Relationale Algebra

Grundbegriffe

Domäne = Datentyp (Zeile). Menge von Werten desselben Typs.
Attribut = Spaltentitel. Attribut ist genau eine Domäne zugeordnet
Tupel = Reihe. Menge von Attributwerten.
Relationsschema = Kopfzeile/Spaltentitel. Menge von Attributen. $R(A_1, \dots A_n)$
Relation = Tabelle. Relationsschema + endliche Menge* von Tupeln

* Mengen sind ungeordnet und haben keine zwei gleichen Elemente.

Grundoperationen

Operationen führen zu genau einer neuen Relation (abgeschlossen)

Operand: Variablen oder Werte = Relation (Tabelle)
Operator: Symbole, die Prozeduren repräsentieren
Prädikat: Berechenbarer logischer Ausdruck

σ	Selektion	$\bowtie_P, \rtimes, \ltimes, \bowtie$	Theta-Join, Outer Joins
π	Projektion	∩	Mengendurchschnitt
ρ	Umbenennung	∪	Mengenvereinigung
x	Kreuzprodukt	\	Mengendifferenz
⋈	Natural Join (Verbund)	÷	*Division*

Selektion σ (Sigma) **= WHERE** (entfernend)

Operator — Operand

$$\sigma_{Selektionsbedingung}\ (R)$$

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

$\sigma_{\Phi 1}(\sigma_{\Phi 2}(r)) = \sigma_{\Phi 2}(\sigma_{\Phi 1}(r))$

Projektion π **= SELECT** (entfernend)
Auswahl von Attributen. Die Projektion eliminiert Duplikate.

$$\pi_{Attributliste}\ (R)$$

$\pi_{Titel,Jahr,Länge}$(Filme)

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

$\pi_{inFarbe}$(Filme)

inFarbe
True
False

$\pi_A(\pi_B(r)) \neq \pi_B(\pi_A(r))$

Erweiterte Projektion

$\pi_{3 * A \rightarrow X, B + C \rightarrow D, C}\ (R)$

R	A	B	C
	1	2	3
	4	5	6

R'	X	D	C
	3	5	3
	12	11	6

R'	X
	1
	16

$\pi_{B \rightarrow Q,C,D}$ ist equivalent zu $\pi_{B,C,D \rightarrow Q,C,D}$

Umbenennung ρ (Rho)

$$\rho_{neuerName}(R)$$

$\rho_{S(D,E)}(R(A,B))$ *Umbenennung R.A → S.D, R.B → S.E*

Joins

Cross-Join / Kreuzprodukt × (kartesisches Produkt), kombinierend
Anzahl Tuple: $R_{Tupel} \cdot S_{Tupel} = (R \times S)_{Tupel}$
Nicht kommutativ: R x S ≠ S x R

$R \times S$

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

Natürlicher Verbund / Natural-Join ⋈ (bowtie), kombinierend
Ein natural join von zwei Relationen, die **keine gemeinsamen Attribute** haben führt zu einem **Kreuzprodukt**. **Nachteil:** Vorraussetzung ist Übereinstimmung in allen Attributwerten der gemeinsamen Attribute.

$R \bowtie S$

R

A	B
1	2
3	4

S

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

R ⋈ S

A	B	C	D
1	2	5	6
3	4	7	8

R

A	B	C
1	2	3
6	7	8
9	7	8

S

B	C	D
2	5	6
2	3	5
7	8	10

R ⋈ S

A	B	C	D
1	2	3	5
6	7	8	10
9	7	8	10

Theta-Join / Equi-Join $\bowtie_P$, kombinierend
Viel flexibler als der natürliche Verbung (natural join), darum in der Praxis der Normalfall. Praxis: Zuerst filtern, dann verbinden.
Equi-Join: Verknüpfungsbedingung enthält nur einen **Gleichheitsoperator**
P = Join-Prädikat (Joinbedingung), beliebiger logischer Ausdruck

$R \bowtie_P S = \sigma_P(R \times S)$ = kartesisches Produkt von R und S mit Selektion

R

A	B	C
1	2	3
6	7	8
9	7	8

S

B	C	D
2	5	6
2	3	5
7	8	10

$R \bowtie_{A<D} S$

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

$R \bowtie_{A<D \text{ AND } R.B \neq S.B} S$

A	R.B	R.C	S.B	S.C	D
1	2	3	7	8	10

$r \bowtie_{r.B<s.C \wedge r.C=s.C \wedge r.A<s.D} s$

r

A	B	C
0	0	0
0	0	1
1	0	0
1	0	1
1	1	0

⋈

s

B	C	D
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

=

A	r.B	r.C	s.B	s.C	D
0	0	1	0	1	1
0	0	1	1	1	1

Mengenoperationen

Nur möglich wenn Attribute übereinstimmen.

Vereinigung ∪

$R \cup S$

R

Name	Adresse	Geschlecht	Geburt
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

S

Name	Adresse	Geschlecht	Geburt
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

R ∪ S

Name	Adresse	Geschlecht	Geburt
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

Durchschnitt ∩

$R \cap S$

R

Name	Adresse	Geschlecht	Geburt
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

S

Name	Adresse	Geschlecht	Geburt
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

R ∩ S

Name	Adresse	Geschlecht	Geburt
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99

Differenz \ oder –

$R \setminus S$ Achtung: $R \setminus S \neq S \setminus R$

R

Name	Adresse	Geschlecht	Geburt
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

S

Name	Adresse	Geschlecht	Geburt
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

R - S

Name	Adresse	Geschlecht	Geburt
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

Relation (= keine Duplikate)

- Relationen können keine zwei gleichen Tupel haben (da eine Relation eine Menge von Tupeln ist).
- Bei einer Relation muss jeder Attributwert genau ein Wert eines bestimmten Typs sein (atomar).

Bag-Algebra

Die relationale Algebra baut auf Mengen auf.
SQL ist aber eine «bag»-Sprache und arbeitet mit sog. Multimengen.

Multimengen = Mengen die mehrfach dasselbe Element haben können

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.
Vereinigung, ∪ «bag union»	Gleich wie relationale Algebra, also Tupel aus dem linken und Tupel aus dem rechten Operanden zusammenführen. 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 zusammengeführt. Behandlung von Duplikaten: Man zählt im linken und im rechten Operanden die Duplikate (= «Multiplizitäten») und nimmt die Summe davon.
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.
Duplikatelimination, δ	Neuer Operator. Entfernt Duplikate.

Mit dem δ**-Operator** (delta) kann man also aus
einem bag wieder eine Relation machen. = DISTINCT

Multiplizität
Die Anzahl Vorkommen (Dublikate) eines Tupels in einem bag

Outer joins ⟕, ⟖, ⟗

Fehlenden Werte durch den Platzhalter **NULL** ersetzt.

Natürlicher 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

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
a_2	b_2	c_2	NULL	NULL

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
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

Äquivalenzen der Relationenalgebra

$\sigma_\Phi(\sigma_\psi(r)) = \sigma_\psi(\sigma_\Phi(r))$ — **Kommutativität**
$\pi_A(\sigma_\Phi(r)) = \sigma_\Phi(\pi_A(r))$ falls Φ nur Attribute aus der Menge A referenziert
$r \bowtie s = s \bowtie r$ (Achtung: Relationenformat ist verschieden!)

$r \bowtie (s \bowtie t) = (r \bowtie s) \bowtie t$ — **Assoziativität**

$\pi_A(\pi_C(r)) = \pi_A(r)$ falls $A \subseteq C$
$\sigma_\Phi(\sigma_\psi(r)) = \sigma_{\Phi \wedge \psi}(r)$ — **Idempotenz**

$\pi_A(r \cup s) = \pi_A(r) \cup \pi_A(s)$
$\sigma_\Phi(r \cup s) = \sigma_\Phi(r) \cup \sigma_\Phi(s)$ — **Distributivität**

$\sigma_\Phi(r \bowtie s) = \sigma_\Phi(r) \bowtie s$ falls Φ nur Attribute von r referenziert

$\pi_{A,B}(r \bowtie s) = \pi_A(r) \bowtie \pi_B(s)$ falls für die Joinattribute J gilt: $J \subseteq A \cap B$

$r \bowtie (s \cup t) = (r \bowtie s) \cup (r \bowtie t)$

Relationale Algebra ↔ Prosa

Gast (Besucher, Restaurant)
Sortiment (Restaurant, Biersorte)
Vorzug (Besucher, Biersorte)

$(g \bowtie s) \setminus (g \bowtie s \bowtie v)$

$g \bowtie s$ = <Besucher, **Restaurant,** Biersorte>
Restaurantbesucher, die irgendein Bier trinken.

$g \bowtie s \bowtie v$ = <**Besucher,** Restaurant, **Biersorte**>
Restaurantbesucher, die ihr bevorzugtes Bier trinken.

Besucher x ist Gast in Restaurant y, dass Biersorte z im Sortiment hat, welche nicht Vorzug von Besucher x ist.

$(\pi_{Restaurant}(g) \setminus \pi_{Restaurant}(s)) \cup (\pi_{Restaurant}(s) \setminus \pi_{Restaurant}(g))$

Restaurants ohne Biere mit Gästen und Restaurants mit Biere ohne Gäste.
Bzw. Restaurants, welche keine Gäste oder keine Biere haben.

$(\pi_{Restaurant}(g) \setminus \pi_{Restaurant}(s))$ = <Restaurant>
Restaurants mit Gästen aber ohne Biere.

$(\pi_{Restaurant}(s) \setminus \pi_{Restaurant}(g))$ = <Restaurant>
Restaurants mit Biere aber ohne Gäste

$\pi_{Restaurant}(s \bowtie (\pi_{Getränk}(s) \setminus \pi_{Getränk}(v)))$

Alle Restaurants, die nicht-bevorzugte Getränke im Sortiment haben.

$\pi_{Getränk}(s) \setminus \pi_{Getränk}(v)$
Alle Getränke im Sortiment ohne Vorzug.

$s \bowtie (\pi_{Getränk}(s) \setminus \pi_{Getränk}(v))$
Sortiment mit nicht-bevorzugte Getränke.

Alle Restaurants in welchem der Besucher 'Müller' sein Vorzugsgetränk serviert bekommt.

$\pi_{Restaurant}(\sigma_{\text{Besucher='Müller'}}(v) \bowtie s)$

$\sigma_{\text{Besucher='Müller'}}(v) \bowtie s$ = <Besucher='Müller', Restaurant, Biersorte>
Alle Vorzugsbiere von Müller im Sortiment.

Alle Restaurants, die Gäste und Bier im Sortiment haben.
$\pi_{Restaurant}(g) \cap \pi_{Restaurants}(s)$ oder $\pi_{Restaurant}(g \bowtie s)$

Alle Restaurants mit Besucher, die ihr bevorzugtes Bier trinken können.
$\pi_{Restaurant}(g \bowtie v \bowtie s)$

Bestellungen, die **ALLE** (verfügbaren) Pizzas enthalten.
Bestellung ←m— enthält —m→ Pizza
$\text{Bestellung} \bowtie (\pi_{BNr}(\text{Bestellung}) \setminus (\pi_{BNr}(\pi_{BNr}(\text{Bestellung}) \times \pi_{PName}(\text{Pizza})) \setminus \pi_{BNr, PName}(\text{enthält})))$
(Nicht auf Korrektheit überprüft)

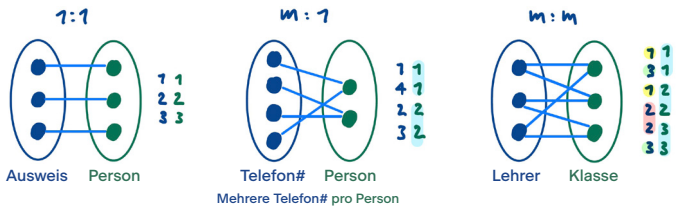
Bestellungen, die **keine** Pizza enthalten.
$\pi_{BNr}(\text{enthält}) \setminus \pi_{BNr}(\text{enthält} \bowtie \textit{Pizzen})$

Entity-Relationship (ER-Diagramm)

Diagrammsprache

Kardinalität

Wie Entitäten mit anderen Entitäten in Beziehung stehen.
Entspricht Einträge im Beziehungstyp:



Schlüssel

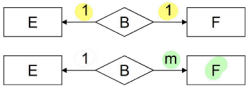
Primärschlüssel: Eindeutige Schlüssel einer Entität.
Braucht es erst **bei eingehenden Pfeilen (Referenzierung)**

Schlüssel (Primärfremdschlüssel): Referenziert ein Primärschlüssel einer anderen Entität. Ist «Primärschlüssel» der referenzierenden Entität/Beziehung. Ist auch ein Fremdschlüssel.

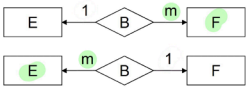
Fremdschlüssel: Referenziert ein Primärschlüssel einer anderen Entität, ist selbst aber kein Schlüssel/«Primärschlüssel».

Beziehungstyp (Verb)

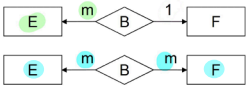
Beziehungstypen referenzieren Primärschlüssel als Fremdschlüssel:



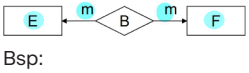
Alle PS von **E** und **F** als (Primärfremd-)Schlüssel



Alle PS von **F** als (PF-)S und alle PS von E als FS

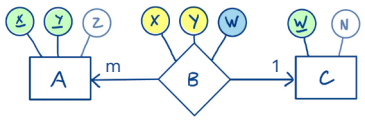


Alle PS von **E** als (PF-)S und alle PS von F als FS



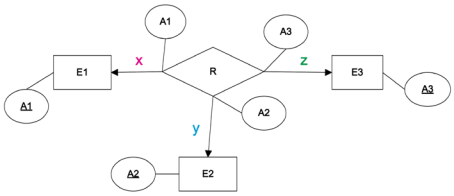
Alle PS von **E** und **F** als (PF-)Schlüssel in Kombin.

Bsp:



Primärfremdschlüssel (Ref. + Unique)
Fremdschlüssel (Reference)
Primärschlüssel (Primary Key)

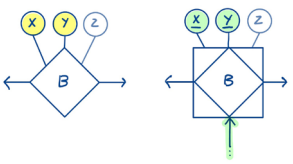
Dreiecksbeziehung



x	y	z	Primärfremdschlüssel von R
m	m	m	{A1, A2, A3}
m	m	1	{A1, A2}
m	1	m	{A1, A3}
1	m	m	{A2, A3}
m	1	1	{A1, A2} *oder* {A1, A3}
1	m	1	{A2, A3} *oder* {A2, A1}
1	1	m	{A2, A3} *oder* {A1, A3}
1	1	1	{A2, A3} *oder* {A1, A3} *oder* {A1, A2}

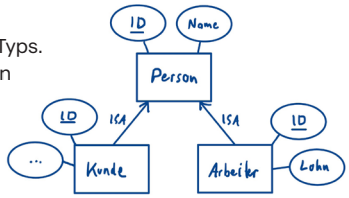
Attribute nicht in einer Menge sind «normale» Fremdschlüssel.

Soblad ein Beziehungstyp refernziert wird, werden alle Primärfremdschlüssel zu Primärschlüssel



ISA-abhängig (is a)

1 → 1 (Kardinalität)
Spezialisierung eines generalisierten Typs.
Wenn **gemeinsame Attribute** zwischen Entitäten vorhanden sind.
Existentielle Abhängigkeit.
Schlüssel von referenzierten Entität.



Abhängigkeit

Unabhängige Entitätstypen sind jene, die keine ausgehenden Pfeil haben bzw. abhängige Entitätstypen sind jene, die eingehenden Pfeile haben.

ID-abhängig

m → 1 (Kardinalität)
Wenn eine **Hierarchie** besteht.
Alle **Primärschlüssel** der oberen Entität müssen als **Schlüssel** referenziert müsssen und mind. um einen neuen Schlüssel ergänzt werden. Bsp:

Mehrere Strassen können den selben Ort referenzieren.

Telefon-Nr (m) ⟶ ID ⟶ (1) **Person**

Mehrere Telefon-Nr. (m) können einer Person (1) zugewiesen sein.
Bzw. Jede Person (1) kann mehrere Telefon-Nr. haben (m).

Regeln und Vorgehen

1. Definiere unabhängigen Entitätstypn
2. Definiere Beziehungstypn
3. Definiere Attribute und Schlüssel
4. Umwandlung Beziehungstyp in Entitätstyp (Rhombus mit Rechteck)
5. Definiere ID-abhängigen Entitätstypn
6. Definiere ISA-abhängigen Entitätstypn

Normalisierung

Wie man Tabellen zerlegen muss, damit Redundanzen und Anomalien vermieden werden.

Atomare Attributwerte

Person

Pid	Name	Telefon-Nr
12	Hans	076 123 45 67, 056 123 45 67, ... *oder* NULL

Nicht atomisch / Normalform

In einer Relation (ohne Primärschlüssel aber mit eindeutigen Einträgen):

Person

Pid	Name	Telefon-Nr
12	Hans	076 123 45 67
12	Hans	056 123 45 67
34	Peter	NULL

Nachteil:
Keine Person ohne Telefon-Nr.

Besser: **Telefon-Nr** (m) ⟶ ID ⟶ (1) **Person**

Person

Pid	Name
12	Hans

Telefon-Nr

Pid	Nr
12	076 123 45 67
12	056 123 45 67

Nachteil:
Keine Telefon-Nr. ohne Person

Am besten mit Beziehung: **Person** (1) ⟵ **Anrufbar** ⟶ (m) **Telefon-Nr**

Person

Pid	Name
12	Hans

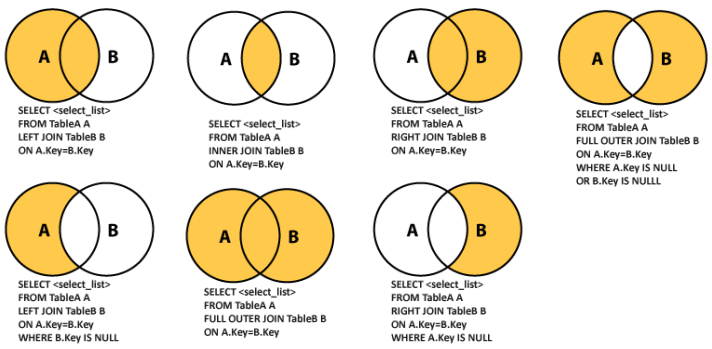
Anrufbar

Pid	Nid
12	97
12	98

Telefon-Nr

Nid	Nr
97	076 123 45 67
98	056 123 45 67

Joins



SQL

SQL 99 (SQL2)
Dokumentation: *postgresql.org/docs*

– Nicht case-sensitiv
– Nicht Turing-complete
– Tabellen (Bags) anstelle Relationen, d.h. Dublikate sind erlaubt
– Tabellen sind nicht geordnet und die Attribute-Reihenfolge is beliebig

Symbol	Bedeutung
""	Bezeichnen Symbole der Sprache, die wörtlich zu übernehmen sind
|	Alternativen werden mit einem senkrechten Strich getrennt
()	Runde Klammern dienen lediglich der Gruppierung.
[]	Eckige Klammern stehen für einen optionalen Inhalt, der Null oder einmal vorkommt.
{}	Geschweifte Klammern stehen für eine beliebige Wiederholung des Inhalts: 0-mal, 1-mal, 2-mal, ...
<>	Spitze Klammern stehen für Nichtterminale/Variablen.
::=	Definition / Produktionsregel (z.b. a ::= b)

DDL (Data Definition Language)

CREATE, ALTER, DROP
Verantwortlich für die **Struktur** einer Datenbank.
Erzeugen und Löschen von Datenbanken, Tabellen und Beziehungen.

Datentypen (Domänen):
Vorteil: Einmal zentral definiert und gepflegt. Gilt für das ganze DB-Schema.

CHAR(n)/CHARACTER(n): Zeichenkette, fixe Länge
CHAR VARYING(n)/VARCHAR(n): Zeichenkette, variable Länge
INT/INTEGER: Ganzzahl
REAL: Fliesskommazahl
NUMERIC(p,s)/DECIMAL(p,s): Festkommazahl z.B. numeric(4) → 8405

Beispiele:

```
CREATE SCHEMA IF NOT EXISTS schema_name AUTHORIZATION user_name;

DROP SCHEMA IF NOT EXISTS schema_name CASCADE;
-- CASCADE: Alles wird ohne Rückfrage gelöscht!

CREATE DOMAIN postleitzahl AS NUMERIC(4) NOT NULL;
CREATE DOMAIN default_name AS VARCHAR(20) DEFAULT('jim') NOT NULL;
CREATE DOMAIN alter INT CHECK(VALUE > 20 AND VALUE < 80) NOT NULL;

CREATE TABLE Person (
  Name VARCHAR(20) NOT NULL,
  Vorname default_name,
  Alter alter
);

ALTER TABLE Person ADD Tel VARCHAR(12) NOT NULL
-- Fügt NULL zu bestehenden Datensätze hinzu.

ALTER TABLE Person ADD Tel VARCHAR(12) NOT NULL DEFAULT 'n/a';
-- Fügt 'n/a' zu bestehenden Datensätze hinzu.
```

Constraints (DDL)

Lektion 7, S.30
Foreign-Key Triggers: Aktionen, die vom DBMS automatisch bei Dateneingabe ausgeführt werden

```
ON DELETE, ON UPDATE, NO ACTION, SET NULL, SET DEFAULT, CASCADE

CONSTRAINT fk_Attr FOREIGN KEY (Attr) REFERENCES Table(Attr)
  ON UPDATE CASCADE, ON DELETE CASCADE
```

DML (Data Manipulating Language)

INSERT, UPDATE, DELETE
Verantwortlich für den **Inhalt** einer Datenbank (pflegen von Daten).

```
INSERT INTO Firma (Name, Gruendungsjahr) VALUES ('Migros', 1925);

UPDATE Person SET PLZ = '8401', HausNr = NULL
  WHERE Name = 'Müller' AND Vorname = 'Heinz';
```

Delete löscht immer ganze Tupel.
Man kann nicht ein einzelnes Attribut löschen.

```
DELETE FROM Gast WHERE Name = 'Meier';
```

SQL-Query

SELECT FROM WHERE

$\pi_{\text{Nachnme}}(\sigma_{\text{Vorname='Müller'}}(\text{Person}))$

```
SELECT Nachnme FROM Person WHERE Vorname = 'Müller';

SELECT vorname FROM person UNION ALL SELECT vorname FROM person;
SELECT name FROM person WHERE person.vorname = 'jim';
```

Distinct

```
SELECT DISTINCT Name FROM Person;
```

DISTINCT erfordert ein internes Sortieren: **O(n log₂ n)**
Beeinträchtigt deshalb die Performance.

Unknown und NULL

Suchprädikate werden mit einer dreiwertigen Logik ausgewertet: true, false, unknown (= NULL). Unknown könnte **true oder false** sein.

¬ unknown = unknown
unknown ∧ unknown = unknown
unknown ∨ unknown = unknown

true ∧ unknown = unknown
true ∨ unknown = true
false ∧ unknown = false
false ∨ unknown = unknown

Matrix Checksumme
Vertikal wird NULL ignoriert.

SELECT SUM(A) + **SUM**(B)
SELECT SUM(A + B)

A	B
1	NULL
3	4

← 1 + NULL = NULL

↑
NULL + 4 = 4

Joins

Equi-Join: Wählt aus beiden Tabellen die Tupel aus, deren Schlüsselwerte übereinstimmen. Resultat des Equi-Joins hat mehr Spalten als Natural Join, Natural Join eliminiert jeweils eines der übereinstimmenden Attribute.

Jeder Natural Join ist ein Equi-Join, aber nicht jeder Equi-Join ist ein Natural Join.

Der Natural-Join zweier Relationen (keine Duplikate) liefert immer eine Relation.

Beim Equi-Join müssen die Schlüssel in der Bedingung eindeutig sein, sonst könnte es Duplikate haben.

```
-- Equi-Join
SELECT Name, FName FROM Mitarbeiter, Angestellt
  WHERE Mitarbeiter.name = Angestellt.mname
  AND Jahreslohn > 70000;

-- oder
SELECT Name, FName FROM Mitarbeiter
  INNER JOIN Angestellt ON Mitarbeiter.name = Angestellt.mname
  WHERE Jahreslohn > 70000;

-- Oder wenn die Spaltennamen übereinstimmen
-- (sonst macht es einen Cross-Join!)
SELECT Name, FName FROM Mitarbeiter
  NATURAL JOIN Angestellt
  WHERE Jahreslohn > 70000;
```

Folgende Queries sind äquivalent:

```
SELECT DISTINCT Name, Vorname FROM Lieblingsbier l
  INNER JOIN Sortiment s ON s.Bsorte = l.Bsorte;

SELECT DISTINCT Name, Vorname FROM Lieblingsbier
  NATURAL JOIN Sortiment;

SELECT DISTINCT Name, Vorname FROM Lieblingsbier
  INNER JOIN Sortiment USING (Bsorte);
```

Folgende Queries sind äquivalent:

```
SELECT * FROM Restaurant CROSS JOIN Besucher;
SELECT * FROM Restaurant r, Besucher b;
```

WHERE-Klausel kommt immer am Schluss aller Joins:

```
SELECT DISTINCT T.Farbe FROM L
  INNER JOIN LTP ON L.LNr = LTP.LNr
  WHERE L.LName = 'Sulzer'; -- Error!
  INNER JOIN T ON LTP.TNr = T.TNr
  WHERE L.LName = 'Sulzer'; -- Korrekt
```

Mengenoperationen

UNION, INTERSECT, EXCEPT
Mit Duplikate: ... ALL, **Ohne Duplikate:** ... DISTINCT (Default)

```
(SELECT ...) EXCEPT ALL (SELECT ...)
```

Existenzbedingung

IN, ALL, ANY (oder **SOME**)

```
SELECT * FROM A
  WHERE A.ID NOT IN (SELECT B.ID FROM B);
```

```
SELECT * FROM A
  WHERE NOT EXISTS (SELECT 1 FROM B WHERE A.ID = B.ID);
```

Nicht ganz das gleiche, Wahrheitswert bei IN könnte Unknown sein, nicht so bei EXISTS.

IN wird bevorzugt, wenn es eine kleine Liste statischer Werte gibt.
EXISTS wird bevorzugt, wenn das Vorhandensein von Werten in einer anderen Tabelle geprüft werden soll (TRUE oder FALSE).

Constraints sind **erfüllt**, wenn immer diese **TRUE** oder **UNKNOWN** ergeben.

Bei GROUP BY entsteht eine Gruppe mit allen NULL-Werten.

Distinct

Der Natural-Join zweier Relationen (keine Duplikate) liefert immer eine Relation.

Beim Equi-Join müssen die Schlüssel in der Bedingung eindeutig sein, sonst könnte es Duplikate haben.

Distinct ist **nötig** weil Attribut Sorte **kein eindeutiger Schlüssel** ist.

```
SELECT DISTINCT * FROM Lieblingsbier l
  JOIN Sortiment s ON l.Sorte = l.Sorte;
```

Distinct ist **nicht nötig** weil Attribute sind (zusammen) **eindeutige Schlüssel**.

```
SELECT * FROM Lieblingsbier l
  JOIN Sortiment s ON l.Name = s.Name AND l.Vorname = s.Vorname;
```

Case (Fallunterscheidung)

Kann als Kategorisierung für Attribut-Selektion genutzt werden.

```
SELECT Vorname, CASE
  WHEN Gehalt >= 100000 THEN 'teuer'
  WHEN Gehalt BETWEEN 50000 AND 100000 THEN 'normal'
  ELSE 'billig' END
  AS Kategorie FROM Mitarbeiter;
  -- oder
  AS Kategorie FROM (SELECT ...) AS Tabelle;
```

View (Sicht) und Common Table expressions (CTE's) = WITH

= Gespeicherte Abfrage (nicht das Resultat). Views kann man nicht updaten. Sichten können nicht sortiert werden (kein ORDER BY).

Komplexere, verschaltelte Abfragen lassen sich so besser strukturieren mit Teilabfragen.

```
CREATE OR REPLACE VIEW AverageSalary AS
SELECT ...;
SELECT * FROM AverageSalary;
```

```
-- oder mit WITH
```

```
WITH AverageSalary AS (SELECT ...)
SELECT * FROM AverageSalary;
```

Coalesce

Fallback-Wert für NULL.

```
SELECT COALESCE(Attribut, '-') FROM ...
```

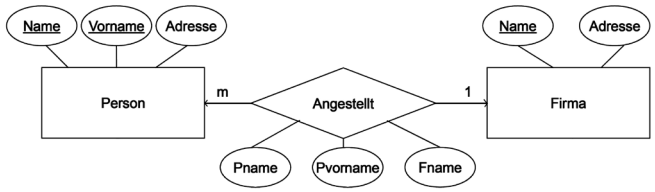
Explain

```
EXPLAIN SELECT * FROM Tabelle;
EXPLAIN (ANALYZE, FORMAT JSON)
```

Variable

```
SELECT column_name INTO variable_name FROM table_name;
```

Beispiel Beziehung




```
CREATE TABLE Mitarbeiter (
    Vorname VARCHAR(100) NOT NULL,
    Name VARCHAR(100) NOT NULL,
    CONSTRAINT p_Mitarbeiter PRIMARY KEY(Vorname, Name)
    -- Vorname together with Name must be unique
);

CREATE TABLE Firma (
    Name VARCHAR(100) NOT NULL,
    CONSTRAINT p_Firma PRIMARY KEY(Name)
    -- Name must be unique
);

CREATE TABLE Angestellt (
    MName VARCHAR(100) NOT NULL,
    MVorname VARCHAR(100) NOT NULL,
    FName VARCHAR(100) NOT NULL,
    CONSTRAINT p_Anstellung PRIMARY KEY(MName, MVorname),
    -- or: CONSTRAINT u_Anstellung UNIQUE (MName, MVorname),
    CONSTRAINT f_Mitarbeiter FOREIGN KEY (MName, MVorname)
        REFERENCES Mitarbeiter(Name, Vorname),
    CONSTRAINT f_Firma FOREIGN KEY (FName)
        REFERENCES Firma(Name)
);
```

Fremdschlüssel umbenennen

```
-- UPDATE Firma SET Name='Migrolino' WHERE Name='Migros';
-- Würde die referentielle Integrität verletzen:
-- stattdessen:
-- Fremdschlüssel-Constraint temporär entfernen
ALTER TABLE Angestellt DROP CONSTRAINT f_Firma;
-- In allen Tabellen muss Name aktualisiert werden!
UPDATE Firma SET Name = 'Migrolino' WHERE Name = 'Migros';
UPDATE Angestellt SET FName = 'Migrolino' WHERE FName = 'Migros';
-- Fremdschlüssel-Constraint wieder herstellen
ALTER TABLE Angestellt
    ADD CONSTRAINT f_Firma FOREIGN KEY (FName) REFERENCES Firma(Name);
```

Concat

```
SELECT CONCAT(Name, ' ', Stadt) AS Name FROM T
-- oder
SELECT (Name || ' ' || Stadt) AS Name FROM T
```

Integritätsbedingungen

Bereichsintegrität: Wert eines Attributes muss in einem bestimmten Wertebereich liegen (Datentyp)

Entitätsintegrität: Der Primärschlüssel einer Tabelle muss eindeutig und immer vorhanden (nicht NULL) sein

Referentielle Integrität: Der Fremdschlüssels muss entweder leer sein (NULL) oder genau einem Tupel in der referenzierten Tabelle zugewiesen sein.

PL/pgSQL

Vorteile
- Reduktion des Datenverkehrs zwischen Client und DBMS
- Realisierung sehr komplexer Abfragen möglich
- Erhöhte Sicherheit
- Parametrisierung von Abfragen

Nachteile
- Syntax und Semantik nicht standardisiert
- Verwaltungsaufwand, Unübersichtlichkeit
- Fehlerbehandliung, Debugging

Funktionen (Stored Procedures)

Aufruf ist **explizit**.
Functionen nehmen Parameter entgegen und geben Skalarwerte oder Tabellen zurück.

Triggers

Aufruf ist **implizit**.

Indexe (Indices) und Datenorganisation

Lektion 13

Heap: Datensätze werden unsortiert hintereinander (sequentiell) geschrieben

Es gibt keine für alle Zwecke optimale Datenorganisation.

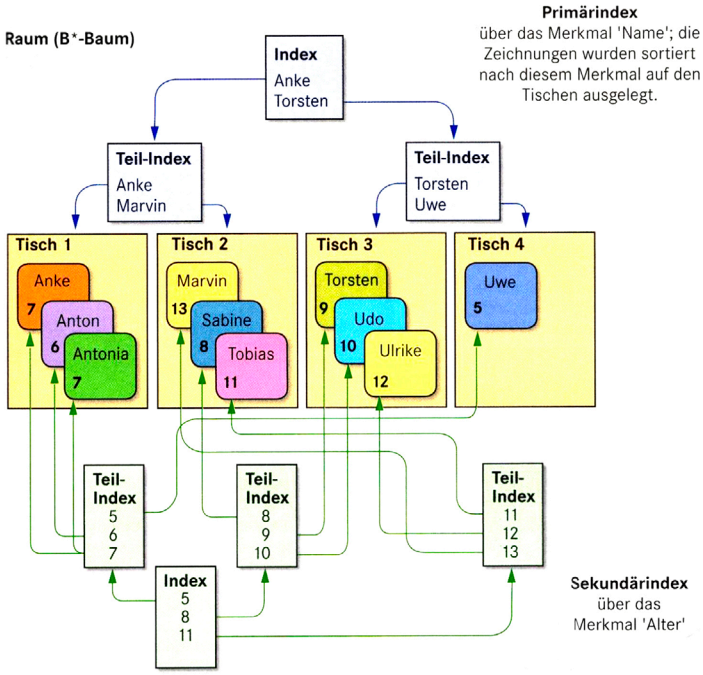
Sekundärindex

Index für ein zusätzliches Merkmal.

Vorteil: Lesezugriff kann erheblich gesteigert werden.
Nachteil: Indexe **erhöhen** die **Einfüge- und Löschzeit** signifikant.
Zudem kostet Überindexierung viel Speicherplatz.

Lohnt sich bei **vielen Abfragen** und **wenigen Einfügeoperationen.**

Attribute, die oft abgefragt werden sollen indexiert werden, insbesondere wenn Primär- mit Fremdschlüssel **gejoint** wird (was oft der Fall ist) oder in **WHERE** Klausel vorkommen.



Indexe erstellen

```
DROP INDEX IF EXISTS myindex;
CREATE INDEX myindex ON Table(first_name, last_name);
```

Transaktionen

Anforderungen an Transaktioneverarbeitung

- **Atomarität:** Zusammengehörige Folge vone Lese- und Schreibezugriffen muss als Ganzes erfolgreich abgeschlossen oder rückgemacht werden könne (Rollback).
- **Konsistenz:** Alle Operationen hinterlassen die Datenbank in einem konsistenten Zustand.
- **Isolation** (Nebenläufigkeit, Concurrency): Zugriff mehrerer Benutzer haben keinen unerwünschten Einfluss aufeinander.
- **Dauerhaftigkeit** (Recovery): Automatische Wiederherstellung (Rollforward) verlorener Daten und Rücksetzen (Rollback) fehlerhaften Daten.

Isolationsebenen

- **Dirty Read** → bei **ROLLBACK**
- **Non-Repeatable Read** → bei **UPDATE**
- **Phantom Read** → bei **INSERT**

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

Uncommitted = commited

Beispiel Nebenläufige Transaktionen (keine Sperre)

Dirty Read: Lesen von Veränderungen noch **nicht abgeschlossener Transaktionen (ROLLBACK).**

	Transaktion Benutzer 1	Transaktion Benutzer 2
1	SELECT Wert INTO W FROM Tbl	
2	UPDATE Tbl SET Wert = NeuerWert	
3		SELECT Wert INTO W FROM Tbl
4	ROLLBACK	
5		UPDATE Tbl SET Wert = W + 1
6		SELECT Wert INTO W FROM Tbl
7	SELECT Wert INTO W FROM Tbl	

Non-Repeatable Read: Lesen von Veränderungen von anderen **zwischenzeitlichen durchgeführten Transaktionen (UPDATE).**

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

Phantom Read: Lesen von Veränderungen von anderen **zwischenzeitlichen durchgeführten Transaktionen (INSERT).**

	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		

Lost-Update: **Überschreiben (UPDATE)** bereits **getätigter Updates.**

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

Beispiel

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-- oder
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED; -- etc
  SELECT ...
COMMIT TRANSACTION;
-- oder
ROLLBACK TRANSACTION;
```

Deadlock

Eine Menge von Transaktionen sperren sich gegenseitig.
Deadlock wird von PostgreSQL entdeckt. Einer der Transaktionen wird abgebrochen (mit Rollback) und der andere wird commited.

Livelock

Eine Transaktion kommt nie dran, weil immer wieder andere vorher berücksichtigt werden.

ACID

Atomarität, **K**onsistenz, **I**solation, und **D**auerhaftigkeit.