DATENBANKEN – LINUS STUHLMANN

RELATIONALE ALGEBRA

- P = Prädikat (Bedingung)
- σ = Selektion (Operator)
- {=, <, >, ≤, ≥, ≠}

σ – Selektion

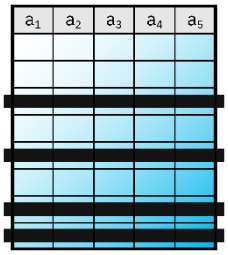
- R' = $\sigma_P(R)$ → R' = $\sigma_{Preis<30}$ (Customers)

Filme

Titel	Jahr	Länge	inFarbe	Studio	ProduzentID
Total Recall	1990	113	True	Fox	12345
Basic Instinct	1992	127	True	Disney	67890
Dead Man	1995	90	False	Paramount	99999

$\sigma_{Länge \geq 100}$(Filme)

Titel	Jahr	Länge	inFarbe	Studio	ProduzentID
Total Recall	1990	113	True	Fox	12345
Basic Instinct	1992	127	True	Disney	67890



π – Projektion

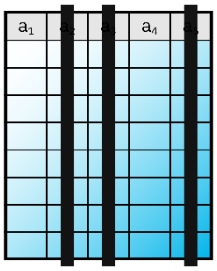
- R' = $\pi_L(R)$ → R' = $\pi_{InFarbe}(R)$

$\pi_{Titel,Jahr,Länge}$(Filme)

Titel	Jahr	Länge
Total Recall	1990	113
Basic Instinct	1992	127
Dead Man	1995	121

$\pi_{inFarbe}$(Filme)

inFarbe
True
False



ρ – Umbenennung

- $\rho_{neuerName}(R)$ → die Relation R erhält somit den Namen „neuerName“
- $\rho_{R(x,y,z)}(R(A,B,C)$ → die Attribute der Relation R haben jetzt die Namen A, B & C

x – Kreuzprodukt

- $R' = R \times S$
- Jede Zeile von R wird mit jeder Zeile von S kombiniert
- Zeilen = Anzahl Zeilen R * Anzahl Zeilen S; Anz. Spalten = Spalten R + Spalten S

R

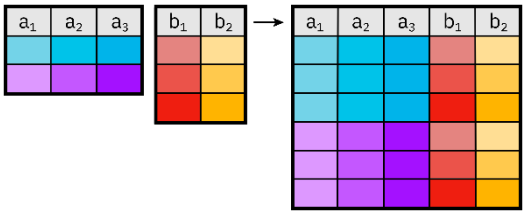
A	B
1	2
3	4

S

B	C	D
2	5	6
4	7	8
9	10	11

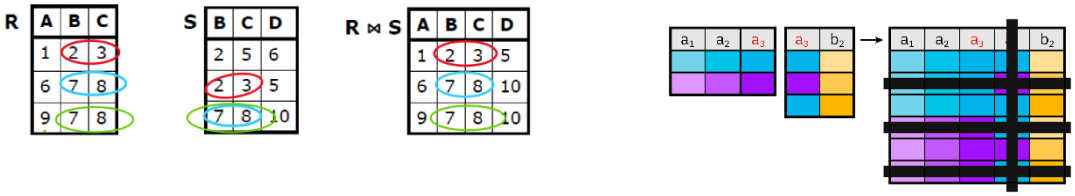
R × S

A	R.B	S.B	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11



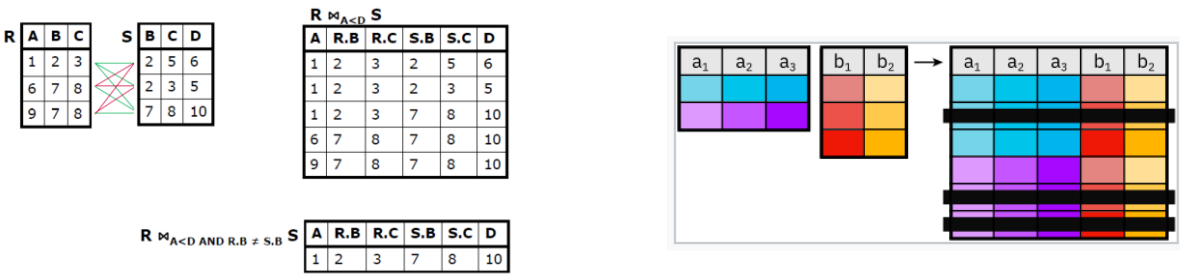
⋈ - Natural Join

- $R' = R \bowtie S$
- Es werden alle Tupel bzw. Zeilen miteinander kombiniert, deren gemeinsamen Attribute übereinstimmen. **Duplikate** werden gelöscht.
- Natural Join mit derselben Tabelle ist immer dieselbe Tabelle
- Gibt es **keine gemeinsamen Attribute** → Kreuzprodukt



$\bowtie_P$ - Theta-Join

- Bildet das Kreuzprodukt
- Selektiere mittels einer Joinbedingung
  - Alle Tupel die Bedingung nicht erfüllen – löschen
- $R \bowtie_p S = \sigma_p (R \times S)$
- P: Join Prädikat: Beliebiger logischer Ausdruck
- Joint nur mit Tupel, wenn die Bedingung erfüllt ist.
- Alle Attribute werden zusammengesetzt



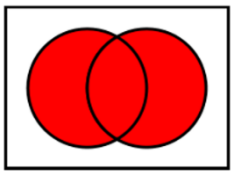
MENGENOPERATOREN

Um Mengenoperationen auf Relationen anwenden zu können, müssen die **Relationen dasselbe Schema** und denselben Wertebereich aufweisen. Es gibt in der Mengensemantik **keine Duplikate**.

∪ - Vereinigung (Mengenoperator)

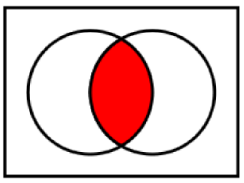
Sammelt Elemente (Tupel) zweier Relationen unter einem **gemeinsamen** Schema auf.

- Sie müssen **dasselbe Schema** (gleiche Attribute) haben.
- R ∪ S = R'
- R' sind alle Elemente, die **in R oder in S** vorkommen
- Resultat R' hat dasselbe Schema wie R



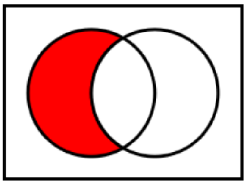
∩ - Schnittmenge bzw. Durchschnitt (Mengenoperator)

- Sie müssen **dasselbe Schema** (gleiche Attribute) haben.
- R ∩ S = R'
- R' sind alle Elemente, die **in R und in S** vorkommen
- Resultat R' hat dasselbe Schema wie R



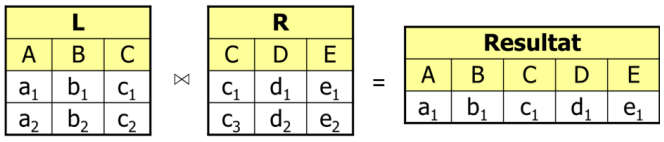
\ - Differenz (Mengenoperator)

- Sie müssen **dasselbe Schema** (gleiche Attribute) haben.
- R \ S = R'
- Differenz R \ S eliminiert die Tupel aus der ersten Relation, die auch in der zweiten Relation vorkommen.
- Resultat R' hat dasselbe Schema wie R



OUTER JOINS

Natural Join



L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

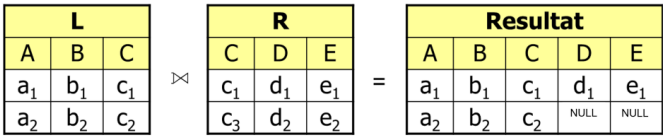
⋈

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

Resultat				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1

Right outer Join



L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

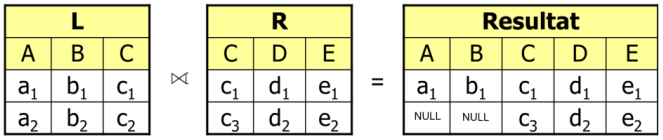
⋈

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

Resultat				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	NULL	NULL

Left outer Join



L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

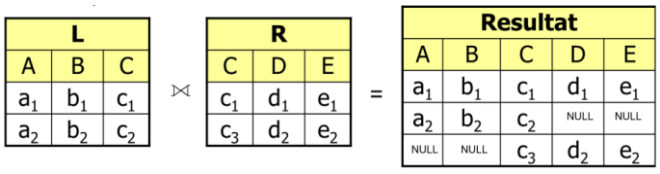
⋈

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

Resultat				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
NULL	NULL	c_3	d_2	e_2

Full outer Join



L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

⋈

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

Resultat				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	NULL	NULL
NULL	NULL	c_3	d_2	e_2

«BAG»-ALGEBRA

DIE BAG-ALGEBRA (MULTIMENGENSEMANTIK) LÄSST DUPLIKATE ZU.

Operation, Symbol	
Selektion, σ	Gleich wie relationale Algebra, Duplikate behandeln wie 'normale' Tupel.
Projektion, π	Wie relationale Algebra, aber Duplikate nicht entfernen.
Kreuzprodukt, x	Gleich wie relationale Algebra, Duplikate behandeln wie 'normale' Tupel.
Joins, ⋈	Gleich wie relationale Algebra, Duplikate behandeln wie 'normale' Tupel.

Operation, Symbol	
Vereinigung, ∪ «bag union»	Gleich wie relationale Algebra, also Tupel aus dem linken und Tupel aus dem rechten Operanden zusammen-führen. Behandlung von Duplikaten: Man zählt im linken und im rechten Operanden die Duplikate (= «Multiplizitäten») und nimmt davon die grössere Anzahl.
Vereinigung, ⊔ «bag concatenation»	Neuer Operator. Auch hier werden Tupel aus dem linken und Tupel aus dem rechten Operanden zusammen-geführt. Behandlung von Duplikaten: Man zählt im linken und im rechten Operanden die Duplikate (= «Multiplizitäten») und nimmt die Summe davon.

Operation, Symbol	
Durchschnitt, ∩	Gleich wie relationale Algebra, also nur Tupel die in beiden Operanden vorkommen, übernehmen. Behandlung von Duplikaten: Man zählt im linken und im rechten Operanden die Duplikate (= «Multiplizitäten») und nimmt davon die kleinere Anzahl.
Differenz, \	Gleich wie relationale Algebra, also nur Tupel die im linken, aber nicht im rechten Operanden vorkommen, übernehmen. Behandlung von Duplikaten: Man zählt im linken und im rechten Operanden die Duplikate (= «Multiplizitäten»). Dann rechnet man die Differenz aus zwischen linker Multiplizität und rechter Multiplizität. Wenn diese positiv ist (links hat es mehr als rechts) dann nimmt man diese Differenz, sonst 0 als Multiplizität.

Operation, Symbol	
Duplikatelimination, δ	Neuer Operator. Entfernt Duplikate.

KONZEPTIONELLES DATENMODELL - ERM

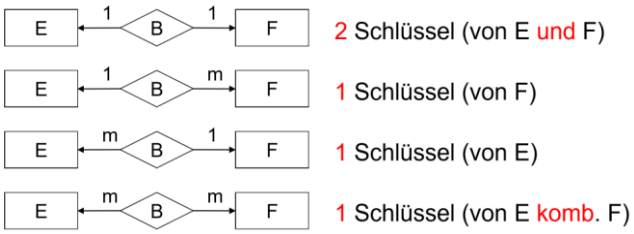
PRIMÄRSCHLÜSSEL

Primärschlüssel, also unterstrichene Attribute werden nur dann vergeben, wenn Pfeile auf den Entitätstypen zeigen. Sobald auf einen Entitätstyp ein Pfeil zeigt, wird der Schlüssel dieser Entität zu einem Primärschlüssel. Dieser muss unterstrichen werden.

SCHLÜSSEL UND BEZIEHUNGEN

- Es werden alle Schlüssel als Attribut der Beziehung abgeschrieben (die Schlüssel {} in Klammern dazugeschrieben)

In diesem Beispiel kann eine Firma zwar «m» Personen angestellt haben, allerdings kann eine Person nur in einer Firma angestellt sein. In diesem Fall braucht es den Schlüssel der Entität «Personen», da «m» Personen in einer Firma arbeiten. Mögliche Kombinationen von Beziehungen sind:



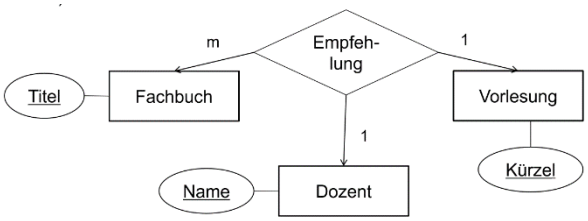
- Bei einer 1:M Beziehung liegt der Schlüssel immer bei dem Entitätstypen mit M, da der andere Typ durch die zu 1 Beziehung eindeutig zugewiesen werden kann.
- Bei einer 1:1 Beziehung muss es von jedem Entitätstypen einen Schlüssel geben.
- Bei einer M:M Beziehung muss ein kombinierter Schlüssel aus beiden erzeugt werden

SONDERFALL 1:1:M, 1:M:M ODER M:M:M

- M:M:M
  - Es gibt einen zusammengesetzten Schlüssel aus drei Attributen
- 1:M:M
  - Es gibt einen zusammengesetzten Schlüssel aus zwei Attributen der Ms
- 1:1:M
  - Es muss zwei Schlüssel geben (die beiden 1 Beziehungen)

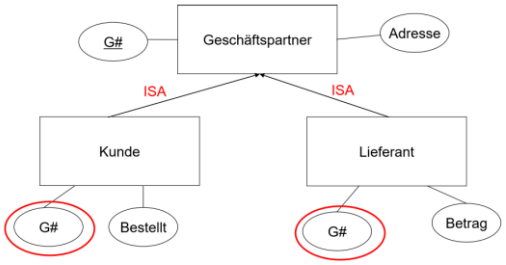
- für Vorlesung {**Titel, Name**}
- für Dozent {**Titel, Kürzel**}

- 1:1:1
  - 3 Zusammengesetzte Schlüssel
  - Alle Kombinationen
  - {A1; A2}, {A2; A3}, {A1; A3}



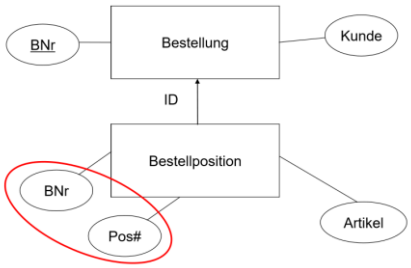
ABHÄNGIGKEITEN

ISA ABHÄNGIGER ENTITÄTSTYP



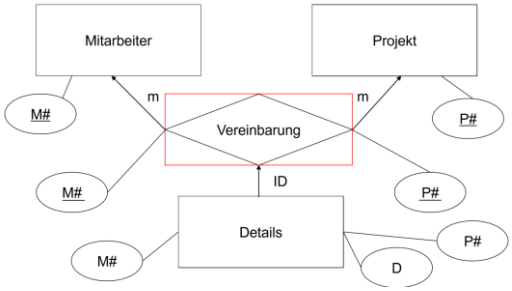
- erben alle Attribute von den Parent-Entitäten
- Übernehmen Primärschlüssel von Parent
- ist eine abhängige 1:1 Beziehung

ID-ABHÄNGIGER ENTITÄTSTYP



- Existiert nur mit Parent
- **Übernimmt Schlüssel** von Parent + **zusätzlichen Schlüssel**
- **Nur Primärschlüssel, wenn Pfeil drauf zeigt**

ZUSAMMENGESETZTER ENTITÄTSTYP



- Um eine Entität mit einer Beziehung in Verbindung zu setzen
- Schlüssel werden nach dem Beziehungsprinzip übernommen
    - Als Primärschlüssel

SQL – DDL (DATA DEFINITION LANGUAGE)

Grundsätzlich gibt es drei DDL-Anweisungen:

1. CREATE
2. ALTER
3. DROP

Sinnvolle Reihenfolge von Befehlen bei Neuerzeugung:

- `DROP SCHEMA IF EXISTS Name_der_DB CASCADE;`
- `CREATE SCHEMA Name_der_DB;`
- `SET SEARCH_PATH TO Name_der_DB;`

DDL-DATENTYPEN

- INTEGER: Ein ganzzahliger Wert, der für numerische Daten verwendet wird.
- CHAR(): Ein festlängiger Text-Datentyp, der eine feste Anzahl von Zeichen speichern kann.
- VARCHAR(): Ein variablelängiger Text-Datentyp, der eine variable Anzahl von Zeichen speichern kann.
- SERIAL: Ein automatisch inkrementierender ganzzahliger Wert, der oft als Primärschlüssel verwendet wird.
- DATE: Ein Datum im Format JJJJ-MM-TT
- TIME: Eine Uhrzeit im Format HH:MM:SS
- DECIMAL: Ein numerischer Wert mit einer festgelegten Anzahl von Dezimalstellen.
- FLOAT(): Ein numerischer Wert mit Gleitkommadarstellung.
- BOOLEAN

CREATE

PRIMARY → Schlüssel, wenn Pfeil draufzeigt

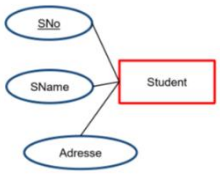
UNIQUE → Schlüssel, wenn kein Pfeil draufzeigt

- `bei m:m UNIQUE(SNo, Dno)`
- `bei 1:1 UNIQUE(SNo), UNIQUE(Dno)`

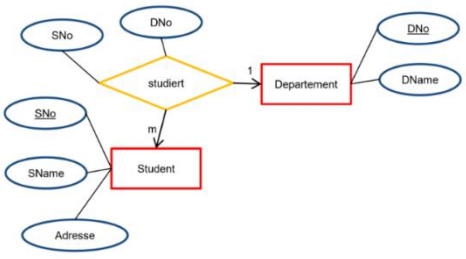
FOREIGN → stellen die Pfeile dar

- `FOREIGN KEY (DNo) REFERENCES Departement (DNo)`
  - wenn `(DNo)` nicht angegeben wird, wird automatisch PK genommen

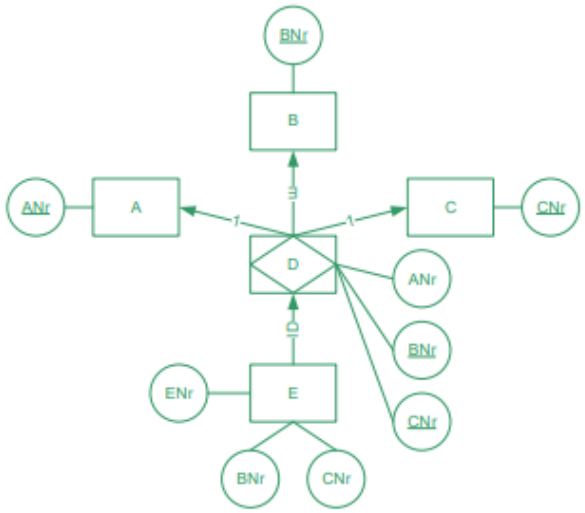
```
CREATE TABLE Student (

            SNo integer,
            SName varchar(20) NOT NULL,
            Adresse varchar(100)NOT NULL
            PRIMARY KEY (Sno);
```




```
CREATE TABLE Studiert (

            SNo char(4) NOT NULL,
            DNo  varchar(100) NOT NULL,
            UNIQUE (SNo),
            FOREIGN KEY (SNo) REFERENCES Student (SNo), (Pfeile)
            FOREIGN KEY (DNo) REFERENCES Departement (DNo)

            );
```



```
CREATE TABLE A(
  ANr integer NOT NULL,
  PRIMARY KEY
);
CREATE TABLE B(
  BNr integer NOT NULL,
  PRIMARY KEY
);
CREATE TABLE C(
  CNr integer NOT NULL,
  PRIMARY KEY
);
CREATE TABLE D(
  ANr integer NOT NULL,
  BNr integer NOT NULL,
  CNr integer NOT NULL,
  UNIQUE (ANr,BNr),
  PRIMARY KEY (BNr,CNr),
  FOREIGN KEY (ANr) REFERENCES A(ANr),
  FOREIGN KEY (BNr) REFERENCES B(BNr),
  FOREIGN KEY (CNr) REFERENCES C(CNr)
);
CREATE TABLE E(
  ENr integer NOT NULL,
  BNr integer NOT NULL,
  CNr integer NOT NULL,
  UNIQUE (ENr,BNr,CNr),
  FOREIGN KEY (BNr,CNr) REFERENCES D(BNr,CNr)
);
```



CASCADE

Verknüpfte Entitätstypen oder Beziehungen können durch CASCADE mitangepasst oder mitgelöscht werden. (ON DELECTE oder ON UPDATE

```
FOREIGN KEY (SNo) REFERENCES Student (SNo) ON DELETE CASCADE
```

CONSTRAINT

Constraints (Einschränkung) schränken die Menge der möglichen Daten, die in eine Tabelle eingegeben werden können, ein. Bsp.:

- NOT NULL
- Primärschlüssel: Schlüsselwert kann nur einmal vorkommen
- Schlüssel: können nur einmal vorkommen
- Fremdschlüssel: Werte müssen als Primärschlüsselwerte in der referenzierten Tabelle vorkommen
- CHECK-Klausel → Bedingung für Eingabe
    - `Note decimal(5,2) NOT NULL CHECK (Note >= 1.0 AND Note <= 6.0)`
- DEFAULT-Klausel → Default-Wert wenn kein Inputwert
    - `Note decimal(5,2) NULL DEFAULT 1.0`
- Constraints können benannt werden.
    - `CONSTRAINT chkNote CHECK(Note >= 1.0 AND Note <= 6.0)`

ALTER

Wird verwendet um eine bestehende Tabelle zu ändern

```
ALTER TABLE Verkauf ADD Datum date NOT NULL;
```

- ALTER: Anweisung etwas zu ändern
- ADD: Hinzufügen eines Attributs

```
ALTER TABLE Verkauf ADD CONSTRAINT FK_Kunde FOREIGN KEY (KundenNummer) REFERENCES Kunde (KNr);
```

DROP

Löschanweisung

```
DROP TABLE Name_der_Tabelle;
```

SQL - DML (DATA MANIPULATION LANGUAGE)

Der Teil von SQL, der den Inhalt, nicht die Struktur, von Datenbanken verändert. Meistens werden DML-Anweisungen von Anwendungsprogrammen generiert und nicht manuell eingegeben.

SQL-Befehle:

- INSERT
  - Fügt immer ganze Tupel in eine Tabelle ein.
  - ```
    INSERT INTO Student (SNo, SName, Adresse)
    VALUES ('81-232-23', 'Meier', 'Basel');
    ```
  - Anzahl und Datentypen im Tupel müssen übereinstimmen
  - Die Namen können weggelassen werden (bad practise)
  - Möglich sind auch komplexere Insert-Befehle:
  - ```
    INSERT INTO Student (SNo, SName, Adresse)
    VALUES ((SELECT MAX(SNo) FROM Students)+1, 'Müller', 'Basel');
    ```
    - Fügt eine neue SNo + 1 ein
- UPDATE
  - ```
    UPDATE Student SET Adresse = 'Zürich' WHERE SNo = '81-232-23';
    ```
  - Anzahl und Datentypen im Tupel müssen übereinstimmen
  - Wenn WHERE-Klausel weggelassen wird, werden alle Tupel geändert
  - Es können auch mehrere Werte geändert werden
  - ```
    UPDATE Student SET Adresse = 'Zürich', SName = 'Maier' WHERE SNo = '81-232-23';
    ```
  - Es können auch neue Attributwerte durch Berechnungen eingesetzt werden.
  - ```
    SET Note = Note + 0.5
    ```
- DELETE
  - DELETE löscht immer ganze Tupel
  - Löschen einzelner Attributwerte geht nicht (NULL eintragen)
  - Wird in der Regel mit Selektion verbunden
  - ```
    DELETE FROM Student WHERE SNo = '81-232-23';
    ```
  - Wenn die WHERE-Klausel weggelassen wird, werden alle Tupel gelöscht

REIHENFOLGE DER BEARBEITUNG VON SQL-ABFRAGEN

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

SQL – DQL (DATA QUERY LANGUAGE)

SELECT … FROM

- SELECT name AS first_name, year_hired FROM employers;

DISTINCT

- SELECT DISTINCT year_hired, dept_no FROM employees; # multiple fields

VIEW

- CREATE VIEW employer_hire_years AS SELECT id, name, hired_year FROM employers;
- Views sind virtuelle Tabellen, nur die Query wird gespeichert und eine neue Abfrage wird gemacht
    - SELECT * FROM employer_hire_years;

LIMITS

- Zeigt nur die obersten n Tupel an
    - SELECT genre FROM books LIMIT 10;

COUNT()

- Für COUNT keine Gruppierung nötig
- **Zählt die Values in einer Tabelle**
    - SELECT COUNT(birthdate), COUNT(name) FROM PEOPLE;
- **Zählt einzigartige Values**
    - SELECT COUNT(DISTINCT birthdate) FROM people;

WHERE

- SELECT title AS older_1960 FROM films WHERE release_year < 1960;
    - != → <> → Except
- Mehrfache Bedingungen
    - OR, AND, BETWEEN (inklusive)
- Mehrfache Bedingungen
    - SELECT title FROM films WHERE (release = 1994 OR release = 1996) AND …
- Bedingung aus Set von Möglichkeiten
    - WHERE country.name IN (‘Pakistan’, ’India’)

LIKE

- SELECT name FROM people WHERE name LIKE ‘Ade%’;
    - LIKE ‘Ade%’ bedeutet alles was mit Ade… beginnt
    - LIKE ‘%p%’ alles, was p enthält
- SELECT name FROM people WHERE name LIKE ‘Ev_’;
- _ bedeutet einen Char einsetzen
- NOT LIKE

WHERE (IN, ALL, ANY)

- SELECT title FROM films WHERE release IN (1999, 2001, 2020);
- SELECT title FROM films WHERE release NOT IN (SELECT release FROM movies);

- SELECT PNr FROM ltp WHERE TNr = ‘T1’ GROUP BY PNr HAVING AVG(menge) > ALL (SELECT menge FROM ltp WHERE pnr = 'P1');

IS NULL

- Null sind fehlende Werte
    - SELECT COUNT(name)
    - FROM people
    - WHERE name IS NULL;
- (IS NULL / IS NOT NULL)

AGGREGATE

- **Nur Gruppierung nötig, wenn nicht einziges Attribut in SELECT**
- AVG() (num)
- SUM() → SUM(Menge * Preis)
- MIN() / MAX()
    - `SELECT MIN (column1) FROM table WHERE some;`
- COUNT()

ORDER BY

- SELECT title, budget FROM films ORDER BY budget;
- ORDER BY budget ASC (aufsteigend <, default)
- ORDER BY budget DESC (absteigend >)
- ORDER BY budget, release_year;

GROUP BY

- Fasst Werte zu Gruppen zusammen (bsp. Anzahl Filme mit demselben Certificate):
    - SELECT certification, COUNT(*) AS count_title
    - FROM films
    - GROUP BY certification
    - ORDER BY title_count DESC;
- Wir häufig mit Aggregation eingesetzt
- Es können nur Werte in der Select-Klausel stehen, die in der group by-Klausel stehen oder als Aggregation
- … GROUP BY certification

HAVING

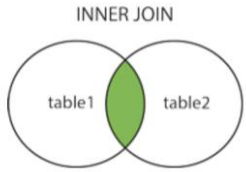
- Aggregationen können nicht mit WHERE gefiltert werden. Sie werden mit HAVING gefiltert.
    - SELECT title, COUNT(languages) AS cou_lan
    - FROM films
    - GROUP BY title
    - HAVING cou_lan > 5;

WHERE & HAVING

- SELECT x FROM y WHERE z GROUP BY x HAVING AVG(x) > 5;

INNER JOIN

Selektiert alle Tupel mit gleichen Werten auf beiden Seiten (Schnittmenge)



- SELECT president, prime_minister, p1.country, p1.continent
- FROM prime_ministers AS p1
- INNER JOIN presidents AS p2
- ON p1.country = p2.country;

USING

- Wenn zwei gleiche Zeilennamen als Schlüssel verwendet werden, kann USING verwendet werden: statt ON p1.country = p2.country
    - USING (country);

MULTIBLE JOINS

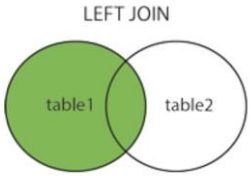
- SELECT *
- FROM left_table AS lt
- INNER JOIN right_table AS rt
- USING(id)
- INNER JOIN another_table AS at
- ON lt.id = at.aid;

Mehrere Join-Kriterien sind möglich:

- AND lt.name = at.name;

LEFT JOIN

Joint alle Elemente aus der linken Tabelle mit allen auf den Key passenden Elemente von Tabelle rechts. Leere Zellen die wegen der fehlenden rechten Seite keine Werte enthalten werden mit NULL aufgefüllt.



- SELECT region, AVG(e.gdp) AS avg_gdp
- FROM countries AS c
- LEFT JOIN economies AS e
- USING(code)
- WHERE year = 2010;
- GROUP BY region
- ORDER BY avg_gdp DESC

FULL JOIN

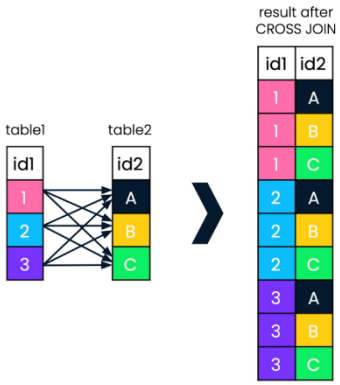
Joint alle Elemente von Tabelle links und rechts. Leere Werte werden mit Null aufgefüllt.

- SELECT region, AVG(e.gdp) AS avg_gdp
- FROM countries AS c
- FULL JOIN economies AS e
- USING(code)
- WHERE year = 2010;
- GROUP BY region
- ORDER BY avg_gdp DESC

CROSS JOIN

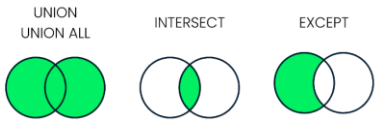
Gibt alle möglichen Kombinationen aus zwei Tabellen zurück. Cross Joins joinen jedes Element aus Tabelle A mit jedem Element der Tabelle B. Der Output besitzt die Länge (Anzahl Zeilen A * Anzahl Zeilen B)

- SELECT president, premier
- FROM presidents AS p1
- CROSS JOIN premiers AS p2
- WHERE p1.continent IN 'South America'
- AND p2.continent IN 'Asia';



SET-OPERATOREN

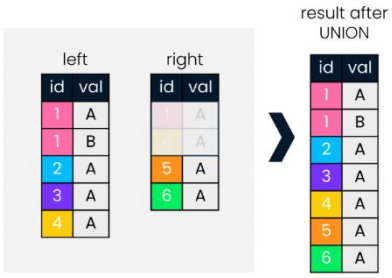
Sind um zwei nicht notwendigerweise zusammenhängende Tabellen miteinander vergleichen oder als Liste ausgeben. Dafür werden Set-Operatoren verwendet. Es müssen identische Spalten bzw. Attribute sein.



UNION

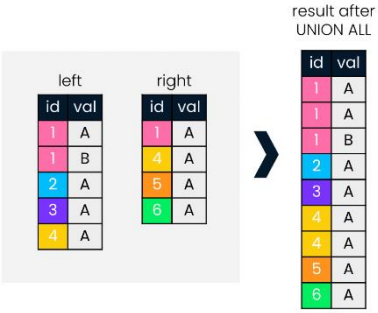
Union verbindet zwei Tabellen miteinander. Identische Elemente werden dabei nur einmal ausgegeben. (**UNION ALL** gibt auch Duplikate aus) Das Schema muss für eine Union-Operation identisch sein. Es kann auch nur Spalte x aus Tabelle 1 und Spalte x aus Tabelle zwei selektiert werden, wenn nur diese Elemente zusammenpassen. Das Resultat wird dann nur aus Spalte x bestehen.

- SELECT x, y
- FROM left_table
- UNION
- SELECT x, y
- FROM right_table;



UNION ALL

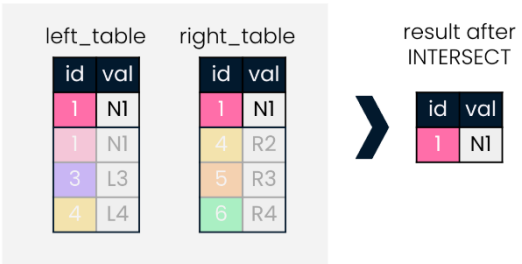
Union All verbindet zwei Tabellen miteinander. Identische UNION ALL gibt alle Elemente aus, auch Duplikate. aus. Die Das Schema muss für eine Union-Operation identisch sein. Es kann auch nur Spalte x aus Tabelle 1 und Spalte x aus Tabelle zwei selektiert werden, wenn nur diese Elemente zusammenpassen. Das Resultat wird dann nur aus Spalte x bestehen.



INTERSECT (SCHNITTMENGE)

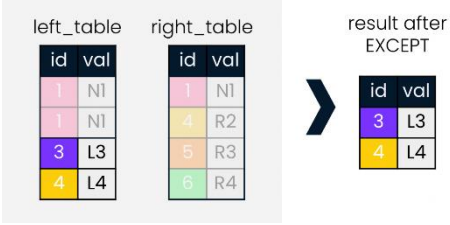
Intersect bzw. Schnittmenge gibt nur gemeinsame Elemente aus zwei Tabellen zurück. Die Anzahl Spalten sowie der enthaltene Datentyp müssen identisch sein. Das Resultat besteht dann genau aus diesen selektierten Spalten.

- SELECT id, val
- FROM left_table
- INTERSECT
- SELECT id, val
- FROM right_table;



EXCEPT

EXCEPT liefert Werte zurück, die nicht imn der linken Tabelle enthalten sind. Also wird ein Bereich ausgewählt der nicht enthalten sein soll.



- SELECT id, val
- FROM left_table
- EXCEPT
- SELECET id, val
- FROM right_table;

RELATIONALE ALGEBRA → SQL

$\sigma_{a=3}(R)$	SELECT * FROM R WHERE a=3
$\pi_{a,b}(R)$	SELECT a, b FROM R
$\pi_{a \to x, 3*b \to y}(R)$	SELECT a AS x, 3*b AS y FROM R
$\rho_{S(d,e)}(R(a,b))$	SELECT a AS d, b AS e FROM R AS S
R x S	SELECT * FROM R,S
	SELECT * FROM R CROSS JOIN S
R ⋈ S	SELECT * FROM R NATURAL JOIN S
$R \bowtie_{R.a <= S.c} S$	SELECT * FROM R, S WHERE R.a <= S.c
	SELECT * FROM R JOIN S ON R.a <= S.c
R ⟕ S	SELECT * FROM R LEFT OUTER JOIN S
R ∩ S	SELECT * FROM R INTERSECT SELECT * FROM S
R ∪ S	SELECT * FROM R UNION SELECT * FROM S
R \ S	SELECT * FROM R EXCEPT SELECT * FROM S
δ(R)	SELECT DISTINCT * FROM R

- Vereinigung ⊔ → UNION ALL

INTEGRITÄTSBEDINGUNGEN

Integritätsbedingungen sind Regeln die Daten einhalten müssen um als konsistent zu gelten. Durch die Festlegung von Integritätsbedingungen will man verhindern, dass falsche Daten in eine Datenbank gelangen.

Konsistenzregeln:

- **Bereichsintegrität** (Wert des Attributs liegt in bestimmten Wertebereich)
  - Datentyp
  - NOT NULL / NULL
- **Entitätsintegrität**
  - **Primärschlüssel** muss eindeutig sein
  - Sichergestellt durch RDBMS
- **Referentielle Integrität**
  - Der Inhalt eines Fremdschlüssels entweder NULL sein oder auf genau einen Primary Key verweisen

Konsistenz durch Constraints:

- UNIQUE-Constraints (Einzigartig)
- CHECK-Constraints → (CHECK (Note >= 1 AND Note <= 6.0)
- DEFAUL-Constraints → Defaultwerte (DEFAULT NULL)

Um komplexere Regeln in einer Datenbank einzuführen, werden mächtigere Verfahren benötigt:

- Gespeicherte Prozeduren, Funktionen & Trigger
  - Business Rules
  - Sicherheit (Schreib- und Leserechte verwalten)

DB – PROGRAMMIERUNG

SQL ist eine deskriptive Sprache. Sie wurde hauptsächlich nur zur Abfrage von Daten aus relationalen Datenbanken entwickelt. Einfache prozedurale Programme können allerdings in SQL trotzdem umgesetzt werden. Folgende Funktionalitäten können umgesetzt werden

1. Gespeicherte Prozeduren (in DB gespeichert)
   a. Skript
2. Funktionen
   a. Parameter
   b. Skalare oder tabellarische Rückgabewerte
3. Trigger (Werden durch gewisse ändernde Datenbankzustände «getriggert»)
   a. INSERT
   b. UPDATE
   c. DELETE

FUNKTIONEN BZW. GESPEICHERTE PROZEDUREN

Gespeicherte Prozeduren / Funktionen werden in der Datenbank abgelegt und können von dort aus von Benutzern oder Anwendungsprogrammen ausgeführt werden. Es handelt sich um ein einfaches prozedurales Programm.

Erstellen eines gespeicherten Prozederes

Sie werden, wie DDL-Anweisungen in einer Datenbank hinterlegt.

- CREATE {Procedure|Funktion|Trigger}
- ALTER {Procedure|Funktion|Trigger}
- DROP {Procedure|Funktion|Trigger}

GRUNDAUFBAU VON FUNCTIONS BZW. STORED PROCEDURES

```
DECLARE
  -- Deklarationsblock
  -- Der DECLARE Abschnitt ist optional
BEGIN
  -- Ausführungsteil
EXCEPTION
  -- Ausnahmeverarbeitung
  -- Der EXCEPTION Abschnitt ist optional
END;
```

KONTROLLSTRUKTUREN IN PL/PGSQL

```
- IF … [ELSE | ELSIF …]
- CASE … WHEN … ELSE … END
- LOOP … EXIT
- WHILE … LOOP … END LOOP
- FOR … IN … LOOP … END LOOP
```

BEISPIEL

1. Erzeugen der Funktion inkl. Parameter und Rückgabewert
2. BEGIN [Ausführungsteil]
3. Ende + [Definition der Sprache]

```
CREATE OR REPLACE FUNCTION IsCHPLZ(PLZ varchar(10))
    RETURNS bit
AS $$
BEGIN

  IF PLZ NOT SIMILAR TO '[0-9][0-9][0-9][0-9]' THEN
    RETURN 0;
  END IF;

  IF PLZ::integer IN (SELECT DISTINCT p.PLZ FROM PLZSchweiz p) THEN
      RETURN 1;
  END IF;

  RETURN 0;
END; $$
LANGUAGE 'plpgsql';
```

CURSOR

Ein Cursor ist eine **Variable,** die für Loops verwendet wird, die den Zugriff auf einzelne Tupel einer Anfrage ermöglicht. Der Cursor wählt jeweils einen Teil einer Tabelle aus, die durchiteriert werden soll. Er übergibt Zeile für Zeile einer Funktion.

1. Deklarieren des Cursors

```
DECLARE cursor_name CURSOR
FOR select_anweisung
[FOR UPDATE];
```

   a.
2. Öffnen des Cursors (wie öffnen einer Datei)
3. Übergeben der einzelnen Werte an Variable (FETCH)
4. Variable führt <Funktion> aus

```
CREATE OR REPLACE FUNCTION Show_AlleBesuchernamen()
RETURNS VOID AS $$
DECLARE
  rec_Besucher record;
  c_Namen CURSOR FOR SELECT Name,Vorname FROM Besucher;
BEGIN
  OPEN c_Namen;
  LOOP
    FETCH c_Namen INTO rec_Besucher;
    EXIT WHEN NOT FOUND;
    RAISE NOTICE 'Name: % Vorname: % ',rec_Besucher.Name,
                                       rec_Besucher.Vorname;
```

TRIGGER

- Wenn CONSTRAINTS nicht ausreichen
- Gut geeignet für Integritätsbedingungen (wenn Integritätsbedingungen nicht ausreichen)
- Protokollieren von Datenänderungen
- - Unübersichtlichkeit, Kompliziert zu testen

Ein Trigger ist eine Funktion oder Prozedur und wird durch sich ändernde Datenbankzustände ausgelöst bzw. getriggert.

- INSERT
- UPDATE
- DELETE

Zeitpunkt der Auslösung des Triggers

- BEFORE (Logging)
- AFTER (nachführen von Berechnungen, Historisierung)
- INSTEAD OF (gibt es nicht in pgSQL) (verhindern unerlaubter DML-Operationen)

Wenn beispielsweise in einer Datenbank mit **Kundendaten jeder Umzug dokumentiert werden** soll, kann dies mit einem Trigger gelöst werden. Bei jedem Umzug wird durch den Trigger dann also eine Funktion aufgerufen, die die neue und alte Adresse des Kunden und das Umzugsdatum speichert.

1. Erzeugen der Funktion, die durch Trigger ausgelöst werden soll
2. Funktionalität und Bedingung festlegen [BEGIN]
3. Ende setzen [END]

```
CREATE OR REPLACE FUNCTION kopiereTupel()
	RETURNS TRIGGER AS $$
		BEGIN
			INSERT INTO log VALUES (NEW.idr,NEW.r);
			RETURN NEW;
		END;
	$$ LANGUAGE plpgsql;
```

4. Trigger erzeugen und benennen
5. Event und Zeitpunkt definieren [BEFORE, AFTER, INSTEAD OF] [UPDATE, INSERT, DELETE]
6. Definition auf welche Tabelle und welche Zeilen Funktion angewendet wird
7. Funktion die Ausgelöst werden soll definieren

```
CREATE TRIGGER <name>
AFTER INSERT ON <Tabelle>
	FOR EACH ROW
	EXECUTE PROCEDURE <Funktion>();
```

Löschen: `DROP TRIGGER <nameTrigger> ON <Tabelle>;`

DATENORGANISATION & OPTIMIERUNG

- Persistenz → dauerhafte Speicherung von Daten
- Primärspeicher → RAM, Cache (nicht persistent)
- Sekundärspeicher → SSD, HDD (persistent)
- Tertiärspeicher → Tape, optische Medien (für DB ungeeignet, langsam)
- Zwischen Primär- und Sekundärspeicherung gibt es Zugriffslücken, da sich die Abfragegeschwindigkeit stark unterscheidet.

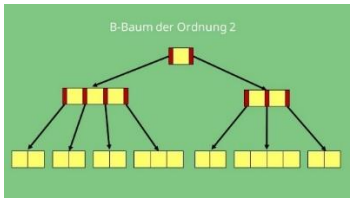
ZUGRIFF AUF DATEN IN SPEICHER

- Die Daten werden in Blöcken abgespeichert. In Postgres entspricht ein Block 8192 Bytes.
- Wichtige Anforderungen an Speichertechniken sind:
    - Effizienz bei Einzel- und Mehrfachzugriffen
    - Effizienz bei sequenziellem Durchlaufen (Sortierung, geclusterte Indexe)
    - Unterstützung für Anfragetypen

DATENORGANISATION

Je nach Zweck kann eine Datenorganisation gewählt werden

- HEAP-Datei (Haufen)
    - Datensätze unsortiert hintereinander (sequentiell) geschrieben
    - Pro: volle Speicherauslastung, schnelles Schreiben
    - Con. Langsames Abfragen
- Hash-Verfahren
    - In Bereiche gegliederte Datensätze
    - Vereinfachtes Bsp. Filelänge Modulo 3: Bereiche: Rest 0,1,2
        - Suche nach File mit Länge 6 → Suche Bereich 0
    - Pro: Abfragen, Löschen & Lesen sehr schnell
    - Con: keine sortierte Ausgabe
- ISAM-Verfahren
    - Nach Indexen organisiert und in Bereich alphabetisch sortiert
    - Anton → Bereich [A-M]
    - Erleichterte Suche durch Indexierung (lesen)
    - Durch Sortierung aufwändiges Löschen und Einfügen
- B-Baum-Verfahren (Binär-Baum)
    - Indexe des ISAM-Verfahren als Datensatz
    - Pro: keine übergrossen Indexdateien
    - Con: aufwändige Lösch- und Einfügeoperationen
    - Für überwiegend Lese-Anwendungen optimal



- Sekundärindex (zusätzlich auf Alter)
    - Zusätzlicher Index der die Suche vereinfacht
    - Pro: Erhöhte Effizienz
    - Noch kompliziertere Lösch- und Einfügeoperationen

INDEXE

Indexe in einer Datenbank werden erstellt, um einen schnelleren Zugriff auf die Datenbank zu erlauben. **Indexe werden auf gewisse Attribute gesetzt**. Wenn dann nach dem Attribut gesucht wird, kann das System über den Index nach dem Attribut suchen, ohne die gesamte Datenbank danach zu durchsuchen.

➔ Beispiel suche nach allen Studenten mit gleichem Geburtsjahr → wenn der Index auf das Geburtsjahr gesetzt ist, sind im Index für das Geburtsjahr <x> die Verweise zu den Daten über die Studenten mit gleichem Geburtsjahr verknüpft (1998 → Studenten x). So muss das System nur nach dem Index Geburtsjahr xy suchen und nicht alle Tupel der Tabelle Index durchiterieren.

Wichtig zu wissen:

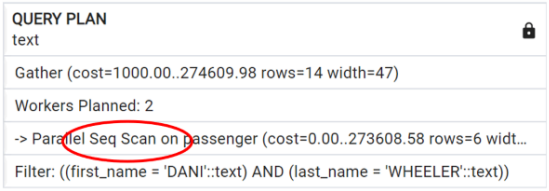
- Auf **Primärschlüsseln** ist durch das System immer ein **geclusterter Index** gesetzt
- Für sonstige Schlüssel (UNIQUE) wird ein nicht geclusterter Index erstellt
- Wenn auf indizierte Spalten zugegriffen wird, wird der Index genutzt.
- → schnellere Abfrage bzw. Suche
- → langsameres Einfügen und löschen
- Trade-Off zwischen Update/ Query
- Überindexierung kostet Rechenpower und Speicherressourcen (Abfragen können langsam werden)

INDEXARTEN

- Geclusterte Indexe → sind in einer B-Baum-Struktur nach dem Indexkriterium sortiert gespeichert
- Nicht geclusterte Indexe → Sind nicht nach dem Indexkriterium sortiert gespeichert
- Primärindex → geclusterter Index nach den Primärschlüsselattributen
- Sekundärindex → weitere (ungeclusterte) Indexe
- Dicht-/ Dünnbesetzte Indexe
    - Dicht → Eintrag für jedes Tupel vorhanden
    - Dünn → nicht für jedes Tupel Eintrag vorhanden (nur geclusterte)
    - Abdeckende Indexe (Enthält neben Suchattributwerten noch weitere Attributwerte)

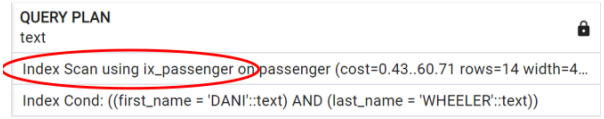
ERSTELLUNG VON INDEXEN

```
EXPLAIN SELECT * FROM passenger
WHERE first_name = 'DANI' AND last_name = 'WHEELER';
```



Ausführungsplan ohne Index

```
CREATE INDEX IX_Passenger
ON Passenger(first_name,last_name);
```



Ausführungsplan mit Index

Liste von Indizes auf Tabelle abfragen:

```
SELECT *
FROM pg_indexes
WHERE tablename = 'kurse';
```

Wann lohnt sich ein Index

1. Wenn gesuchte Tupel **weniger als ~4%** von der Gesamtanzahl Tupel ausmachen.
2. Attribute, die **oft abgefragt** werden, sollten indiziert werden
    a. Attribute die in der WHERE-Klausel in Queries vorkommen
    b. Wenn LIKE in einer Abrage vorkommt, kann der Index allerdings nicht verwendet werden. Die Suche muss trotzdem sequenziell durchgeführt werden.
3. Attribute, die **oft gejoint** werden (wenn nicht Primary Key → besitzt bereits Index)
4. Wenn **oft abgefragt** wird und **wenig eingefügt**
5. Fremdschlüssel

Wann lohnt sich ein Index nicht

1. Attribute, die eine niedrige Kardinalität aufweisen erstellt werden.
2. Attribute mit vielen Nullwerten, weil diese nicht indexiert werden können
3. Wenn LIKE in der häufigen Abfrage vorkommt, kann der Index nicht verwendet werden. Die Suche muss trotzdem sequenziell durchgeführt werden.

TRANSAKTIONEN

ACID-ANFORDERUNGEN

Atomarität (Alles oder nichts)

- zusammengehörende Lese- und Schreibzugriffe müssen als Ganzes entweder erfolgreich abgeschlossen oder rückgängig gemacht werden.

Konsistenz

- Alle Operationen hinterlassen die Datenbank in einem konsistenten Zustand, also alle Daten einen gleichen Zustand haben.

Isolation / Nebenläufigkeit

- Gleichzeitiger Zugriff mehrerer Benutzer darf zu keiner Inkonsistenz in den Daten führen. Deswegen werden alle Zugriffe hintereinander verarbeitet.

Dauerhaftigkeit

- Automatische Behandlung von Ausnahmesituationen (Fehlern) und schneller Wiederanlauf nach schwerwiegenden Fehlern. Widerherstellung verlorener Daten und Rücksetzung fehlerhafter Daten.

PROBLEME KONKURRENTER TRANSAKTIONEN

➔ Lost-Update (fast nie tolerierbar)

Wenn von zwei Benutzern direkt hintereinander ein Update an den Daten durchgeführt wird, geht das erste der beiden Updates verloren.

Lösung: **Isolation** der Transaktionen. Daten dürfen bis zur Beendigung der Transaktion nicht verändert werden.

	Transaktion Benutzer 1	Transaktion Benutzer 2
1	SELECT Wert FROM Tbl	
2		SELECT Wert FROM Tbl
3	UPDATE Tbl SET Wert = 100	
4		UPDATE Tbl SET Wert = 200
5	SELECT Wert FROM Tbl	
6		SELECT Wert FROM Tbl

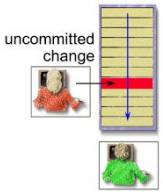
➔ Dirty-Read-Problem

Wert
100

Nutzer arbeitet mit Werten von noch nicht abgeschlossenen (committete) Transaktionen, die später vlt. sogar zurückgenommen werden. → Inkonsistenz in der DB.

	Transaktion Benutzer 1	Transaktion Benutzer 2
1	SELECT Wert FROM Tbl	
2	UPDATE Tbl SET Wert = 1000	
3		SELECT Wert FROM Tbl
4	ROLLBACK	
5		UPDATE Tbl SET Wert = Wert + 100
6		SELECT Wert FROM Tbl
7	SELECT Wert FROM Tbl	

Lösung: Isolationsebene **«READ COMMITTED»**, nur bestätigte Transaktionen lesen.



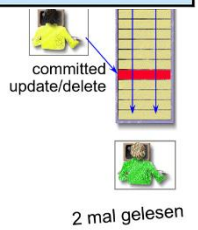
- Änderungen können erst von anderen gelesen werden, wenn die Transaktion «committed» wurde.

➔ Non-Repeatable-Read-Problem (Trans_2: UPDATE)

Wiederholte Lesevorgänge liefern unterschiedliche Ergebnisse.

	Transaktion Benutzer 1	Transaktion Benutzer 2
1	SELECT Wert FROM Tbl	
2		UPDATE Tbl SET Wert = Wert + 5
3		COMMIT
4	SELECT Wert FROM Tbl	

Lösung: Isolationsebene **«REPEATABLE READ»**. Nur Datenbankzustand sichtbar, der bei Beginn einer Transaktion vorliegt.



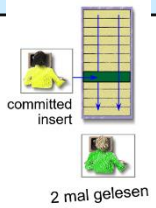
- Wurde eine Transaktion auf der Isolationsebene «REPEATABLE READ» gestartet, wird nur der Datenbankzustand gesehen, der bei Beginn der Transaktion vorlag, selbst wenn zwischenzeitlich Daten geändert und «committed» wurden.

Phantom-Read (Trans_2: INSERTS / DELETES)

Unterschiedliche Resultate von einer COUNT-Abfrage, nachdem eine andere Transaktion die Tabelle bearbeitet hat.

	Transaktion Benutzer 1	Transaktion Benutzer 2
1	SELECT count(*) INTO cnt FROM Tbl	
2	N = cnt	
3		INSERT INTO Tbl VALUES (…)
4		COMMIT
5	SELECT count(*) INTO cnt FROM Tbl	
6		

Lösung 1: **Isolationsebene «SERIALIZABLE» oder «REPEATABLE READ»** (nur in pg).



- Nur den Datenbankzustand sehen, der zu Beginn der Transaktion 1 vorlag, selbst wenn zwischenzeitlich neue Daten in die DB hinzugefügt und «committed» wurden.

ASPEKTE VON NEBENLÄUFIGKEIT

Lost-Update ist in fast keinem Transaktionssystem tolerierbar. Die anderen konkurrenten Transaktionen können in gewissem Ausmass tolerierbar sein. Es hat immer einen Einfluss auf die Performance eines Systems, je strenger die Isolationen sind, desto langsamer.

- SET TRANSACTION ISOLATION LEVEL {READ UNCOMMITTED|…};

Isolationsebene	Dirty Read	Non-Repeatable Read	Phantom Read	Lost Update
READ UNCOMMITTED	~~möglich~~ (nicht in Postgres)	möglich	möglich	~~möglich~~ (nicht in Postgres)
READ COMMITTED (Häufig in der Praxis)	verhindert	möglich	möglich	verhindert
REPEATABLE READ	verhindert	verhindert	~~möglich~~ (nicht in Postgres)	verhindert
SERIALIZABLE	verhindert	verhindert	verhindert	verhindert

→ In Postgres –> READ UNCOMMITTED = READ COMMITTED

Wichtig!!! → Je mehr Beschränkungen, desto langsamer die Anwendung

TRANSAKTIONSANWEISUNGEN IN SQL

- BEGIN TRANSACTION
  - BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
- COMMIT TRANSAKTION
- ROLLBACKTRANSAKTION (Before Image wiederherstellen)

➔ Wenn BEGIN weggelassen wird → **«Auto-Commit-Modus»**

Zustand bei einer offenen Transaktion:

- Before Image wird festgehalten, um bei einem ROLLBACK den Zustand wiederherstellen zu können
- Betroffene Datensätze sind durch Schreib-/ Lesesperren blockiert.
- Andere Transaktionen sehen geänderte Daten nicht

Nach Commit einer Transaktion:

1. Geänderte Daten sind in DB festgeschrieben
2. Alle Änderungen – alter Zustand (Before Image) und neuer Zustand (After Image) sind in Transaktionsprotokoll festgehalten.
3. Kann nicht mehr mittels Rollback zurückgesetzt werden
4. **Alle Sperren sind aufgehoben**
   a. Geänderte Daten können, wenn Isolation <Read Comitted>, wieder gelesen werden.
   b. Geänderte Daten können in anderen Transaktionen wieder geändert werden.

SCHEDULE

- Definition Schedule
  - Folge von Lese bzw. Schreibeoperationen für die parallele Ausführung einer oder mehrerer Transaktionen
- Definition Scheduler/ Transaktionsmanager
  - Managet die Ausführungsreihenfolge von Transaktionen
    - Aggressiver (liberaler) Scheduler (höheres Konfliktrisiko)
    - Konservativer Scheduler (geringes Konfliktrisiko)

Bei einem Schedule, also einem Ausführungsplan, kann es zu folgenden Konflikten kommen:

Konfliktarten durch reihenfolgeabhängige Operationen:

1. Schreib-Lese-Konflikt
2. Lese-/ Schreib-Konflikt
3. Schreib- / Schreib-Konflikt

– Lost-Update Schedule; $w_1(x)$ geht verloren:
$s_{lu} = r_1(x)r_2(x)\boldsymbol{w_1(x)}w_2(x)\ldots$

Wobei: r = read, w = write, c = commit

Diese Konflikte werden durch das BDMS möglichst vermieden. Dies kann durch einen seriellen Schedule gelöst werden.

Serieller Schedule:

- Alle Transaktionen werden nacheinander ausgeführt. Serielle Schedules werden als **sicher** und konsistenzerhaltend angesehen, haben aber eine **schlechte Performance**.

Serialisierbarer Schedule:

- Sind auch konsistenzerhalten. Parallele Ausführung. **Schnell**, aber **Konfliktanfälliger**.

KONZEPT DER SPERREN

Sperren bzw. Locks werden eingesetzt, um Zugriffskonflikte zu vermeiden. Es gibt folgende Sperroperationen:

1. Lese-Sperre
2. Schreibe-Sperre
3. Unlock

Regeln zur Sperrdisziplin:

- Schreibzugriff w(x) nur nach Setzen einer Schreibsperre wl(x) möglich.
- Lesezugriffe r(x) nur nach Setzen einer Lesesperre rl(x) oder wl(x) erlaubt.
- Eine Schreibsperre w(x) kann nur gesetzt werden, wenn auf x keine Sperren existiert.
- Eine Lesesperre r(x) kann nur gesetzt werden, wenn auf x keine Schreibsperre existiert.
- Nach u(x) darf die Transaktion kein erneutes rl(x) oder wl(x) ausführen.
- Eine Transaktion darf eine Sperre der selben Art auf demselben Objekt nicht nochmals anfordern.
- Beim Commit/Rollback müssen alle Sperren aufgehoben werden.

BLOCKING, LIVELOCK UND DEADLOCK

Sperren können zu folgenden vier Problemen führen:

1. **Blocking** → eine gesperrte Ressource zwingt andere Prozesse zu warten
   a. Lösung: kurze Transaktionen
2. **Livelock** (Verhungern) → wenn eine Transaktion warten muss und vor ihr immer wieder andere Transaktionen dran kommen.
   a. RDBMS muss «faire» Reihenfolge definieren
3. **Deadlock** (Verklemmung) → Eine Menge von Transaktionen sperren sich gegenseitig. Sie warten gegenseitig, bis eine andere aufgelöst wird.
   a. Lösung: Abfragen immer in derselben Reihenfolge starten
   b. RDBMS erkennt Deadlocks durch Zyklen-Suche und setzt die als zweite gestartete Transaktion zurück

Beispiel Deadlock:

Wenn beide Transaktionen schrittweise abwechselnd ausgeführt werden, kommt es zu einem Deadlock, welcher vom RDBMS-System erkannt wird, und die zweite Transaktion abbricht (Rollback).

t_1	t_2
$wl(x)$	
	$wl(y)$
$wl(y)$	
	$wl(x)$
Verklemmung!	

wl = Write-Lock

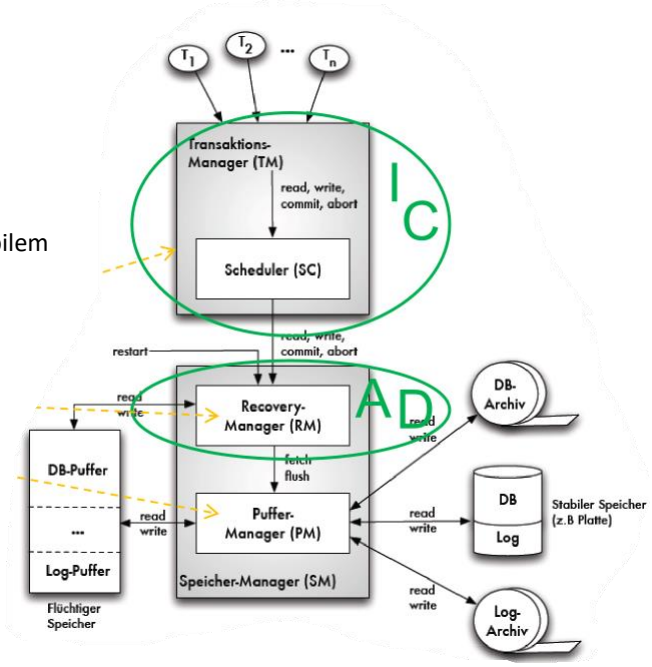
beide Transaktionen warten darauf, dass die Schreibesperren aufgelöst werden.

```
BEGIN TRANSACTION;
UPDATE airport SET airport_name = 'Malaga Airport' WHERE airport_code = 'AGP';
UPDATE passenger SET first_name = 'DANY'WHERE passenger_id = 8583106;
COMMIT;

BEGIN TRANSACTION;
UPDATE passenger SET first_name = 'DANY'WHERE passenger_id = 8583106;
UPDATE airport SET airport_name = 'Malaga Airport' WHERE airport_code = 'AGP';
COMMIT;
```

RECOVERY

Recovery umfasst alle Massnahmen zur Wiederherstellung verlorengegangener Datenbestände.

- Transaktions-Manager (TM) wahrt
  - **Isolationeigenschaften**
  - **Konsistenzeigenschaften**
- Recovery-Manager (RM) sichert
  - **Atomaritätseigenschaft**
  - **Dauerhaftigkeitseigenschaft**
  - Speicher-Manager
- Bildet Schnittstelle zwischen flüchtigem (RAM) und stabilem Speicher (SSD)



Recovery-Manager

- Sorgt dafür, dass alle comitteten Änderungen im stabilen Speicher abgelegt werden
- Keine Änderungen von aktiven oder abgebrochenen Transaktionen im stabilen Speicher verbleiben.

Puffer-Manager

- Holt Daten vom stabilen Speicher in den Puffer
- Schreibt Daten vom Puffer in den Speicher

Verschiedene Fehlerarten die zu Recovery-Massnahmen führen

1. Transaktionsfehler
   a. Eingabefehler durch Benutzer
   b. Division durch 0
   c. Lösung: Zurücksetzen der Änderungen (UNDO, rollback)
2. Systemfehler
   a. Zerstörung der Daten im Hauptspeicher
   b. Betriebssystemfehler
   c. Lösung: Wiederherstellung (REDO) oder UNDO der nicht beendeten Transaktionen
3. Medienfehler
   a. Verlust der Daten der «stabilen» Datenbank
   b. Ausfall des Speichermediums
   c. Mithilfe von Sicherungskopien und Logfiles den alten Zustand probieren herzustellen.

LOGGING

Physische Logging

Alle Transaktionen werden in einem Logfile festgehalten. Durch Sicherungspunkte werden die Logfiles in zeitliche Abstände gegliedert. Mithilfe dieser Logfiles ist es möglich Zustände wiederherzustellen.

Dafür müssen zwei Regeln eingehalten werden:

1. Vor einem Commit müssen alle zugehörigen Log-Einträge auf einen stabilen Speicher ausgelagert werden. **(REDO)**
2. Vor dem Ablegen von Daten aus dem gepufferten Speicher in den stabilen Speicher müssen alle zugehörigen Log-Einträge auf einen stabilen Speicher ausgelagert werden. (UNDO)