totalitätsbedingung bei Spezialisierung).
- Domänen (d.h. zulässige Werte) von Attributen (legt fest, welche Werte für ein Attribut gültig sind).
- Primärschlüssel und Fremdschlüssel (Primärschlüssel identifiziert Tupel eindeutig, Fremdschlüssel stellt Beziehungen zwischen Tabellen sicher und referenziert existierende Primärschlüsselwerte).
- NOT NULL, UNIQUE (Constraints, die sicherstellen, dass ein Attribut keine Nullwerte enthält bzw. dass alle Werte in einem Attribut eindeutig sind).

## Statische Integritätsbedingungen

Statische Integritätsbedingungen müssen von jedem Zustand der Datenbank erfüllt werden.

Beispiele:

- NOT NULL -Constraint:

```
CREATE TABLE professor (
    ...
    empid INTEGER NOT NULL,
    ...
);
```

- Stellt sicher, dass die Spalte `empid` in der Tabelle `professor` keine Nullwerte enthalten darf.

- CHECK -Constraint (Einschränkung des Wertebereichs):

```
CREATE TABLE student (
    ...
    semester INTEGER CHECK (semester BETWEEN 1 AND 20),
    ...
);
```

- Stellt sicher, dass der Wert der Spalte `semester` in der Tabelle `student` zwischen 1 und 20 (inklusive) liegen muss.
- **CHECK -Constraint (Aufzählungstypen):**

```
CREATE TABLE professor (
    ...
    rank VARCHAR(2) CHECK (rank IN ('C2', 'C3', 'C4')),
    ...
);
```

- Stellt sicher, dass der Wert der Spalte `rank` in der Tabelle `professor` einer der Werte 'C2', 'C3' oder 'C4' sein muss.
- **CREATE DOMAIN (Festlegung eines benutzerdefinierten Wertebereichs):**

```
CREATE DOMAIN wineColor varchar(5)
DEFAULT 'red'
CHECK (VALUE IN ('red', 'white', 'rose'));

CREATE TABLE wine (
    wineID INT PRIMARY KEY,
    name VARCHAR(20) NOT NULL,
    color wineColor,
    ...
);
```

- `CREATE DOMAIN wineColor ...` : Definiert einen benutzerdefinierten Datentyp namens `wineColor`, der ein `VARCHAR(5)` ist und standardmäßig den Wert 'red' hat. Ein `CHECK`-Constraint stellt sicher, dass nur die Werte 'red', 'white' oder 'rose' für diesen Datentyp zulässig sind.
- In der Tabelle `wine` wird die Spalte `color` mit dem benutzerdefinierten Datentyp `wineColor` deklariert, wodurch die für diesen Datentyp definierten Einschränkungen automatisch angewendet werden.

## Dynamische Integritätsbedingungen

### Referentielle Integrität

Referentielle Integrität bedeutet

Fremdschlüssel müssen auf existierende Tupel verweisen oder einen Nullwert enthalten.

Was passiert, wenn kein:e Professor:in mit empid 007 existiert?

```
insert into course
values (5100, 'Spying for Dummies', 4, 007);
```

Wie kann dieses „insert“ verhindert werden?

## Festlegung von Schlüsseln

Kandidatenschlüssel

- UNIQUE
- Mehrere Kandidatenschlüssel für eine Relation sind möglich
- Darf null sein!

Primärschlüssel

- PRIMARY KEY
- Nur ein Primärschlüssel pro Relation
- **Impliziert UNIQUE NOT NULL**

Fremdschlüssel

- FOREIGN KEY
- null ist möglich, kann aber mit NOT NULL verhindert werden

## Verhalten bei Änderungsoperationen

Dynamische Integritätsbedingungen müssen von Zustandsänderungen erfüllt werden.

Bei Änderung von referenzierten Daten haben wir mindestens 3 Optionen

- Zurückweisen der Änderungsoperation (Default-Verhalten)
- Propagieren der Änderungen (**CASCADE**)
- Verweise auf „unbekannt“ setzen: **set null**

In Postgres außerdem

- Auf Defaultwert setzen (SET DEFAULT)

## Beispiel

Update on table R

S		R	
	$\alpha$	$\kappa$	
	$\kappa_1$	$\kappa_1$	
	$\kappa_2$	$\kappa_2$	
...	...	...	...

UPDATE R

SET  $\kappa = \kappa'_1$   
WHERE  $\kappa = \kappa_1$

DELETE FROM R

WHERE  $\kappa = \kappa_1$

Wie soll damit umgegangen werden?

Da gibts mehrere Optionen:

### Option 1

CREATE TABLE S

(...,  
 $\alpha$  integer REFERENCES R( $\kappa$ ));

S		R	
	$\alpha$	$\kappa$	
	$\kappa_1$	$\kappa_1$	
	$\kappa_2$	$\kappa_2$	
...	...	...	...

Ergebnis der Änderungsoperation: DB bleibt unverändert

### Option 2

CREATE TABLE S

(...,  
 $\alpha$  integer REFERENCES R( $\kappa$ )  
ON UPDATE CASCADE);

S		R	
	$\alpha$		$\underline{\kappa}$
	$\kappa'_1$		$\kappa'_1$
	$\kappa_2$		$\kappa_2$
...	...	...	...

CREATE TABLE S

(...,  
 $\alpha$  integer REFERENCES R( $\kappa$ )  
ON DELETE CASCADE);

S		R	
	$\alpha$		$\underline{\kappa}$
	$\kappa_2$		$\kappa_2$
...	...	...	...

Ergebnis der Änderungsoperation:  
Schlüssel in R und S geändert.

Ergebnis der Änderungsoperation:  
Schlüssel in R und S gelöscht.

### Option 3:

CREATE TABLE S

(...,  
 $\alpha$  integer REFERENCES R( $\kappa$ )  
ON UPDATE SET NULL);

S		R	
	$\alpha$		$\underline{\kappa}$
	null		$\kappa'_1$
	$\kappa_2$		$\kappa_2$
...	...	...	...

Ergebnis der Änderungsoperation:  
Tupel aus R auf  $\kappa'_1$ , in S auf null  
geändert.

CREATE TABLE S

(...,  
 $\alpha$  integer REFERENCES R( $\kappa$ )  
ON DELETE SET NULL);

S		R	
	$\alpha$		$\underline{\kappa}$
	null		...
	$\kappa_2$		$\kappa_2$
...	...	...	...

Ergebnis der Änderungsoperation:  
Tupel aus R gelöscht, Schlüssel in S  
auf null geändert.

### Kaskadierendes Löschen

Referentielle Integrität: ON DELETE CASCADE

```

CREATE TABLE course (
    ...
    taughtBy INTEGER REFERENCES professor(epid)
        ON DELETE CASCADE
);

CREATE TABLE takes (
    ...
    courseid INTEGER REFERENCES course(courseid)
        ON DELETE CASCADE
);

```

```
DELETE FROM professor WHERE name = 'Socrates';
```

Beispiel siehe Slides: [hier](#)

## Komplexe Konsistenzbedingung

```
CREATE TABLE grades (
    studid      integer REFERENCES student ON DELETE CASCADE,
    courseid    integer REFERENCES course,
    grade       numeric(2,1) CHECK (grade BETWEEN 0.7 AND 5.0),
    PRIMARY KEY(studid, courseid)
    CONSTRAINT hasTaken
        CHECK (EXISTS (SELECT *
                      FROM takes h
                     WHERE h.courseid = grades.courseid AND
                           h.studid = grades.studid))
);
```

- Die CHECK-Klausel wird für jedes UPDATE und INSERT ausgeführt.
- Operation wird zurückgewiesen, wenn der Ausdruck zu false evaluiert!  
True und unknown widersprechen der Bedingung nicht!

- PostgreSQL unterstützt keine CHECK-Constraints, die andere Tabellen involvieren.
- Workaround durch die Verwendung von **Triggern**

# Zusammenfassung

---

- SQL ist mehr als eine Anfragesprache.
- DDL, DML, DCL, TCL, DQL.
- SQL (DQL) ist eine deklarative Anfragesprache.
- SFW-Block ist die Grundlage.
- SQL bietet Operationen an, aus relationaler Algebra bekannt.
- Duplikate.
- Komplexe Bedingungen (=, <>, !=), >, <, >=, <=, LIKE, BETWEEN, IN, AND, OR, NOT, IS, ...).
- Geschachtelte Anfragen:
  - Korreliert und unkorreliert.
  - `IN`, `EXISTS`, `ANY`, `ALL`.
- Gruppierung und Aggregation:
  - `SELECT`-Klausel kann nur Spalten referenzieren, die in der `GROUP BY`-Klausel vorkommen oder in Aggregatfunktionen.
  - `HAVING`, um gewisse Gruppen zu selektieren.
- Temporäre Tabellen (`WITH`).
- Null-Werte führen möglicherweise zu unerwarteten Ergebnissen.
- Rekursive Anfragen.
- Größe des Ergebnisses beschränken (`LIMIT`, `FETCH FIRST n ROWS ONLY`, ...).
- Views können wie „Makros“ verwendet werden.
- Materialisierte Sichten für schnellere Anfragebearbeitung.
- Integritätsbedingungen, um Inkonsistenzen in den Daten zu vermeiden.
- `CASCADE` vorsichtig verwenden.
- Trigger sind mächtiger als `CHECK`-Bedingungen aber auch „gefährlicher“.