Relationale Algebra

Begriffe

Begriffe und Hierarchie:

- **Zeichen**: Grundelemente (z. B. „A“, „B“, „1“).
- **Daten**: Strukturierte Zeichen (z. B. „Hans“, „293“).
- **Information**: Daten im Kontext (z. B. „Hans hat 293 Punkte“).
- **Wissen**: Verknüpfte Information mit Anwendung.

Datenarten:

- **Strukturierte Daten**: Klare, tabellarische Struktur (z. B. Relationale Datenbanken).
- **Semi-strukturierte Daten**: Teilweise Struktur (z. B. XML, JSON).
- **Unstrukturierte Daten**: Keine festgelegte Struktur (z. B. Texte, Bilder).

Relationales Modell

Domäne: Wertebereich (z. B. Zahlen, Zeichenketten).
Attribut: Spalteneigenschaft mit Domäne (z. B. Name, Geburtsdatum).
Tupel: Zeile, Sammlung von Attributwerten.
Relation: Tabelle mit Schema und Tupeln.
Formel: $R(A_1, A_2, ..., A_n) \subseteq D_1 \times D_2 \times ... \times D_n$.
Schlüssel:

- **Primärschlüssel (PK)**: Eindeutige Identifikation (z. B. Matrikelnummer).
- **Fremdschlüssel (FK)**: Verknüpft Tabellen über PK einer anderen Relation.

Selektion

Operation: Wählt Tupel aus, die eine Bedingung erfüllen.

Schreibweise: $\sigma_{Bedingung}(R)$

Beispiel: Gegeben: Relation `Studenten(Matrikelnummer, Name, Geburtstag)`
Inhalt:

Matrikelnummer	*Name*	*Geburtstag*
1	*Meier*	19.04.1992
2	*Müller*	23.08.1998
3	*Huber*	11.09.2001

Anfrage: $\sigma_{Name='Meier'}(Studenten)$
Ergebnis:

Matrikelnummer	*Name*	*Geburtstag*
1	*Meier*	19.04.1992

Natural Join

Operation: Verbindet Tupel mit gleichen Werten in gemeinsamen Attributen.

Schreibweise: $R \bowtie S$

Beispiel:
Gegeben: `Studenten(Matrikelnummer, Name, FachID)` und `Fächer(FachID, FachName)`
Studenten:

Matrikelnummer	*Name*	*FachID*
1	*Meier*	101
2	*Müller*	102

Fächer:

FachID	*FachName*
101	*Mathe*
102	*Physik*

Anfrage: $Studenten \bowtie Fächer$
Ergebnis:

Matrikelnummer	*Name*	*FachName*
1	*Meier*	*Mathe*
2	*Müller*	*Physik*

Neue Duplikate werden entfernt, alte nicht

Projektion

Operation: Wählt bestimmte Attribute aus.

Schreibweise: $\pi_{Attribute}(R)$

Beispiel: Anfrage: $\pi_{Name,Geburtstag}(Studenten)$
Ergebnis:

Name	*Geburtstag*
Meier	19.04.1992
Müller	23.08.1998
Huber	11.09.2001

Entfernt Duplikate

Kreuzprodukt

Operation: Kombiniert alle Tupel aus zwei Relationen.

Schreibweise: $R \times S$

Beispiel:
Gegeben: `Studenten(Matrikelnummer, Name)` und `Fächer(FachID, FachName)`
Inhalte:
Studenten:

Matrikelnummer	*Name*
1	*Meier*
2	*Müller*

Fächer:

FachID	*FachName*
101	*Mathe*
102	*Physik*

Anfrage: $Studenten \times Fächer$
Ergebnis:

Matrikelnummer	*Name*	*FachID*	*FachName*
1	*Meier*	101	*Mathe*
1	*Meier*	102	*Physik*
2	*Müller*	101	*Mathe*
2	*Müller*	102	*Physik*

Joins

Theta-Join (θ)

- **Operation**: Verbindet Tupel basierend auf einer Bedingung.
- **Schreibweise**: $R\theta S$
- **Beispiel**:
  Bedingung: $R.FachID < S.FachID$.

Outer Joins

- **Left Join**: Alle Tupel aus der linken Relation; fehlende Werte rechts mit `NULL`.
- **Right Join**: Alle Tupel aus der rechten Relation; fehlende Werte links mit `NULL`
- **Full Join**: Alle Tupel aus beiden Relationen; fehlende Werte mit `NULL`.

Beispiel (Left Join):
Anfrage: Zeige alle Studenten und ihre Fächer (auch ohne Fachzuordnung).
Ergebnis:

Matrikelnummer	*Name*	*FachName*
1	*Meier*	*Mathe*
2	*Müller*	*Physik*
3	*Huber*	*NULL*

3-Ebenen-Architektur

Externe Ebene: Benutzersicht (individuelle Anwendungen).

Konzeptionelle Ebene: Gesamtsicht der Datenbank.

Interne Ebene: Speicherung und Zugriff.

Logische Datenunabhängigkeit: Änderungen in externen Sichten ohne Einfluss auf die konzeptionelle Ebene.

Physische Datenunabhängigkeit: Speicheränderungen ohne Einfluss auf konzeptionelle Sichten.

Umbenennung

Ausgangsrelation:
Studierende(MatrikelNr, Name, GebDatum)

Operation:

$\rho_{\text{Studenten(ID, Vorname, Geburtsdatum)}}(\text{Studierende})$

Ergebnis:
Studenten(ID, Vorname, Geburtsdatum) (Daten bleiben identisch).

Mengenoperationen

1. Vereinigung (∪)

- **Beschreibung**: Kombiniert Tupel aus zwei Relationen mit gleichem Schema.
- **Duplikate**: Werden **entfernt**.
- Beispiel:

  Tabelle A:

ID	$Name$
1	$Anna$
2	Ben

  Tabelle B:

ID	$Name$
2	Ben
3	$Clara$

  Ergebnis: $A \cup B$

ID	$Name$
1	$Anna$
2	Ben
3	$Clara$

2. Differenz (−)

- **Beschreibung**: Tupel aus der ersten Relation, die nicht in der zweiten vorkommen.
- **Duplikate**: Werden **entfernt**.
- **Beispiel**: $A - B$

  Ergebnis:

ID	$Name$
1	$Anna$

3. Schnitt (∩)

- **Beschreibung**: Gemeinsame Tupel beider Relationen.
- **Duplikate**: Werden **entfernt**.
- **Beispiel**: $A \cap B$

  Ergebnis:

ID	$Name$
2	Ben

Gesetze

$\sigma_\Phi(\sigma_\psi(r)) = \sigma_\psi(\sigma_\Phi(r))$ **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: $J \subseteq A \cap B$

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

Bag Algebra

1. Projektion (π)

- Entfernt keine Duplikate; Multiplizitäten bleiben erhalten.

2. Vereinigung (∪)

- Übernimmt die **höhere Multiplizität** eines Tupels aus beiden Bags.

3. Bag Concatenation (⊎)

- Multiplizitäten der Tupel werden **aufsummiert**.

4. Durchschnitt (∩)

- Multiplizität im Ergebnis entspricht der **kleineren Multiplizität** der Tupel.

5. Differenz (\)

- Subtrahiert die Multiplizitäten der Tupel; negative Werte werden auf 0 gesetzt.

6. Duplikatelimination (δ)

- Entfernt alle Duplikate und setzt Multiplizitäten auf 1.

Outer Joins

Natürlicher join:

L				R				Resultat				
A	B	C	⋈	C	D	E	=	A	B	C	D	E
a_1	b_1	c_1		c_1	d_1	e_1		a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2		c_3	d_2	e_2						

Left outer join:

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

Right outer join:

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

Full outer join:

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

Muster

1. **Alle X ohne Y:** $\pi_A(X) \setminus \pi_A(X \bowtie Y)$
2. **Alle X, die mit Y übereinstimmen:** $X \bowtie_{\text{Bedingung}} Y$
3. **Alle X, die nicht mit Y übereinstimmen:** $X \setminus (X \bowtie Y)$
4. **Alle X, die sowohl Y als auch Z erfüllen:** $(X \bowtie Y) \cap (X \bowtie Z)$
5. **Alle X, die Y haben, aber kein Z:** $(X \bowtie Y) \setminus (X \bowtie Z)$
6. **Alle X, die nur mit bestimmten Y übereinstimmen:** $X \bowtie_{\text{Bedingung}} Y$
7. **Kombination aus Selektion und Projektion:** $\pi_{\text{Attribute}}(\sigma_{\text{Bedingung}}(X))$
8. **Kombinationen aus Differenzen und Vereinigungen:** $(X \setminus Y) \cup Z$

ERD

Begriffe

Entitätstyp: Eine Klasse von Objekten, z. B. "Kunde" oder "Lieferant".
Attribut: Eine Eigenschaft eines Entitätstyps, z. B. "Name" oder "Adresse".
Beziehungstyp: Verknüpfung zwischen Entitätstypen, z. B. "Angestellt".
Primärschlüssel: Attribut oder Kombination von Attributen, die eine Entität eindeutig identifiziert.
Fremdschlüssel: Attribut in einer Tabelle, das auf den Primärschlüssel einer anderen Tabelle verweist.
Normalisierung: Prozess zur Strukturierung von Datenbanken, um Redundanzen und Anomalien zu minimieren.

ISA

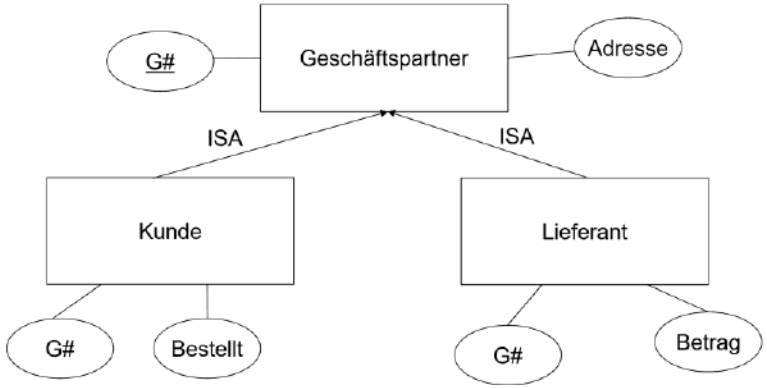
Die **IS-A Beziehung** wird für Generalisierung und Spezialisierung genutzt. Sie beschreibt, dass eine Entität eine spezifischere Version einer anderen Entität ist.

Eigenschaften der IS-A Beziehung

- **Generalisierung:** Gemeinsame Attribute verschiedener Entitäten werden in einem Supertyp zusammengefasst.
- **Spezialisierung**: Eine Entität ist eine Unterkategorie eines Supertyps mit zusätzlichen Attributen oder Beziehungen.
- **Schlüsselvererbung**: Der Primärschlüssel des Supertyps wird in die Subtypen vererbt.

Darstellung

- Ein Pfeil mit der Beschriftung "ISA" verbindet die Subtypen mit dem Supertyp.



Genauere Spezifikation (Spezialisierung)
Ist Entitätstyp F von Entitätstyp E ISA-abhängig, so gilt:
Ist M die Menge der Primärschlüsselattribute in E, so muss M in F ein Schlüssel sein.

ID

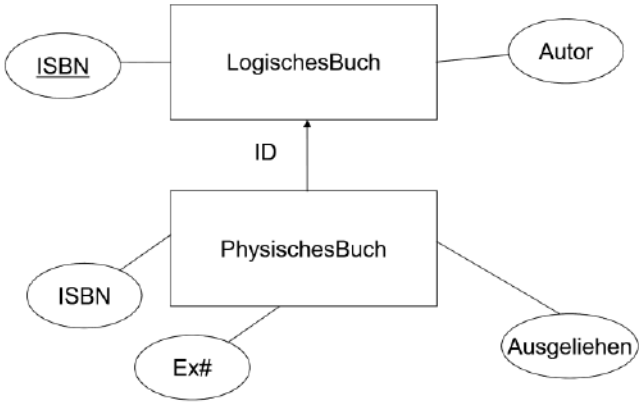
Die **ID-Abhängigkeit** beschreibt eine Hierarchie, in der eine Entität nur innerhalb eines übergeordneten Entitätstyps eindeutig identifiziert werden kann.

Eigenschaften der ID-Abhängigkeit

- Die untergeordnete Entität kann nicht ohne die übergeordnete existieren.
- Der Schlüssel des übergeordneten Entitätstyps ist Teil des Schlüssels der untergeordneten Entität.

Darstellung

- Ein ID-markierter Pfeil verbindet die abhängige Entität mit der übergeordneten Entität.



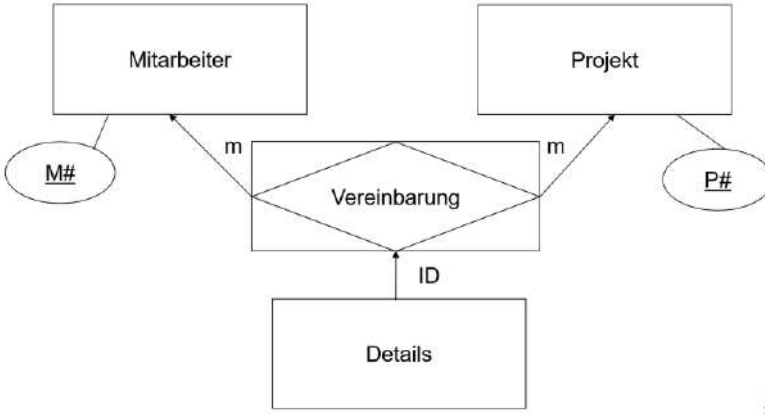
Hierarchie, hängt an "Oberklasse". Ist nur innerhalb der Hierarchie definiert
Ist Entitätstyp F von Entitätstyp E ID-abhängig, so gilt:
Ist M die Menge der Primärschlüsselattribute in E, so muss $M \cup N$ ein Schlüssel in F sein, wobei N eine zu M elementfremde Menge von Attributen von F ist (min. 1 Element)

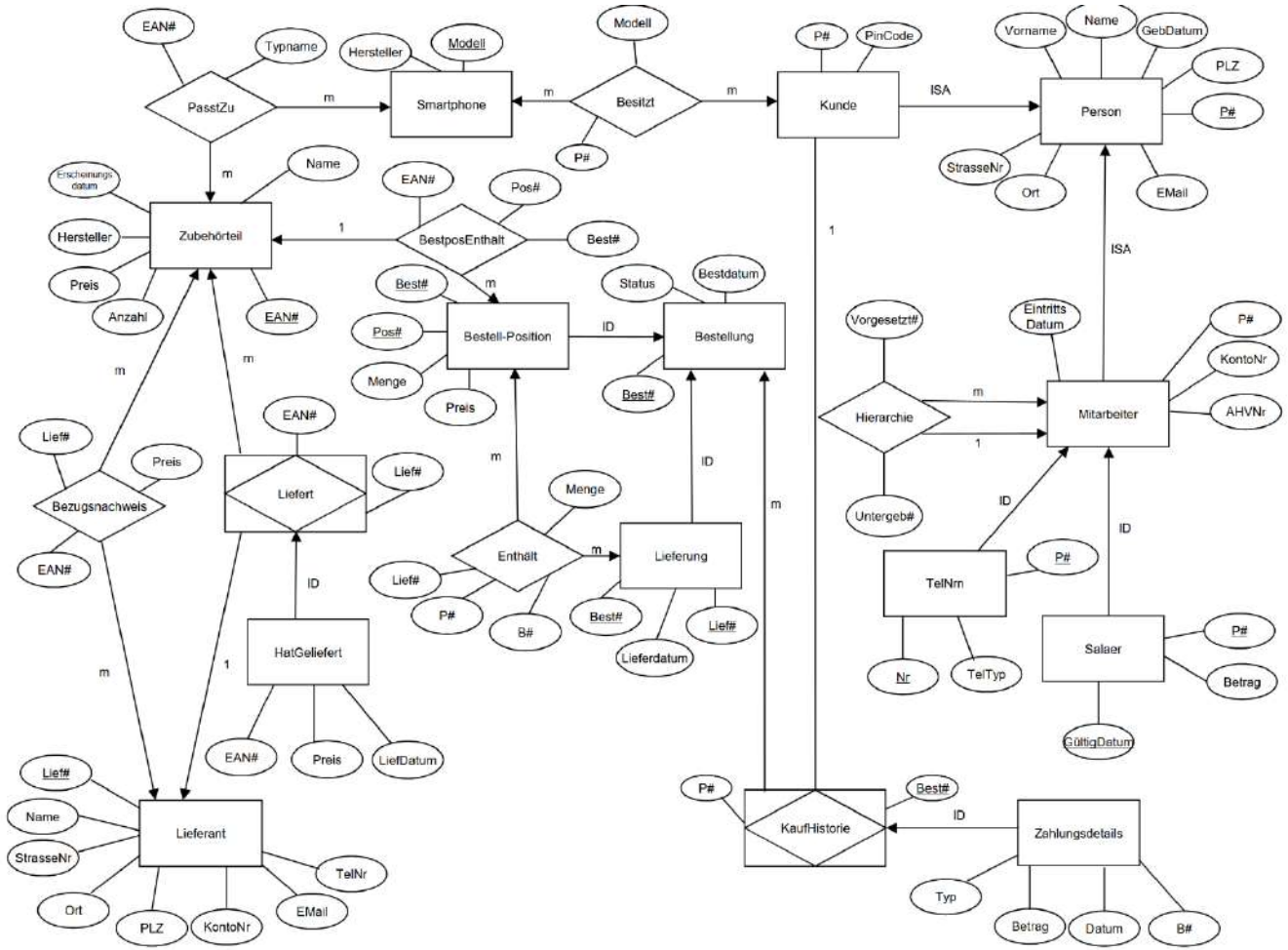
Composite Entity

Ein **Composite Entity Type** entsteht, wenn ein Beziehungstyp zusätzliche Attribute benötigt oder andere Entitätstypen anhängen soll. Dadurch wird der ursprüngliche Beziehungstyp in einen neuen Entitätstyp umgewandelt, der diese Attribute und Abhängigkeiten aufnimmt.

Eigenschaften des Composite Entity Type

1. **Transformation:**
   - Ein Beziehungstyp wird in ein Rechteck umgewandelt, das die Attribute und Schlüsselinformationen enthält.
   - Der neue Entitätstyp übernimmt die Fremdschlüssel der beteiligten Entitäten als Teil seines Primärschlüssels.
2. **Flexibilität:**
   - Ermöglicht die Speicherung zusätzlicher Informationen, die spezifisch für diese Beziehung sind.
   - Unterstützt komplexe Szenarien mit variabler Detailtiefe.
3. **ID-Abhängigkeit:**
   - Der zusammengesetzte Entitätstyp ist in der Regel ID-abhängig von den verbundenen Entitätstypen.





SQL

Projektion, Selektion und Joins

Projektion und Selektion:

- **Projektion:** Der Befehl `SELECT` wählt bestimmte Spalten aus. Beispielsweise:

```sql
SELECT Name FROM Mitarbeiter;
```

Dies entspricht in der relationalen Algebra der Projektion (π): $\pi_{\text{Name}}(\text{Mitarbeiter})$.

- **Mit DISTINCT:** `SELECT DISTINCT Name FROM Mitarbeiter;` entfernt Duplikate und entspricht $\delta(\pi_{\text{Name}}(\text{Mitarbeiter}))$.

- **Selektion:** Der Befehl `WHERE` filtert Zeilen basierend auf einer Bedingung:

```sql
SELECT * FROM Mitarbeiter WHERE Name = 'Müller';
```

Dies entspricht in der relationalen Algebra der Selektion (σ): $\sigma_{\text{Name='Müller'}}(\text{Mitarbeiter})$

JOINs:

- SQL unterstützt verschiedene Arten der Verbindung zwischen Tabellen. Zum Beispiel:

```sql
SELECT Salaer FROM Mitarbeiter m JOIN Verkauft v ON m.mnr = v.mnr;
```

Dies ist ein äquivalenter Ausdruck zu einem Equi-Join in der relationalen Algebra ($\bowtie$).

Vergleich zur relationalen Algebra:

- SQL erweitert die Konzepte der relationalen Algebra, indem es zusätzliche Operatoren (z. B. OUTER JOINS) und Flexibilität (z. B. NULL-Werte) einführt. Zudem werden bei SQL nicht immer Duplikate entfernt, was der "Bag Algebra" entspricht.

NULL-Werte und CASE

NULL-Werte:

- NULL repräsentiert unbekannte oder fehlende Werte. Besonderheiten:
  - `NULL = NULL` ist `false`.
  - Filterung: `WHERE preis IS NULL`.
- Problem: Aggregatfunktionen wie `AVG` und `COUNT` behandeln NULL unterschiedlich.

CASE:

- Ermöglicht Fallunterscheidungen:

```sql
SELECT Name,
       CASE
           WHEN Salaer > 50000 THEN 'Gut bezahlt'
           ELSE 'Normal bezahlt'
       END AS Status
FROM Mitarbeiter;
```

Aggregationen und Gruppierungen

Aggregatfunktionen:

- **COUNT:** Zählt alle Zeilen oder nur nicht-NULL-Werte eines Attributs.

```sql
SELECT COUNT(*) FROM Smartphone;
SELECT COUNT(DISTINCT Modell) FROM Smartphone;
```

- **SUM/AVG:** Berechnet Summen oder Durchschnitte. Beispiel:

```sql
SELECT AVG(Salaer) FROM Mitarbeiter;
```

  - Unterschied: `SUM` ignoriert NULL-Werte, aber `COUNT` zählt sie.

GROUP BY:

- Gruppiert Zeilen nach einem Attribut und wendet Aggregatfunktionen an:

```sql
SELECT Abteilung, COUNT(*) AS Mitarbeiteranzahl
FROM Mitarbeiter
GROUP BY Abteilung;
```

HAVING:

- Filtert Gruppen nach der Aggregation:

```sql
SELECT Abteilung, SUM(Salaer) AS Gesamtgehalt
FROM Mitarbeiter
GROUP BY Abteilung
HAVING SUM(Salaer) > 100000;
```

Vergleich zur relationalen Algebra:

- Aggregationen und Gruppierungen haben keine direkte Entsprechung in der klassischen relationalen Algebra, da diese auf Mengenoperationen basiert. SQL fügt hier erweiterte Fähigkeiten hinzu.

OUTER JOINS

LEFT/RIGHT/FULL OUTER JOIN:

- Integriert Zeilen ohne Entsprechung:

```sql
SELECT Mitarbeiter.Name, Abteilungen.Name
FROM Mitarbeiter LEFT OUTER JOIN Abteilungen
ON Mitarbeiter.Abteilung = Abteilungen.ID;
```

  - NULL-Werte werden eingefügt, wenn keine Übereinstimmung vorliegt.

Vergleich:

- Outer Joins erweitern die Möglichkeiten der relationalen Algebra durch Berücksichtigung fehlender Tupel.

Subqueries und Prädikate

EXISTS / NOT EXISTS:

- Wird verwendet, um Existenzbedingungen zu prüfen:

```sql
SELECT Name FROM Restaurant
WHERE EXISTS (
    SELECT 1 FROM Sortiment
    WHERE Sortiment.RName = Restaurant.Name AND Sortiment.Bsorte = 'Sorte1'
);
```

IN / NOT IN:

- `IN` prüft, ob ein Wert in einer Liste oder einem Subquery-Ergebnis vorkommt:

```sql
SELECT Name FROM Besucher WHERE Name IN ('Meier', 'Müller');
```

Vergleich:

- Diese Konstrukte erweitern die Algebra, indem sie komplexe Bedingungen ermöglichen, die schwer direkt in relationaler Algebra auszudrücken sind.

UNION, INTERSECT und EXCEPT

UNION:

- Kombiniert Ergebnisse von zwei Abfragen (mit oder ohne Duplikatelimination):

```sql
SELECT Name FROM Besucher1
UNION
SELECT Name FROM Besucher2;
```

INTERSECT:

- Findet die Schnittmenge:

```sql
SELECT Name FROM Besucher1
INTERSECT
SELECT Name FROM Besucher2;
```

EXCEPT:

- Subtrahiert die Ergebnisse:

```sql
SELECT Name FROM Besucher1
EXCEPT
SELECT Name FROM Besucher2;
```

Vergleich:

- Diese Operatoren entsprechen direkt der relationalen Algebra ($\cup, \cap, -$).

Views und Common Table Expressions (CTEs)

Views:

- Virtuelle Tabellen, die Abfragen abstrahieren:

```sql
CREATE VIEW AktiveMitarbeiter AS
SELECT Name, Salaer FROM Mitarbeiter WHERE Salaer > 50000;
```

  - Vorteil: Wiederverwendbarkeit und Abstraktion.
  - Nachteil: Performance, da Views oft bei jeder Nutzung berechnet werden.

CTEs:

- Temporäre Abfragen, die komplexe Abfragen vereinfachen:

```sql
WITH Durchschnitt AS (
    SELECT Abteilung, AVG(Salaer) AS Durchschnittsgehalt
    FROM Mitarbeiter
    GROUP BY Abteilung
)
SELECT * FROM Durchschnitt WHERE Durchschnittsgehalt > 60000;
```

Vergleich:

- Views und CTEs sind Erweiterungen, die Abfragen strukturieren und wiederverwendbar machen.

Prädikate

LIKE:

- Musterabgleiche mit `_` (ein Zeichen) und `%` (mehrere Zeichen).
- **Beispiel:**

```sql
SELECT Name FROM Mitarbeiter WHERE Name LIKE 'M%';
```

IS NULL / IS NOT NULL:

- Prüfen auf NULL-Werte.
- **Beispiel:**

```sql
SELECT * FROM Mitarbeiter WHERE Gehalt IS NULL;
```

BETWEEN:

- Überprüfen auf Werte in einem Bereich.
- **Beispiel:**

```sql
SELECT Name FROM Mitarbeiter WHERE Gehalt BETWEEN 40000 AND 60000;
```

IN / NOT IN:

- Prüfung, ob ein Wert in einer Liste oder Subquery enthalten ist.
- **Beispiel:**

```sql
SELECT * FROM Mitarbeiter WHERE Name IN ('Meier', 'Huber');
```

ANY/SOME und ALL Operatoren

ANY/SOME:

- Prüft, ob die Bedingung für mindestens einen Wert der Subquery zutrifft.
- **Beispiel:**

```sql
SELECT Name FROM Mitarbeiter WHERE Gehalt > ANY (
    SELECT Gehalt FROM Mitarbeiter WHERE Abteilung = 'Vertrieb'
);
```

ALL:

- Prüft, ob die Bedingung für alle Werte der Subquery zutrifft.
- **Beispiel:**

```sql
SELECT Name FROM Mitarbeiter WHERE Gehalt > ALL (
    SELECT Gehalt FROM Mitarbeiter WHERE Abteilung = 'Vertrieb'
);
```

Tabelle erstellen

Ein **Fremdschlüssel** (`FOREIGN KEY`) verweist auf die Primärschlüssel-Spalte einer anderen Tabelle.

```sql
CREATE TABLE tabelle2 (
    id SERIAL PRIMARY KEY,
    tabelle1_id INT NOT NULL,
    beschreibung TEXT,
    CONSTRAINT fk_tabelle1 FOREIGN KEY (tabelle1_id) REFERENCES tabelle1 (id)
);
```

- `tabelle1_id INT`: Spalte, die den Fremdschlüssel enthält.
- `CONSTRAINT fk_tabelle1`: Definiert den Fremdschlüssel-Constraint.
- `REFERENCES tabelle1 (id)`: Verweist auf die `id`-Spalte von `tabelle1`.

Primärschlüssel:

```sql
CREATE TABLE tabelle4 (
    spalte1 INT,
    spalte2 INT,
    daten TEXT,
    PRIMARY KEY (spalte1, spalte2)
);
```

Fremdschlüssel:

```sql
CREATE TABLE tabelle5 (
    spalte1 INT,
    spalte2 INT,
    info TEXT,
    CONSTRAINT fk_tabelle4 FOREIGN KEY (spalte1, spalte2)
        REFERENCES tabelle4 (spalte1, spalte2)
);
```

```sql
CREATE TABLE tabelle3 (
    id SERIAL PRIMARY KEY,
    tabelle1_id INT NOT NULL,
    status VARCHAR(50),
    CONSTRAINT fk_tabelle1 FOREIGN KEY (tabelle1_id)
        REFERENCES tabelle1 (id)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

- `ON DELETE CASCADE`: Löscht Zeilen in `tabelle3`, wenn der zugehörige Eintrag in `tabelle1` gelöscht wird.
- `ON UPDATE CASCADE`: Aktualisiert den Fremdschlüssel in `tabelle3`, wenn sich der Primärschlüssel in `tabelle1` ändert.

Programmierung

Views

Definition:
Eine **View** ist eine gespeicherte SQL-Abfrage, die wie eine Tabelle verwendet wird. Sie liefert immer aktuelle Daten aus den zugrunde liegenden Basistabellen und eignet sich hervorragend, um komplexe Datenabfragen zu vereinfachen.

Eigenschaften:

- **Dynamisch:** Views reflektieren immer den aktuellen Stand der Daten.
- **Materialisierte Views:** Speichern die Ergebnisse und aktualisieren diese automatisch, wenn sich die Basistabellen ändern (nicht in SQL92 standardisiert).

Operationen auf Views:

1. **Lesen (SELECT):** Views verhalten sich wie Tabellen bei Abfragen.
2. **Schreiben (UPDATE, DELETE):**
   - **UPDATE:** Nur möglich, wenn die View eindeutig auf eine Tabelle referenziert und keine Aggregatfunktionen oder Joins enthält.
   - **DELETE:** Funktioniert nur, wenn klar ist, welche Einträge in der Basistabelle betroffen sind.

Vorteile:

- **Abstraktion:** Verbirgt komplexe Abfragen.
- **Datenunabhängigkeit:** Änderungen in Tabellen erfordern keine Anpassungen in den Views.
- **Sicherheit:** Zugriffskontrolle durch eingeschränkte Rechte.
- **Wiederverwendbarkeit:** Bereitstellung häufig genutzter Abfragen.

Nachteile:

- **Performanz:** Views werden bei jeder Abfrage neu berechnet.
- **Komplexität:** Unübersichtlichkeit durch viele verschachtelte Views.

```sql
CREATE VIEW Ochsengast AS
SELECT x.Name, x.Vorname, x.Strasse, x.Gebtag, y.Frequenz
FROM Besucher x, Gast y
WHERE x.Name = y.Bname AND x.Vorname = y.Bvorname
  AND y.Rname = 'Ochsen';
```

CTEs

CTEs sind temporäre Abfragen, die als Alias für andere Abfragen in einer SQL-Statement definiert werden. Sie erleichtern das Lesen und Strukturieren von SQL-Befehlen.

Beispiel:

Berechnung des durchschnittlichen Suppenpreises pro Straße:

```sql
WITH DSPpS AS (
    SELECT Strasse, AVG(Suppenpreis) AS Durchschnittspreis
    FROM Restaurant
    GROUP BY Strasse
)
SELECT Name, Suppenpreis, Durchschnittspreis
FROM Restaurant
JOIN DSPpS ON Restaurant.Strasse = DSPpS.Strasse;
```

Integritätsbedingungen und Konsistenz

Definitionen:

- **Integrität:** Regeln, die den Zustand von Daten beschreiben, um Konsistenz sicherzustellen.
- **Konsistenz:** Übereinstimmung von Daten mit Regeln, aber nicht zwingend korrekt.

Arten:

- **Bereichsintegrität:** Werte müssen in definierten Bereichen liegen (z. B. Datentypen).
- **Entitätsintegrität:** Primärschlüssel eindeutig und nicht NULL.
- **Referentielle Integrität:** Fremdschlüssel muss auf existierende Datensätze zeigen oder NULL.

Constraints

- UNIQUE-Constraints: Nebst Primär- und Fremdschlüsseln können weitere Schlüssel definiert werden.
- CHECK-Constraints: Es können Regeln definiert werden, die Aussagen über Attribute einer Tabelle (genauer: eines Tupels) festlegen.
  Beispiel: `CONSTRAINT ck_artikel_ekpreis_vkpreis CHECK (ekpreis >= 0 AND vkpreis >= ekpreis);`
- DEFAULT-Constraints: Es können Regeln definiert werden, welche Werte als Vorgabewerte verwendet werden sollen, falls für ein Attribut kein Wert geliefert wird.
  Beispiel: `CONSTRAINT df_auftrag_datum DEFAULT SYSDATETIME();`

Komplexe Geschäftsregeln

Regeln, die Beziehungen zwischen Tabellen betreffen, z. B. Kreditvergabe nur bei keinem Überzug im letzten Jahr.
Lösung: **Gespeicherte Prozeduren, Funktionen und Trigger**.

Stored Procedures

- Beispiel: «Hallo Welt»:

```sql
CREATE OR REPLACE FUNCTION HalloWelt() RETURNS void AS
$body$
BEGIN
  RAISE NOTICE 'Hallo Welt'; -- RAISE: report messages and raise errors
END;
$body$ -- Ende des Funktionskörpers
LANGUAGE plpgsql;
```

Ausgabe erfolgt im Meldungsfenster von pgAdmin

- `SELECT HalloWelt();`
- `DROP FUNCTION HalloWelt();`

Trigger

Automatische Ausführung bei Datenänderungen (z. B. INSERT, UPDATE).

ECA-Prinzip: Event → Condition → Action.

Beispiel:

```sql
CREATE TRIGGER trigger_name
AFTER UPDATE ON table_name
FOR EACH ROW
EXECUTE PROCEDURE procedure_name();
```

Kriterium	Stored Procedures	Trigger
Aufruf	Explizit durch Benutzer/Anwendung	Implizit durch DBMS (z. B. Update)
Verwendung	Kapselung von Geschäftsregeln	Konsistenzsicherung, Logging
Transaktionskontrolle	Ja	Nein
Komplexität	Mittel	Höher, schwerer zu debuggen

Index

Definition

Einführung in Datenbankindexe

- **Motivation**: Beschleunigung von Abfragen, z. B. durch Verwendung von Indexen.
- **Problem**: Abfragen großer Tabellen können ohne Index zu langsam sein.
    - Beispiel: Abfrage von Filmen zwischen 2011 und 2020.
- **Lösung**: Erstellung eines Indexes auf Spalten, die häufig abgefragt werden.

Funktionsweise von Indexen

- **Definition**: Ein Index ist eine zusätzliche Datenstruktur, die schnellen Zugriff auf Tabellen ermöglicht.
- **Grundlagen**:
    - **Binärbaum**: Effiziente Suche mit O(log n).
    - **B-Bäume**: Dynamische Anpassung an Datenmengen.
- **Formen von Indexen**:
    - **Primärindex**: Basierend auf dem Primärschlüssel, eindeutig.
    - **Sekundärindex**: Unterstützt nicht-schlüssige Abfragen, kann mehrere Einträge umfassen.

Schlüssel & Index

Index vs. Schlüssel:
- Index: physische Datenstruktur, um Datensatz schnell zu finden
- Schlüssel: eindeutige Identifizierung eines Datensatzes (relational Modellierung)

Primärindex:
- Zugriffspfad, der die Dateiorganisationsform nutzen kann
- über Primärschlüssel oder eventuell Schlüsselkandidat (duplikatenfrei)
- Maximal einer pro Tabelle

Sekundärindex
- Jeder Zugriffspfad ohne Nutzung der Dateiorganisationsform
- Mehrere pro Tabelle

Primärschlüssel (wichtiger Kandidat für Primär-/Sekundärindex):
- Echter Schlüssel (d.h. identifizierend, ohne Duplikate)
- Maximal einer pro Tabelle

Sekundärschlüssel / Suchschlüssel (Kandidat für Sekundärindex):
- Nicht zwingend Schlüssel
- Mehrere pro Tabelle

Wann Index

1. Attribute, die oft abgefragt werden, sollten indiziert werden.
2. Fremdschlüssel sollten indexiert werden, insbesondere dann, wenn über «Primär-Fremdschlüssel» gejoint wird (was häufig der Fall ist).
3. Attribute über die oft gejoint wird; wenn über mehrere Attribute gejoint wird dann muss ein zusammengesetzter Index verwendet werden.
4. Attribute mit niedriger Kardinalität (Extrembeispiele: Geschlecht, Ja/Nein-Flags u.ä.) sollten nicht indexiert werden (es gibt dafür spezielle Indexstrukturen, hier aber nicht behandelt).

Achtung:

Überindexierung kostet Ausführungszeit und Speicherplatz und kann den Optimizer in die Irre führen.

Transaktion

ACID

Atomicity (Atomarität): Einheitliche Transaktion, die entweder vollständig ausgeführt (Commit) oder rückgängig gemacht (Rollback) wird.

Consistency (Konsistenz): Datenbank bleibt nach einer Transaktion in einem gültigen Zustand.

Isolation (Isolation): Transaktionen beeinflussen sich gegenseitig nicht.

Durability (Dauerhaftigkeit): Nach Commit bleiben Daten auch bei Fehlern erhalten.

Nebenläufigkeitsprobleme

1. **Lost Update:** Änderungen einer Transaktion werden überschrieben.
   - **Lösung:** Sperren oder isolierte Datenzugriffe.
2. **Dirty Read:** Transaktion liest unbestätigte Änderungen.
   - **Lösung:** Nur bestätigte Änderungen lesen.
3. **Non-Repeatable Read:** Unterschiedliche Ergebnisse bei wiederholtem Lesen.
   - **Lösung:** Datenbankzustand zu Transaktionsbeginn fixieren.
4. **Phantom Read:** Neue Daten beeinflussen Abfragen.
   - **Lösung:** Isolation oder Sperren der Zugriffswege.

Isolationsebenen

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

Schedules

Schedule (Ablaufplan):

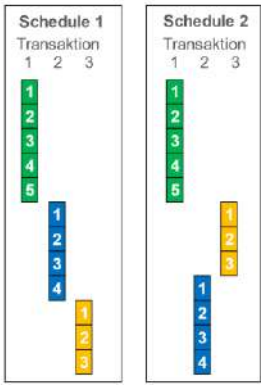
- Folge von Lese- bzw. Schreiboperationen für die (parallele) Ausführung einer oder mehrerer Transaktionen.
- Schedules können ACID-Eigenschaften verletzen.

Vollständiger Schedule = History

- Sämtliche Schritte aller anstehender Transaktionen inklusive Terminierung (COMMIT, ABORT)
- Für jede Transaktion ist festgehalten, ob sie erfolgreich endet oder abbricht (dies kann in einem Schedule noch offen sein).

Serieller Schedule:

- Alle Transaktionen nacheinander.
- Für n Transaktionen existieren n! verschiedene serielle Schedules.
- Serielle Schedules werden als konsistenzerhaltend betrachtet.
- Sicher, aber schlechte Performance → verschränkte Ausführung ist erwünscht.
- Jeder Schedule kann zu anderem Resultat führen.



Strategien:

1. **Aggressiv (liberal):**
   - Maximale Parallelität, erlaubt Konflikte.
   - Erkennung und Behebung von Konflikten nachträglich.
   - Beispiel: Multiversion Concurrency Control (MVCC) in PostgreSQL, Oracle.
2. **Konservativ:**
   - Vermeidet Konflikte durch vorsichtiges Scheduling.
   - Nachteil: Reduzierte Parallelität.

Serialisierbarkeit

Ein nicht-serialer Schedule ist serialisierbar, wenn er das gleiche Ergebnis liefert wie ein entsprechender serialer Schedule. Dies stellt sicher, dass parallele Ausführungen konsistent bleiben.

Konfliktserialisierbarkeit:

- Ein Schedule ist konfliktserialisierbar, wenn:

Die Reihenfolge konfliktärer Operationen (z. B. w1(x), r2(x)) mit der eines seriellen Schedules übereinstimmt.

- Nachweis erfolgt über einen Abhängigkeitsgraphen:
  - Knoten: Transaktionen
  - Kanten: Konflikte zwischen Transaktionen
  - **Zyklusfrei:** Der Schedule ist serialisierbar.

Konflikte in Schedules

1. **Schreib-Lese-Konflikt:** w1(x)w1(x)w1(x), r2(x)r2(x)r2(x)
2. **Lese-Schreib-Konflikt:** r1(x)r1(x)r1(x), w2(x)w2(x)w2(x)
3. **Schreib-Schreib-Konflikt:** w1(x)w1(x)w1(x), w2(x)w2(x)w2(x)

Keine Konflikte: Nur Lesezugriffe auf dieselben Daten.

Transaktionsmanagement

Sperren (Locks):

- **Lese-Sperre:** Mehrere Transaktionen können lesend zugreifen.
- **Schreib-Sperre:** Exklusiver Zugriff für eine Transaktion.

Probleme:

- **Blocking:** Verzögerung durch gesperrte Ressourcen.
- **Livelock:** Transaktion wird ständig übergangen.
- **Deadlock:** Gegenseitige Sperrung mehrerer Transaktionen.
- **Lösung:** Fairer Scheduler, Deadlock-Erkennung und Rücksetzen von Transaktionen.

Recovery (Wiederherstellung)

- **Fehlerarten:**
  1. **Transaktionsfehler:** Lokale Fehler; Änderungen werden durch Rollback rückgängig gemacht.
  2. **Systemfehler:** Flüchtige Daten verloren; REDO und UNDO erforderlich.
  3. **Medienfehler:** Permanente Datenverluste; Wiederherstellung durch Backups und Logs.
- **Logging:** Vorherige und neue Datenwerte werden in Transaktionslogs gespeichert.