POSTGRESQL - DATENBANKENTWICKLUNG AUFBAUKURS

ÜBER MICH

- **Manuel Kübler, B.Sc.**
- Full-Stack-Entwickler
- Berater & Dozent
- Technologien:
  - Frontend: Vue.js (Nuxt.js) & React (Next.js) mit TypeScript
  - Backend: ASP.NET Core, Laravel, Nest.js, Spring Boot
  - Database: SQL, MySQL, PostgreSQL
  - Testing: NUnit, PHPUnit, JUnit, Jest, Vitest, Cypress, Storybook
  - Tooling: Nx (Monorepo), CI/CD mit GitHub Actions
  - Presentation: reveal.js
- Kontakt:
  - mail@softwaredesign-solution.de
  - github.com/softwaredesign-solution

ZEITPLAN

- 9 - 16 Uhr
- 1 Stunde Mittagspause

09.02.2026

- Einstieg & Wiederholung
- Erweiterte Tabellenverknüpfungen
- Komplexe SQL-Abfragen
- Erweiterte Aggregation & Analyse
- Fortgeschrittene Constraints
- Sequences & Identity Columns
- Datenaktualisierung (INSERT / UPDATE / DELETE)
- PostgreSQL-spezifische Datentypen
- JSON & JSONB Deep Dive

10.02.2026

- Funktionen & Prozeduren
- Kontrollstrukturen
- Set-basierte Verarbeitung in Funktionen
- Dynamisches SQL
- Arbeiten mit Cursorn
- Trigger & Triggerfunktionen (DML)
- Event Trigger (DDL)
- Fehler- & Ausnahmebehandlung

11.02.2026

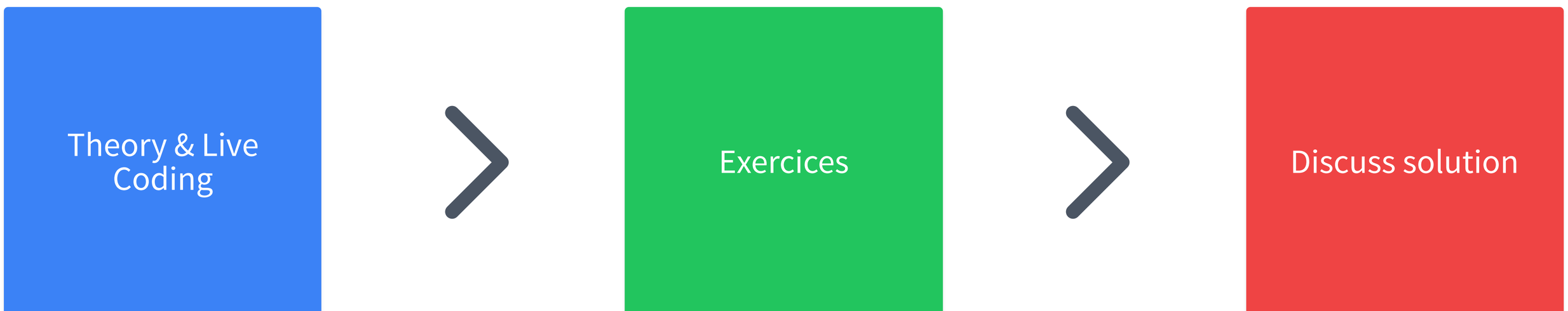
- Transaktionen & Nebenläufigkeit
- Performance-Grundlagen
- Indexing Deep Dive
- EXPLAIN & EXPLAIN ANALYZE

THEMEN

Tag 2
- (x) Funktionen & Prozeduren
- (x) Kontrollstrukturen
- Set-basierte Verarbeitung in Funktionen
- (x) Dynamisches SQL
- (x) Arbeiten mit Cursorn
- (x) Trigger & Triggerfunktionen (DML)
- Event Trigger (DDL)
- Fehler- & Ausnahmebehandlung

Tag 3
- Transaktionen & Nebenläufigkeit
- Performance-Grundlagen
- Indexing Deep Dive
- EXPLAIN & EXPLAIN ANALYZE
- Partitionierung
- Architektur & Strukturierung
- Monitoring, Logging & Tools

ABLAUF DES KURSES



VORAUSSETZUNGEN

- Grundlegende Kenntnisse im Bereich Datenbank
- Eigener Laptop
- Empfohlen: Installation von Docker

AUFGABENBLÄTTER & MATERIALIEN ZUR SCHULUNG

- Zu jedem Thema der Schulung steht ein eigenes Aufgabenblatt bereit
  - inklusive Schritt-für-Schritt-Übungen
  - und Show-Solution-Bereich mit Musterlösungen
- Alle Aufgabenblätter und Materialien können heruntergeladen werden unter: https://github.com/SoftwareDesign-Solution/2026-02-09-postgresql-aufbaukurs-workshop
- Im Repository enthalten:
  - database/ – Datenbank-Schema und Beispiel-Daten
  - exercises/ – alle Aufgabenblätter (Markdown)
  - slides/ – Präsentationsfolien
  - psqltraining-docker.yml – Docker Compose zum Starten eines PostgreSQL- und pgAdmin-Containers
- Mit der Docker-Datei kannst du:
  - eine vollständige PostgreSQL-Umgebung starten
  - pgAdmin direkt zur Visualisierung und Abfrage verwenden
  - ohne lokale Installation arbeiten
- Hinweis:
  - Beim Verbinden in pgAdmin muss ggf. die IP-Adresse des PostgreSQL-Containers ermittelt werden.
  - Befehl (PowerShell):

```
docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' psqltraining
```

EINSTIEG & WIEDERHOLUNG

- JOIN-Varianten (Kurzüberblick)
- Constraints – Überblick
- Aggregationen & GROUP BY

ÜBERBLICK: JOIN-VARIANTEN

INNER JOIN

- Liefert nur Datensätze mit Übereinstimmung in beiden Tabellen
- häufigste JOIN-Variante

```sql
SELECT *
FROM orders o
INNER JOIN customers c ON c.id = o.customer_id;
```

LEFT / RIGHT JOIN

- LEFT JOIN: alle Zeilen der linken Tabelle + passende der rechten
- RIGHT JOIN: alle Zeilen der rechten Tabelle + passende der linken
- Nicht passende Werte → NULL

```sql
SELECT *
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.id;
```

FULL OUTER JOIN

- Kombination aus LEFT + RIGHT JOIN
- Liefert alle Datensätze beider Tabellen

```sql
SELECT *
FROM table_a a
FULL OUTER JOIN table_b b ON a.id = b.a_id;
```

CROSS JOIN

- Kartesisches Produkt
- Jede Zeile mit jeder Zeile kombiniert
- Vorsicht bei großen Tabellen ⚠️

```sql
SELECT *
FROM products
CROSS JOIN categories;
```

ÜBERBLICK: CONSTRAINTS

Was sind Constraints?

- Regeln zur Datenintegrität
- Werden direkt auf Tabellenebene definiert
- Erzwingen korrekte und konsistente Daten

WICHTIGE CONSTRAINT-TYPEN

- PRIMARY KEY
  - Eindeutige Identifikation
  - Implizit NOT NULL + UNIQUE
- FOREIGN KEY
  - Verknüpfung zwischen Tabellen
  - Erzwingt referenzielle Integrität

```
customer_id INT REFERENCES customers(id);
```

WEITERE CONSTRAINTS

- NOT NULL
  - Wert darf nicht leer sein
- UNIQUE
  - Wert muss eindeutig sein
- CHECK
  - Definieren benutzerdefinierte Bedingungen
  - z.B. CHECK (age >= 18)

WARUM CONSTRAINTS WICHTIG SIND

- Schutz vor fehlerhaften Daten
- Vereinfachen Logik in Anwendungen
- Grundlage für:
  - saubere Joins
  - korrekte Aggregationen
  - verlässliche Trigger & Funktionen

AGGREGATIONEN & GROUP BY

AGGREGATFUNKTIONEN

- COUNT()
- SUM()
- AVG()
- MIN(), MAX()

```sql
SELECT COUNT(*) FROM orders;
```

GROUP BY – GRUNDPRINZIP

- Gruppiert Datensätze nach Spalten
- Aggregatfunktionen arbeiten pro Gruppe

```sql
SELECT customer_id, COUNT(*)
FROM orders
GROUP BY customer_id;
```

GROUP BY – WICHTIGE REGELN

- Jede Spalte im SELECT
  - muss entweder:
    - aggregiert sein oder
    - im GROUP BY stehen
- Reihenfolge:
  1. FROM
  2. WHERE
  3. GROUP BY
  4. HAVING
  5. SELECT
  6. ORDER BY

HAVING VS. WHERE

- WHERE
  - Filtert Zeilen vor der Gruppierung
- HAVING
  - Filtert Gruppen nach Aggregation

```sql
SELECT customer_id, COUNT(*)
FROM orders
GROUP BY customer_id
HAVING COUNT(*) > 5;
```

BRÜCKE ZUM AUFBAUKURS

Diese Grundlagen sind Voraussetzung für:

- Window Functions (OVER, PARTITION BY)
- Cursor & Schleifen
- Performance-Optimierung
- Trigger & Aggregationen
- Komplexe Reports & Analysen

ERWEITERTE TABELLENVERKNÜPFUNGEN

MEHRFACH-JOINS

Was sind Mehrfach-Joins?

- Verknüpfung von mehr als zwei Tabellen
- Sehr häufig in realen Datenmodellen
- Grundlage für Reports, Auswertungen und APIs

Typische Einsatzszenarien

- Kunde → Bestellung → Bestellposition → Produkt
- Benutzer → Rolle → Berechtigung
- Logische Entitäten mit mehreren Beziehungen

Beispiel: Mehrfach-Join

```sql
SELECT
  c.company_name,
  o.order_id,
  p.product_name,
  od.quantity
FROM customers c
JOIN orders o
  ON o.customer_id = c.customer_id
JOIN order_details od
  ON od.order_id = o.order_id
JOIN products p
  ON p.product_id = od.product_id;
```

SELF-JOINS

Was sind Self-Joins?

- Eine Tabelle wird mit sich selbst verknüpft
- Meist bei hierarchischen oder referenziellen Strukturen

Typische Anwendungsfälle

- Mitarbeiter ↔ Vorgesetzter
- Kategorien mit Parent-Kategorie
- Vergleich von Datensätzen innerhalb einer Tabelle

Beispiel: Self-Join

```sql
SELECT
  e.employee_name AS employee,
  m.employee_name AS manager
FROM employees e
LEFT JOIN employees m
  ON e.manager_id = m.employee_id;
```

CROSS JOIN

Was sind CROSS JOINs?

- Kartesisches Produkt zweier Tabellen
- Jede Zeile der linken Tabelle wird mit jeder Zeile der rechten Tabelle kombiniert

Eigenschaften

- Keine JOIN-Bedingung
- Kann sehr schnell sehr viele Zeilen erzeugen
- Vorsicht bei großen Tabellen ⚠️

Beispiel: CROSS JOIN

```sql
SELECT
  p.product_name,
  s.shipper_name
FROM products p
CROSS JOIN shippers s;
```

LATERAL JOIN

Was ist ein LATERAL JOIN?

- Erlaubt Subqueries, die auf Spalten der linken Tabelle zugreifen
- Subquery wird pro Zeile ausgeführt

Wann LATERAL nutzen?

- „Top-N pro Gruppe“
- Abhängige Subqueries
- Wenn klassische JOINs oder GROUP BY unübersichtlich werden 👉

PostgreSQL-Spezialität mit hoher Praxisrelevanz

Beispiel: LATERAL JOIN

```sql
SELECT
  c.company_name,
  o.order_id,
  o.order_date
FROM customers c
JOIN LATERAL (
  SELECT order_id, order_date
  FROM orders o
  WHERE o.customer_id = c.customer_id
  ORDER BY order_date DESC
  LIMIT 1
) o ON true;
```

SEMIJOINS (EXISTS)

Was ist ein Semijoin?

- Prüft nur das Vorhandensein von Datensätzen
- Liefert keine Spalten aus der Unterabfrage zurück
- Sehr performant

Beispiel: EXISTS

```sql
SELECT
  c.company_name
FROM customers c
WHERE EXISTS (
  SELECT 1
  FROM orders o
  WHERE o.customer_id = c.customer_id
);
```

ANTIJOINS (NOT EXISTS)

Was ist ein Antijoin?

- Gegenspieler zum Semijoin
- Liefert Datensätze, für die keine passende Zeile existiert

Beispiel: NOT EXISTS

```
SELECT
  c.company_name
FROM customers c
WHERE NOT EXISTS (
  SELECT 1
  FROM orders o
  WHERE o.customer_id = c.customer_id
);
```

EXISTS VS. JOIN

Variante	Vorteil
EXISTS	Sehr gute Performance
JOIN	Gefahr von Duplikaten
NOT EXISTS	Sicherer als NOT IN

KOMPLEXE SQL-ABFRAGEN

SUBQUERIES – GRUNDLAGEN

Was ist eine Subquery?

- Eine SELECT-Abfrage innerhalb einer anderen SQL-Abfrage
- Kann in verschiedenen SQL-Klauseln vorkommen:
  - SELECT
  - FROM
  - WHERE
  - HAVING

Typische Einsatzfälle

- Filter auf aggregierte Werte
- Abhängige Berechnungen
- Verschachtelte Logik ohne Joins

UNKORRELIERTE SUBQUERIES

Eigenschaften

- Innere Abfrage wird einmal ausgeführt
- Keine Abhängigkeit von der äußeren Abfrage
- Liefert einen festen Wert oder eine feste Ergebnismenge

Beispiel

```sql
SELECT product_name, price
FROM products
WHERE price > (
    SELECT AVG(price)
    FROM products
);
```

KORRELIERTE SUBQUERIES

Eigenschaften

- Subquery bezieht sich auf Spalten der äußeren Abfrage
- Wird für jede Zeile der äußeren Abfrage ausgeführt
- Oft weniger performant als Joins oder CTEs

Beispiel

```sql
SELECT c.customer_id, c.company_name
FROM customers c
WHERE EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.customer_id = c.customer_id
);
```

SUBQUERIES – VORTEILE & NACHTEILE

Vorteile

- Sehr kompakte Syntax
- Intuitiv für einfache Logik
- Gut geeignet für einmalige Berechnungen

Nachteile

- Korrelierte Subqueries können langsam sein
- Schlechte Lesbarkeit bei Verschachtelung
- Schwer zu debuggen

COMMON TABLE EXPRESSIONS (CTE)

Was ist eine CTE?

- Temporäre Ergebnismenge, die innerhalb einer Abfrage definiert und verwendet wird
- Verbessert Lesbarkeit und Struktur komplexer Abfragen
- Kann rekursiv sein (RECURSIVE CTEs)

Grundsyntax

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

CTE – EINFACHES BEISPIEL

```sql
WITH order_totals AS (
    SELECT order_id, SUM(unit_price * quantity) AS total
    FROM order_details
    GROUP BY order_id
),
large_orders AS (
    SELECT *
    FROM order_totals
    WHERE total > 1000
)
SELECT *
FROM large_orders;
```

MEHRERE CTES KOMBINIEREN

```sql
WITH order_totals AS (
    SELECT order_id, SUM(unit_price * quantity) AS total
    FROM order_details
    GROUP BY order_id
),
large_orders AS (
    SELECT *
    FROM order_totals
    WHERE total > 1000
)
SELECT *
FROM large_orders;
```

REKURSIVE CTES

Was sind Rekursive CTEs?

- CTEs, die sich selbst referenzieren
- Ermöglichen rekursive Abfragen, z.B. Hierarchien
- Bestehen aus einem Anker-Teil und einem rekursiven Teil

Beispiele:

- Kategoriebaum
- Organisationsstruktur
- Bill of Materials (Stücklisten)
- Parent-Child-Tabellen

Grundstruktur

```sql
WITH RECURSIVE cte_name AS (
    -- Anker-Abfrage
    SELECT ...
    UNION ALL
    -- Rekursiver Teil
    SELECT ...
    FROM cte_name
    JOIN ...
)
SELECT *
FROM cte_name;
```

REKURSIVE CTE – HIERARCHIE-BEISPIEL

```sql
WITH RECURSIVE category_tree AS (
    SELECT category_id, category_name, parent_id, 1 AS level
    FROM categories
    WHERE parent_id IS NULL

    UNION ALL

    SELECT c.category_id, c.category_name, c.parent_id, ct.level + 1
    FROM categories c
    JOIN category_tree ct ON c.parent_id = ct.category_id
)
SELECT *
FROM category_tree
ORDER BY level, category_name;
```

REKURSIVE CTES – HINWEISE

Wichtig

- Immer eine saubere Abbruchbedingung
- Endlosschleifen vermeiden
- UNION ALL meist performanter als UNION

Typische Erweiterungen

- level / depth
- Pfad (path)
- Sortierung nach Hierarchie

BEST PRACTICES: CTE VS. SUBQUERY

Subquery verwenden, wenn

- Sehr einfache Logik
- Einmalige Berechnung
- Keine Wiederverwendung nötig

CTE verwenden, wenn

- Komplexe Logik
- Mehrere Verarbeitungsschritte
- Lesbarkeit & Wartbarkeit wichtig
- Schulungs- oder Team-Code

ZUSAMMENFASSUNG

- Subqueries: kompakt, aber schnell unübersichtlich
- CTEs: sauber, strukturiert, gut wartbar
- Rekursive CTEs: mächtig für Hierarchien
- Gute SQL ist lesbar, erklärbar & performant

ERWEITERTE AGGREGATION & ANALYSE

WARUM ERWEITERTE AGGREGATIONEN?

- Klassisches GROUP BY ist oft zu eingeschränkt
- Reporting & BI-Abfragen benötigen:
  - Mehrere Aggregationsebenen
  - Rangfolgen
  - Zeitliche Vergleiche
- Ziel: Weniger SQL, mehr Aussagekraft

WAS SIND GROUPING SETS?

- Erweiterung von GROUP BY
- Mehrere Gruppierungen in einer einzigen Abfrage
- Alternative zu:
  - Mehreren SELECTs
  - UNION ALL

Ergebnis:

- Umsatz pro Land & Stadt
- Umsatz pro Land
- Gesamtumsatz

Syntax

```
SELECT country, city, SUM(sales) AS total_sales
FROM orders
GROUP BY GROUPING SETS (
  (country, city),
  (country),
  ()
);
```

GROUPING SETS: VORTEILE & HINWEISE

Vorteile

- Deutlich weniger SQL
- Einheitliche Aggregation
- Bessere Performance als UNION ALL

Hinweise

- NULL zeigt aggregierte Ebenen an
- Mit GROUPING() unterscheidbar

```
GROUPING(country)
```

ROLLUP

Was ist ROLLUP?

- Hierarchische Aggregation
- Automatische Zwischensummen
- Typischer Einsatz:
  - Zeitdimensionen
  - Region → Land → Stadt

Ergebnis

- Umsatz pro Jahr & Monat
- Umsatz pro Jahr
- Gesamtumsatz

Beispiel

```
SELECT year, month, SUM(amount) AS revenue
FROM sales
GROUP BY ROLLUP (year, month);
```

CUBE

Was ist CUBE?

- Aggregiert alle Kombinationen
- Vollständige Kreuzaggregation
- Vorsicht: sehr viele Ergebniszeilen

Beispiel

```sql
SELECT region, product, SUM(amount) AS total
FROM sales
GROUP BY CUBE (region, product);
```

Ergebnis u.a.:

- Region + Produkt
- Nur Region
- Nur Produkt
- Gesamtsumme

ROLLUP VS CUBE

Feature	ROLLUP	CUBE
Hierarchisch	✅	❌
Alle Kombinationen	❌	✅
Ergebnisgröße	Klein	Groß
Typische Nutzung	Reports	Analysen

EINFÜHRUNG WINDOW FUNCTIONS

Was sind Window Functions?

- Berechnungen über Datenfenster
- Keine Reduktion der Zeilenanzahl
- Arbeiten nach WHERE & GROUP BY

Syntax

```
FUNCTION() OVER (PARTITION BY ... ORDER BY ...)
```

RANK & DENSE_RANK

Rangfunktionen

```sql
SELECT employee,
       salary,
       RANK() OVER (ORDER BY salary DESC) AS rank,
       DENSE_RANK() OVER (ORDER BY salary DESC) AS dense_rank
FROM employees;
```

Unterschied

- RANK: Lücken bei gleichen Werten
- DENSE_RANK: Keine Lücken

TYPISCHE EINSATZFÄLLE FÜR RANK

- Top-N Analysen
- Bestenlisten
- Vergleich innerhalb von Gruppen

Syntax

```
PARTITION BY department
```

ZUGRIFF AUF VORHERIGE / NÄCHSTE ZEILE

LAG

- Zugriff auf vorherige Zeile
- nützlich für Vergleiche oder Differenzen

LEAD

- Zugriff auf nächste Zeile

```sql
SELECT date,
       revenue,
       revenue - LAG(revenue) OVER (ORDER BY date) AS diff
FROM sales;
```

EINSATZ VON LAG / LEAD

- Zeitreihenanalysen
- Vergleich von aufeinanderfolgenden Datensätzen
- Erkennung von Trends oder Veränderungen
- Vorher-Nachher-Vergleiche

Optional

```
LAG(value, 2, 0)
```

Zugriff auf 2 Zeilen vorher, Standardwert 0

WINDOW FRAMES

- Welche Zeilen innerhalb des Fensters
- Für die Berechnung verwendet werden

ROWS FRAME

- Zeilenbasiert
- Exakte Anzahl von Zeilen

Beispiel

```
SUM(amount) OVER (
  ORDER BY date
  ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
)
```

RANGE FRAME

Eigenschaften

- Wertebasiert
- Abhängig vom Sortierschlüssel

Beispiel

```
SUM(amount) OVER (
  ORDER BY date
  RANGE BETWEEN INTERVAL '7 days' PRECEDING AND CURRENT ROW
)
```

ROWS VS RANGE

Merkmal	ROWS	RANGE
Basis	Zeilen	Werte
Deterministisch	✅	❌ (bei Duplikaten)
Performance	Besser	Teurer

TYPISCHE FEHLER & BEST PRACTICES

Fehler

- Zu große CUBEs
- Fehlendes ORDER BY bei Window Functions
- RANGE ohne eindeutige Sortierung

Best Practices

- ROLLUP für Reports
- CUBE nur gezielt einsetzen
- Window Functions statt Subqueries

ZUSAMMENFASSUNG

- GROUPING SETS: mehrere Aggregationen in einer Abfrage
- ROLLUP & CUBE: Hierarchie vs. Kombinationen
- Window Functions: Analyse ohne Zeilenverlust
- Unverzichtbar für Reporting & Analytics

FORTGESCHRITTENE CONSTRAINTS

- CHECK Constraints
- DEFERRABLE Constraints
- Exclusion Constraints
- VALIDATE Constraints

WAS IST EIN CHECK CONSTRAINT?

- Prüft boolesche Bedingungen beim INSERT oder UPDATE
- Bedingung muss TRUE oder UNKNOWN ergeben
- Bestandteil des Tabellen-Schemas

EIGENSCHAFTEN VON CHECK CONSTRAINTS

- Können eine oder mehrere Spalten betreffen
- Können benannt werden
- Werden pro Zeile geprüft
- Können nicht auf andere Tabellen zugreifen

BEISPIEL: EINFACHER CHECK CONSTRAINT

```sql
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    price NUMERIC(10,2),
    CHECK (price > 0)
);
```

BEISPIEL: BENANNTER CHECK CONSTRAINT

```sql
ALTER TABLE products
ADD CONSTRAINT chk_price_positive
CHECK (price > 0);
```

TYPISCHE EINSATZFÄLLE

- Wertebereiche (z. B. Alter, Preise)
- Statuswerte (z. B. status IN ('open','closed'))
- Abhängigkeiten zwischen Spalten

GRENZEN VON CHECK CONSTRAINTS

- Keine Subqueries
- Keine Tabellenübergreifenden Prüfungen
- Für komplexe Logik → Trigger oder Exclusion Constraints

WAS BEDEUTET DEFERRABLE CONSTRAINTS?

- Constraint-Prüfung kann verzögert werden
- Prüfung erfolgt:
  - sofort (IMMEDIATE)
  - oder erst beim COMMIT (DEFERRED)

WARUM DEFERRABLE CONSTRAINTS?

- Nützlich bei zyklischen Abhängigkeiten
- Erleichtert komplexe Transaktionen
- Verhindert temporäre Verletzungen während eines Transaktionsablaufs

BEISPIEL: DEFERRABLE FOREIGN KEY

```
CREATE TABLE orders (
    id INT PRIMARY KEY,
    customer_id INT,
    CONSTRAINT fk_customer
        FOREIGN KEY (customer_id)
        REFERENCES customers(id)
        DEFERRABLE INITIALLY DEFERRED
);
```

STEUERUNG ZUR LAUFZEIT

```
SET CONSTRAINTS fk_customer IMMEDIATE;
SET CONSTRAINTS ALL DEFERRED;
```

WICHTIGE HINWEISE

- Standard: NOT DEFERRABLE
- Nur für:
  - PRIMARY KEY
  - UNIQUE
  - FOREIGN KEY
  - EXCLUSION

WAS IST EIN EXCLUSION CONSTRAINT?

- Verhindert bestimmte Kombinationen von Werten
- Nutzt Index-Operatoren
- Sehr mächtig für Zeiträume & Überschneidungen

KLASSISCHER ANWENDUNGSFALL

- Keine überlappenden Zeiträume
- Raum- / Ressourcen-Belegung
- Termin- oder Buchungssysteme

BEISPIEL: KEINE ÜBERLAPPENDEN BUCHUNGEN

```
CREATE TABLE room_bookings (
    room_id INT,
    booking_period TSRANGE,
    EXCLUDE USING gist (
        room_id WITH =,
        booking_period WITH &&
    )
);
```

Erklärung

- room_id WITH = → gleicher Raum
- booking_period WITH && → Zeitraum überschneidet sich
- && = Overlap-Operator
- gist Index notwendig

VORTEILE VON EXCLUSION CONSTRAINTS

- Datenbank erzwingt komplexe Regeln
- Kein zusätzlicher Trigger nötig
- Sehr performant bei korrektem Index

EINSCHRÄNKUNGEN

- Etwas komplexe Syntax
- Benötigt passende Index-Typen (z. B. GiST)
- Weniger bekannt → erklärungsbedürftig

VALIDATE CONSTRAINT

Problemstellung

- Tabelle enthält bereits Daten
- Neues Constraint soll hinzugefügt werden
- Bestehende Daten könnten ungültig sein

LÖSUNG: NOT VALID

- Constraint wird hinzugefügt, aber nicht auf bestehende Daten angewendet
- Neue Daten müssen gültig sein
- Bestehende ungültige Daten bleiben bestehen

```
ALTER TABLE orders
ADD CONSTRAINT chk_amount_positive
CHECK (amount > 0)
NOT VALID;
```

NACHTRÄGLICHE VALIDIERUNG

- Prüft bestehende Daten
- Sperrt Tabelle nicht vollständig
- Produktionsfreundlich

```
ALTER TABLE orders
VALIDATE CONSTRAINT chk_amount_positive;
```

TYPISCHE EINSATZSZENARIEN

- Große Tabellen
- Migrationen
- Nachträgliche Qualitätssicherung
- Zero-Downtime-Deployments

BEST PRACTICES

- Constraints bevorzugen statt Trigger
- Constraints klar benennen
- Komplexität dokumentieren
- Performance & Wartbarkeit beachten

ZUSAMMENFASSUNG

Wann welches Constraint?

- CHECK → einfache fachliche Regeln
- DEFERRABLE → komplexe Transaktionen
- EXCLUSION → Überschneidungen & Konflikte
- NOT VALID / VALIDATE → sichere Migrationen

WAS SIND SEQUENCES?

Definition

- Sequences sind eigenständige Datenbankobjekte
- Erzeugen fortlaufende, eindeutige numerische Werte
- Häufige Verwendung: Primärschlüssel / IDs

Eigenschaften

- Tabellenunabhängig
- Nebenläufig (mehrere Sessions gleichzeitig)
- Transaktionsunabhängig

WARUM SEQUENCES VERWENDEN?

Probleme ohne Sequences

- Race Conditions bei MAX(id) + 1
- Locks und schlechte Performance

Vorteile von Sequences

- Garantierte Eindeutigkeit
- Hohe Performance
- Keine Konflikte bei parallelen Inserts

SEQUENCE ERSTELLEN

Syntax:

```
CREATE SEQUENCE customer_seq
```

Mit Optionen:

```
CREATE SEQUENCE customer_seq
  START WITH 1
  INCREMENT BY 1
  MINVALUE 1
  CACHE 20;
```

Wichtige Optionen

- **START WITH**: Anfangswert der Sequence
- **INCREMENT BY**: Schrittweite (Standard: 1)
- **MINVALUE / MAXVALUE**: Grenzen der Sequence
- **CACHE**: Anzahl vorab reservierter Werte für Performance

SEQUENCE VERWENDEN

Nächsten Wert abrufen:

```
SELECT NEXTVAL('customer_seq');
```

Aktuellen Wert abrufen:

```
SELECT CURRVAL('customer_seq');
```

⚠️ Nur gültig nach nextval() in derselben Session

Wert setzen

```
SELECT SETVAL('customer_seq', 100);
```

SEQUENCES & TRANSAKTIONEN

- Sequences sind transaktionsunabhängig
- nextval() erhöht die Sequence sofort
- Rollback einer Transaktion beeinflusst die Sequence nicht
- Kann zu Lücken in der Sequenz führen

```
BEGIN;
SELECT nextval('customer_seq');
ROLLBACK;
```

Ergebnis:

- Wert ist verbraucht
- Keine Rücksetzung bei Rollback

VERWENDUNG IN TABELLEN

Manuell

```
INSERT INTO customers (id, name)
VALUES (nextval('customer_seq'), 'Alice');
```

Automatisch mit DEFAULT

```
CREATE TABLE customers (
  id INTEGER DEFAULT nextval('customer_seq'),
  name TEXT
);
```

SERIAL VS IDENTITY

SERIAL (Legacy)

- PostgreSQL-spezifisch
- Erstellt implizit eine Sequence

```
id SERIAL
```

IDENTITY (empfohlen)

- SQL-Standard
- Bessere Kontrolle & Klarheit

```
id INT GENERATED ALWAYS AS IDENTITY
```

BEST PRACTICES

- Sequences für technische IDs
- Nicht für fachliche Nummern (Rechnung, Beleg)
- Keine Annahme von Lückenfreiheit
- IDENTITY statt SERIAL bevorzugen

ZUSAMMENFASSUNG

- Sequences erzeugen eindeutige Zahlen
- Nebenläufig & performant
- Transaktionsunabhängig
- Standardlösung für Primärschlüssel

DATENAKTUALISIERUNG (SQL-EBENE)

INSERT / UPDATE / DELETE (ADVANCED USAGE)

INSERT

- Einfügen mehrerer Zeilen in einem Statement
- Kombination mit SELECT
- Nutzung der RETURNING-Klausel

UPDATE

- Aktualisierung über Joins (UPDATE … FROM)
- Nutzung von Bedingungen und Subqueries

DELETE

- Löschen mit Referenz auf andere Tabellen (DELETE … USING)

👉 Fokus: Mehr als nur einfache Einzelzeilen-Operationen

INSERT … SELECT

Einsatzszenarien

- Daten aus anderen Tabellen übernehmen
- Migrationen & Archivierung
- Duplizieren oder Transformieren von Daten

Beispiel

```sql
INSERT INTO archived_orders (id, customer_id, order_date)
SELECT id, customer_id, order_date
FROM orders
WHERE order_date < '2023-01-01';
```

Vorteile

- Sehr performant
- Keine Schleifen notwendig
- Ideal für große Datenmengen

UPDATE … FROM

Warum UPDATE … FROM?

- Aktualisierung basierend auf anderen Tabellen
- Vermeidet komplexe Subqueries

Beispiel

```sql
UPDATE products p
SET price = p.price * 1.1
FROM categories c
WHERE p.category_id = c.id
  AND c.name = 'Electronics';
```

Typische Anwendungsfälle

- Preis- oder Statusanpassungen
- Synchronisation von Stammdaten

DELETE … USING

DELETE mit Join-Logik

- Löschen abhängig von Daten in anderen Tabellen

Beispiel

```
DELETE FROM orders o
USING customers c
WHERE o.customer_id = c.id
  AND c.is_inactive = true;
```

Vorteile

- Klarer als verschachtelte Subqueries
- Set-basiert & performant

⚠️ Vorsicht bei fehlenden WHERE-Bedingungen!

RETURNING-KLAUSEL

Was macht RETURNING?

- Gibt betroffene Datensätze direkt zurück
- Funktioniert mit INSERT, UPDATE, DELETE

Beispiel

```
UPDATE users
SET last_login = now()
WHERE id = 42
RETURNING id, last_login;
```

Einsatz

- Logging
- Weiterverarbeitung in Anwendungen
- Debugging

MERGE (UPSERT)

Was ist MERGE?

- Kombination aus INSERT und UPDATE
- SQL-Standard (ab PostgreSQL 15)

Beispiel

```
MERGE INTO inventory i
USING new_inventory n
ON i.product_id = n.product_id
WHEN MATCHED THEN
  UPDATE SET quantity = n.quantity
WHEN NOT MATCHED THEN
  INSERT (product_id, quantity)
  VALUES (n.product_id, n.quantity);
```

Vorteile

- Klarer als INSERT … ON CONFLICT
- Gut lesbar bei komplexen Logiken

MEHRZEILIGE & SET-BASIERTE UPDATES

Set-basiert statt zeilenweise

- Ein SQL-Statement für viele Datensätze
- Vermeidet Cursor & Loops

Beispiel

```
UPDATE orders
SET status = 'archived'
WHERE order_date < now() - interval '2 years';
```

Warum wichtig?

- Bessere Performance
- Weniger Lock-Overhead
- Einfachere Wartung

TRANSAKTIONSSICHERHEIT BEI DATENÄNDERUNGEN

Grundlagen

- BEGIN, COMMIT, ROLLBACK
- Änderungen sind atomar

Beispiel

```sql
BEGIN;

UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;

COMMIT;
```

Best Practices

- Mehrere zusammengehörige Änderungen immer in Transaktionen
- Fehler gezielt abfangen und zurückrollen

BEST PRACTICES (DATENAKTUALISIERUNG)

Performance

- ✅ Set-basierte Updates sind in der Regel performanter als zeilenweise Verarbeitung
- Indizes für WHERE-Bedingungen nutzen
- Große Updates ggf. in Batches aufteilen

Sicherheit

- Immer mit WHERE-Klausel arbeiten
- Vorher SELECT zur Kontrolle ausführen
- Transaktionen bewusst einsetzen

Wartbarkeit

- Lesbare SQL-Statements
- Klare Trennung von Logik und Daten

ZUSAMMENFASSUNG

- PostgreSQL bietet mächtige Erweiterungen für Datenänderungen
- INSERT … SELECT, UPDATE … FROM, DELETE … USING sind essenziell
- RETURNING spart zusätzliche Abfragen
- MERGE vereinfacht Upsert-Logiken
- Set-basierte Verarbeitung ist performant, sicher und skalierbar
- Transaktionen schützen vor inkonsistenten Daten

👉 Saubere Datenänderungen sind genauso wichtig wie saubere Abfragen

POSTGRESQL-SPEZIFISCHE DATENTYPEN

Warum spezielle Datentypen?

- PostgreSQL bietet Datentypen über den SQL-Standard hinaus
- Ziel:
  - bessere Datenmodellierung
  - weniger Workarounds
  - mehr Typensicherheit & Ausdrucksstärke
- Viele Operationen sind direkt im Typ integriert (Operatoren, Funktionen)

➡️ „Daten möglichst nah an der fachlichen Bedeutung modellieren“

ARRAYS IN POSTGRESQL

Was sind Arrays?

- Ein Feld kann mehrere Werte desselben Typs enthalten
- Beliebige Dimensionen möglich (1D, 2D, …)

```
tags TEXT[];
scores INT[];
```

Vorteile

- Einfaches Speichern von Listen
- Kein zusätzliches Join-Table nötig

Nachteile

- Begrenzte Normalisierung
- Vorsicht bei komplexen Abfragen

ARRAY-OPERATOREN & FUNKTIONEN

Zugriff & Abfragen

```
SELECT tags[1] FROM articles;
```

Wichtige Operatoren

- ANY / ALL
- @> enthält
- <@ ist enthalten in
- && überschneidet sich

```
SELECT *
FROM articles
WHERE tags @> ARRAY['postgres'];
```

Best Practice

- Arrays eher für kleine, feste Listen
- Bei häufigen Abfragen: GIN-Index nutzen

COMPOSITE TYPES

Was sind Composite Types?

- Benutzerdefinierte strukturierte Datentypen
- Vergleichbar mit „Mini-Records“

```
CREATE TYPE address_type AS (
  street TEXT,
  city   TEXT,
  zip    TEXT
);

address address_type
```

Vorteile

- Klare Struktur
- Wiederverwendbar
- Typisch für fachliche Modelle

ARBEITEN MIT COMPOSITE TYPES

Zugriff auf Felder

```
SELECT (address).city
FROM customers;
```

Einsatzszenarien

- Adressen
- Koordinaten
- Fachliche Value Objects

⚠️ Änderungen am Typ wirken sich auf alle Tabellen aus

ENUM VS. CHECK CONSTRAINT

ENUM

```
CREATE TYPE order_status AS ENUM (
  'open', 'paid', 'shipped'
);
```

Vorteile

- Klare erlaubte Werte
- Gute Lesbarkeit
- Platzsparend

Nachteile

- Änderungen nur mit ALTER TYPE
- Migrationen aufwendiger

CHECK Constraint

```
status TEXT CHECK (status IN ('open','paid','shipped'))
```

Vorteile

- Flexibler
- Einfach erweiterbar

Nachteile

- Weniger strikt
- Kein echter Typ

ENTSCHEIDUNG: ENUM ODER CHECK?

Faustregel

Kriterium	ENUM	CHECK
Feste Werte	✅	⚠️
Häufige Änderungen	❌	✅
Typensicherheit	✅	❌
Migrationen	Aufwendig	Einfach

➡️ ENUM für stabile Fachzustände,

➡️ CHECK für flexible Regeln

RANGE TYPES

Was sind Range Types?

- Abbildung eines Wertebereichs
- Inklusive / exklusive Grenzen

Beispiele:

- int4range
- numrange
- tsrange, tstzrange
- daterange

```
valid_period daterange;
```

ARBEITEN MIT RANGE TYPES

```sql
SELECT *
FROM bookings
WHERE valid_period @> CURRENT_DATE;
```

Wichtige Operatoren

- @> enthält
- && überlappt
- -|- grenzt an

Typische Einsatzfälle

- Terminplanung
- Gültigkeitszeiträume
- Ressourcenbelegung

➡️ Sehr mächtig in Kombination mit Exclusion Constraints

UUIDS IN POSTGRESQL

Was ist eine UUID?

- Universell eindeutiger Identifikator
- 128 Bit
- Keine zentrale ID-Vergabe nötig

```
id UUID DEFAULT gen_random_uuid()
```

(Extension: pgcrypto)

UUID VS. SERIAL / IDENTITY

Vorteile von UUID

- Global eindeutig
- Sicher für verteilte Systeme
- Keine Kollisionen

Nachteile

- Größer als INT
- Indizes langsamer
- Weniger lesbar

➡️ Empfehlung:

- UUID: Microservices, externe IDs
- IDENTITY: interne, hochfrequente Tabellen

BEST PRACTICES: POSTGRESQL-DATENTYPEN

- PostgreSQL-Typen bewusst einsetzen
- Fachliche Bedeutung im Schema abbilden
- Arrays nicht als Ersatz für saubere Relationen
- ENUM nur bei stabilen Wertemengen
- Range Types für Zeiträume statt from / to
- UUIDs mit Bedacht verwenden

ZUSAMMENFASSUNG

- PostgreSQL bietet mächtige, erweiterte Datentypen
- Sie verbessern:
  - Lesbarkeit
  - Datenqualität
  - Abfragefähigkeit
- Richtig eingesetzt → weniger Logik im Code, mehr in der DB

👉 „Das Datenmodell ist Teil der Anwendung – nicht nur Speicher“

JSON & JSONB DEEP DIVE

Warum JSON in PostgreSQL?

- Flexible, schemafreie Daten innerhalb relationaler Tabellen
- Ideal für semi-strukturierte Daten
- Kombination aus SQL + JSON-Abfragen möglich

JSON VS. JSONB – ÜBERBLICK

Merkmal	JSON	JSONB
Speicherung	Textbasiert	Binärformat
Schlüsselreihenfolge	Bleibt erhalten	Wird normalisiert
Whitespace	Bleibt erhalten	Entfernt
Duplikate Keys	Erlaubt	Letzter Key gewinnt
Performance Abfragen	Langsamer	Deutlich schneller
Indexierbar	❌ Nein	✅ Ja

Merksatz:

👉 JSONB fast immer bevorzugen, JSON nur bei Spezialfällen (z. B. exakte Texttreue).

JSON VS. JSONB – BEISPIEL

```
SELECT
  '{"a": 1, "b": 2}'::json,
  '{"b": 2, "a": 1}'::jsonb;
```

Beobachtung

- JSON: Reihenfolge bleibt erhalten
- JSONB: Struktur wird intern normalisiert

ZUGRIFF AUF JSON / JSONB – OPERATOREN

Operator	Bedeutung
->	JSON-Wert (Objekt / Array)
->>	Textwert
#>	Pfad als JSON
#>>	Pfad als Text

PFADABFRAGEN – BASIS

```
SELECT data->'address'->>'city'
FROM customers;
```

Erklärung

- data->'address' gibt das JSON-Objekt zurück
- ->>'city' extrahiert den Textwert
- Sehr wichtig für Vergleiche & WHERE-Klauseln

PFADABFRAGEN MIT ARRAYS

```
SELECT data->'orders'->0->>'id'
FROM customers;
```

Erklärung

- data->'orders' gibt das Array zurück
- ->0 greift auf das erste Element zu
- ->>'id' extrahiert den Textwert des id-Felds
- Funktioniert für beliebige JSON-Tiefen

JSONPATH (AB POSTGRESQL 12)

```sql
SELECT *
FROM orders
WHERE data @? '$.items[*] ? (@.price > 100)';
```

Erklärung

- JSONPath ermöglicht komplexe Abfragen auf JSON-Strukturen
- Syntax ähnlich zu XPath / JavaScript
- Sehr mächtig für tief verschachtelte Daten

JSONB – SUCHOPERATOREN

Operator	Bedeutung
@>	Enthält
<@	Ist enthalten
?	Schlüssel existiert
?`	`
?&	Alle Keys vorhanden

BEISPIEL – ENTHÄLT-ABFRAGE

```sql
SELECT *
FROM users
WHERE profile @> '{"role": "admin"}';
```

Erklärung

- profile @> '{"role": "admin"}' prüft, ob das JSONB-Feld profile das angegebene Objekt enthält
- Sehr effizient mit GIN-Index auf profile

👉 Ideal für flexible Berechtigungsprüfungen

INDEXIERUNG – WARUM ENTSCHEIDEND?

❌ **Ohne Index:**

- Vollständige Tabellen-Scans
- Langsame JSON-Abfragen

✅ **Mit Index:**

- Performante Filter
- Skalierbar auch bei großen Datenmengen

GIN-INDEX AUF JSONB

```
CREATE INDEX idx_users_profile
ON users
USING GIN (profile);
```

Erklärung

- GIN-Index ist speziell für komplexe Datentypen wie JSONB optimiert
- Beschleunigt Operatoren wie @>, ?, ?|, ?&
- Unverzichtbar für performante JSONB-Abfragen

GIN MIT JSONB_PATH_OPS

```sql
CREATE INDEX idx_users_profile_path
ON users
USING GIN (profile jsonb_path_ops);
```

Erklärung

- jsonb_path_ops optimiert für @> Operator
- Geringerer Speicherbedarf als Standard-GIN
- Nur für bestimmte Abfragearten geeignet

👉 Auswahl des Index-Typs abhängig von den häufigsten Abfragen

INDEX-TYPEN – VERGLEICH

Index	Vorteil	Nachteil
GIN	Flexibel, viele Operatoren	Größer
GIN (path_ops)	Schnell & klein	Eingeschränkt
Expression Index	Sehr gezielt	Nur für definierte Pfade

TYPISCHE USE-CASES

- Konfigurationsdaten
- Feature Flags
- Dynamische Attribute
- Event-Daten / Logs
- Externe API-Payloads

HYBRID-MODELL – BEST PRACTICE

```sql
CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  customer_id INT,
  status TEXT,
  metadata JSONB
);
```

Erklärung

- Feste Spalten für häufig abgefragte Daten
- JSONB-Feld für flexible, selten abgefragte Informationen
- Beste Performance & Flexibilität kombiniert

👉 „Das Beste aus beiden Welten“

ANTI-PATTERNS – WAS VERMEIDEN?

- JSONB als Ersatz für Tabellen
- JSON-Felder für JOIN-relevante Daten
- Keine Indizes auf JSONB
- Stark verschachtelte Strukturen
- Business-Logik in JSON verstecken

TYPISCHER FEHLER

```
WHERE data->'price' > 100
```

Vergleich von JSONB-Werten ohne Typumwandlung führt zu Fehlern

Richtige Lösung:

```
WHERE (data->>'price')::numeric > 100
```

BEST PRACTICES

- JSONB statt JSON verwenden
- Pfade bewusst wählen
- Indizes gezielt setzen
- Struktur regelmäßig hinterfragen
- JSONB ≠ Schemafreiheit ohne Verantwortung

ZUSAMMENFASSUNG

- JSONB ist der Standard in PostgreSQL
- Leistungsfähige Operatoren & JSONPath
- Indexierung ist Pflicht
- Ideal für semi-strukturierte Daten
- Relationale Stärken nicht ersetzen, sondern ergänzen

JSONPATH IN WHERE-BEDINGUNGEN

Seit PostgreSQL 12 unterstützt PostgreSQL JSONPath.

Verwendung in WHERE über:

- @? → Existiert ein Treffer?
- @@ → Liefert der Ausdruck TRUE oder FALSE basierend auf dem Treffer

@? VS @@ – DER UNTERSCHIED

Operator	Bedeutung	Rückgabe
@?	Prüft, ob ein JSONPath Treffer existiert	TRUE/FALSE
@@	Prüft, ob JSONPath-Ausdruck TRUE ergibt	TRUE/FALSE

BEISPIEL MIT @?

```
SELECT *
FROM person
WHERE info @? '$.age ? (@ > 25)';
```

➞ TRUE, wenn mindestens ein Treffer existiert

BEISPIEL MIT @@

```sql
SELECT *
FROM person
WHERE info @@ '$.age > 25';
```

➞ Der Ausdruck selbst muss TRUE ergeben

👉 In einfachen Fällen oft gleichwertig.

ROOT ($) & CURRENT (@) IM WHERE-KONTEXT

```
WHERE info @? '$.pets[*] ? (@.species == "Dog")'
```

Bedeutung:

- $ → Root-Dokument
- pets[*] → Iteration über Array
- @ → aktuelles Element
- Filter ? (...) → Bedingung

VERGLEICHSOPERATOREN IN JSONPATH

Operator	Bedeutung
`==`	Gleich
`!=`	Ungleich
`> < >= <=`	Numerische Vergleiche: Größer als / Kleiner als / Größer gleich / Kleiner gleich
`like_regex`	Regulärer Ausdruck
`&&`	Logisches UND

KOMBINIERTE BEDINGUNGEN

📦 Beispiel-Datensatz

```
{
  "name": "John",
  "age": 30,
  "city": "New York",
  "pets": [
    { "name": "Max", "species": "Dog" }
  ]
}
```

🔎 SQL

```
SELECT name
FROM person
WHERE info @@ '
  $.age >= 18 &&
  $.city == "New York" &&
  $.pets[*] ? (@.species == "Dog")
';
```

✅ Ergebnis

name
John

💡 Warum?

Bedingung	Bewertung
age >= 18	TRUE
city == "New York"	TRUE
$.pets[*] ? (@.species == "Dog")	TRUE

KOMBINIERTE BEDINGUNGEN – NEGATIVBEISPIEL

📦 Beispiel-Datensatz

```
{
  "name": "Tom",
  "age": 16,
  "city": "New York",
  "pets": [
    { "name": "Rex", "species": "Dog" }
  ]
}
```

🔎 Gleiches SQL wie zuvor

❌ Ergebnis

name

(keine Zeilen)

ARBEITEN MIT ARRAYS

Existenz eines Elements

```
WHERE info @? '$.pets[*] ? (@.species == "Cat")';
```

Alle Elemente müssen passen

```
WHERE info @@ 'all($.pets[*].species == "Dog")';
```

Anzahl prüfen

```
WHERE info @@ '$.pets.size() >= 2';
```

ARBEITEN MIT ARRAYS - EXISTENZ EINES ELEMENTS

📦 Beispiel-Datensatz in person.info

```
{
  "name": "John",
  "age": 30,
  "city": "New York",
  "pets": [
    { "name": "Max", "species": "Dog" },
    { "name": "Whiskers", "species": "Cat" }
  ]
}
```

🔎 SQL

```
SELECT name
FROM person
WHERE info @? '$.pets[*] ? (@.species == "Cat")';
```

✅ Ergebnis

name
John

💡 Warum?

- pets[*] → alle Haustiere
- Filter sucht species == "Cat"
- „Whiskers“ ist eine Katze
- Mindestens ein Treffer → TRUE

ARBEITEN MIT ARRAYS – KEIN TREFFER

📦 Beispiel-Datensatz

```
{
  "name": "Sarah",
  "age": 28,
  "city": "Berlin",
  "pets": [
    { "name": "Bello", "species": "Dog" }
  ]
}
```

🔎 SQL

```
SELECT name
FROM person
WHERE info @? '$.pets[*] ? (@.species == "Cat")';
```

❌ Ergebnis

name
(keine Zeilen)

💡 Warum?

- Es existiert kein Pet mit species == "Cat"
- JSONPath liefert FALSE

ARBEITEN MIT ARRAYS - ANZAHL PRÜFEN MIT SIZE()

📦 Beispiel-Datensatz

```
{
  "name": "John",
  "pets": [
    { "name": "Max", "species": "Dog" },
    { "name": "Whiskers", "species": "Cat" }
  ]
}
```

🔎 SQL

```
SELECT name
FROM person
WHERE info @@ '$.pets.size() >= 2';
```

✅ Ergebnis

name
John

💡 Warum?

- pets-Array hat 2 Elemente
- Bedingung >= 2 erfüllt

JSONPATH MIT VARIABLEN

PostgreSQL unterstützt Variablen:

```
SELECT *
FROM person
WHERE jsonb_path_exists(
    info,
    '$.age > $min_age',
    '{"min_age": 25}'
);
```

Vorteile:

- Dynamisch
- Sicher
- Kein String-Bauen nötig

JSONB_PATH_EXISTS VS OPERATOREN

Funktion	Beschreibung
`jsonb_path_exists()`	Entspricht @?
`jsonb_path_match()`	Entspricht @@
`jsonb_path_query()`	Gibt JSON-Werte zurück

TYPISCHE FEHLERQUELLEN

- ❌ String-Vergleich bei Zahlen
- ❌ @ außerhalb eines Filters
- ❌ Vergessen von $
- ❌ Kein GIN-Index vorhanden
- ❌ Zu komplexe verschachtelte Pfade

PERFORMANCE IN WHERE

JSONPath profitiert von:

```
CREATE INDEX idx_person_info
ON person
USING GIN (info jsonb_path_ops);
```

Besonders effizient für:

- @?
- @@
- jsonb_path_exists()

WANN JSONPATH STATT @> VERWENDEN?

Verwende @>	Verwende JSONPath
Strukturvergleich	Komplexe Bedingungen
Exakte Teilstruktur	Numerische Filter
Key-Existenz	Array-Filter
Einfache Abfragen	Logische Operatoren

BEISPIEL – KOMPLEXE WHERE-KLAUSEL

```
SELECT *
FROM person
WHERE info @@ '
    $.age >= 18 &&
    $.pets[*] ? (@.species == "Dog") &&
    $.city == "New York"
';
```

➞ Kombination aus:

- Numerischem Vergleich
- Array-Filter
- String-Vergleich

BEST PRACTICES FÜR JSONPATH IN WHERE

- JSONB verwenden (nicht JSON)
- GIN-Index setzen
- Logik nicht überkomplex machen
- Bei einfachen Strukturvergleichen lieber @> nutzen
- JSONPath gezielt einsetzen, nicht inflationär

ZUSAMMENFASSUNG JSONPATH IN WHERE

- $ → Root
- @ → aktuelles Element
- @? → Existenz
- @@ → TRUE-Bedingung
- Sehr mächtig für Array-Filter & komplexe Bedingungen
- Performance hängt stark von Indexierung ab

FUNKTIONEN & PROZEDUREN

- Werden im Server ausgeführt → sehr performant
- Können SQL oder PL/pgSQL verwenden
- Dienen zur Kapselung von Logik, Wiederverwendbarkeit
- Unterschied:
  - FUNCTION → gibt immer etwas zurück (mindestens NULL)
  - PROCEDURE → gibt nichts zurück, aber erlaubt TRANSAKTIONALE Befehle (COMMIT, ROLLBACK)

WARUM BENÖTIGT POSTGRESQL $$ IN FUNKTIONEN? (DOLLAR-QUOTING)

- Funktionscode (PL/pgSQL) wird in PostgreSQL als Textblock/String gespeichert
- SQL-Parser muss den Funktionskörper klar abgrenzen können
- Dollar-Quoting ($$ ... $$) definiert einen String ohne Escape-Zeichen
- Vorteil:
  - Kein Konflikt mit einfachen Hochkommas '...'
  - Kein Maskieren nötig (z. B. für 'Test')
  - Übersichtlicher bei komplexer Logik
- Wird verwendet bei:
  - Funktionen
  - Prozeduren
  - Trigger-Funktionen
  - DO-Blöcken
- Syntax:

```
AS $$
BEGIN
  -- Funktionscode
END;
$$ LANGUAGE plpgsql;
```

FUNKTIONSWEISE VON DOLLAR-QUOTING & BEISPIELE

Was passiert intern?

- PostgreSQL speichert alles zwischen $$ ... $$ als String unverändert
- Dieser String wird an den PL/pgSQL-Interpreter weitergegeben
- Dadurch beeinflusst der SQL-Parser den Inhalt nicht mehr

❌ Problem ohne Dollar-Quoting

```
AS '
BEGIN
  RAISE NOTICE 'Hallo';
END;
'
```

→ " 'Hallo' " kollidiert mit den äußeren ' ... ' → Parserfehler

✔️ Lösung mit Dollar-Quoting:

```
AS $$
BEGIN
  RAISE NOTICE 'Hallo';
END;
$$;
```

💡 Alternative Quoting-Varianten

```
AS $BODY$
BEGIN
  RAISE NOTICE 'Dollarzeichen: $$';
END;
$BODY$;
```

FUNKTIONSARTEN

FUNKTIONSARTEN

RETURNS VOID

- Funktion führt nur Aktionen aus
- Kein Rückgabewert
- Beispiel:

```
CREATE FUNCTION log_msg(msg TEXT)
RETURNS VOID AS $$
BEGIN
  RAISE NOTICE 'Log: %', msg;
END;
$$ LANGUAGE plpgsql;
```

FUNKTIONSARTEN

RETURNS VOID

- Funktion führt nur Aktionen aus
- Kein Rückgabewert
- Beispiel:

```
CREATE FUNCTION log_msg(msg TEXT)
RETURNS VOID AS $$
BEGIN
  RAISE NOTICE 'Log: %', msg;
END;
$$ LANGUAGE plpgsql;
```

RETURNS SCALAR (einzelner Wert)

- Gibt z. B. einen Text, Zahl, Datum zurück
- Beispiel:

```
CREATE FUNCTION add_tax(amount NUMERIC)
RETURNS NUMERIC AS $$
BEGIN
  RETURN amount * 1.19;
END;
$$ LANGUAGE plpgsql;
```

FUNKTIONSARTEN

RETURNS VOID

- Funktion führt nur Aktionen aus
- Kein Rückgabewert
- Beispiel:

```
CREATE FUNCTION log_msg(msg TEXT)
RETURNS VOID AS $$
BEGIN
  RAISE NOTICE 'Log: %', msg;
END;
$$ LANGUAGE plpgsql;
```

RETURNS TABLE (...)

- Gibt mehrere Spalten/Zeilen zurück
- Wie eine virtuelle Tabelle
- Kann in SELECT eingebunden werden
- Beispiel:

```
CREATE FUNCTION get_products()
RETURNS TABLE(id INT, name TEXT) AS $$
BEGIN
  RETURN QUERY SELECT id, name FROM product;
END;
$$ LANGUAGE plpgsql;
```

RETURNS SCALAR (einzelner Wert)

- Gibt z. B. einen Text, Zahl, Datum zurück
- Beispiel:

```
CREATE FUNCTION add_tax(amount NUMERIC)
RETURNS NUMERIC AS $$
BEGIN
  RETURN amount * 1.19;
END;
$$ LANGUAGE plpgsql;
```

FUNKTIONSARTEN

RETURNS VOID

- Funktion führt nur Aktionen aus
- Kein Rückgabewert
- Beispiel:

```
CREATE FUNCTION log_msg(msg TEXT)
RETURNS VOID AS $$
BEGIN
  RAISE NOTICE 'Log: %', msg;
END;
$$ LANGUAGE plpgsql;
```

RETURNS SCALAR (einzelner Wert)

- Gibt z. B. einen Text, Zahl, Datum zurück
- Beispiel:

```
CREATE FUNCTION add_tax(amount NUMERIC)
RETURNS NUMERIC AS $$
BEGIN
  RETURN amount * 1.19;
END;
$$ LANGUAGE plpgsql;
```

RETURNS TABLE (…)

- Gibt mehrere Spalten/Zeilen zurück
- Wie eine virtuelle Tabelle
- Kann in SELECT eingebunden werden
- Beispiel:

```
CREATE FUNCTION get_products()
RETURNS TABLE(id INT, name TEXT) AS $$
BEGIN
  RETURN QUERY SELECT id, name FROM product;
END;
$$ LANGUAGE plpgsql;
```

RETURNS RECORD

- Flexibler, aber benötigt Typangabe beim SELECT
- Beispiel:

```
CREATE FUNCTION get_stats()
RETURNS RECORD AS $$
DECLARE result RECORD;
BEGIN
  SELECT COUNT(*), MAX(price) INTO result FROM product;
  RETURN result;
END;
$$ LANGUAGE plpgsql;
```

IMMUTABLE / STABLE / VOLATILE

IMMUTABLE

- Funktion liefert immer das gleiche Ergebnis für gleiche Eingaben
- Keine Datenbankzugriffe
- Keine Datenbankzugriffe
- Optimierbar (z. B. Vorberechnung im Query-Planer)

STABLE

- Ergebnis bleibt innerhalb einer Query gleich
- Darf lesen, aber nicht schreiben
- Typisches Beispiel: CURRENT_DATE

VOLATILE

- Ergebnis kann sich jederzeit ändern
- Darf Tabellen lesen & schreiben
- Beispiel: random(), NOW()

PARAMETERARTEN

IN (Standard)

- Werte werden an die Funktion übergeben

OUT

- Parameter dient als Rückgabewert
- Funktion benötigt kein RETURNS mehr

INOUT

- Kombination: Wert wird übergeben und geändert zurückgegeben

VARIADIC

- Variable Anzahl an Parametern
- Beispiel:

```
CREATE FUNCTION sum_all(VARIADIC nums INT[])
RETURNS INT AS $$
BEGIN
  RETURN (SELECT SUM(n) FROM UNNEST(nums) n);
END;
$$ LANGUAGE plpgsql;
```

FUNKTIONSOVERLOADING

- PostgreSQL erlaubt mehrere Funktionen mit gleichem Namen
- Die Parameterliste entscheidet, welche Funktion genutzt wird
- Vergleichbar mit Overloading in Java, C#, etc.

Beispiel:

```
CREATE FUNCTION get_tax(amount NUMERIC)
RETURNS NUMERIC AS $$ SELECT amount * 0.19 $$ LANGUAGE sql;

CREATE FUNCTION get_tax(amount NUMERIC, tax NUMERIC)
RETURNS NUMERIC AS $$ SELECT amount * tax $$ LANGUAGE sql;
```

Aufruf:

```
SELECT get_tax(100);       -- nutzt 19%
SELECT get_tax(100, 0.07); -- nutzt 7%
```

PROZEDUREN

Beispiel:

```
CREATE PROCEDURE transfer_money(p_from INT, p_to INT, p_amount NUMERIC)
LANGUAGE plpgsql
AS $$
BEGIN
  UPDATE account SET balance = balance - p_amount WHERE id = p_from;
  UPDATE account SET balance = balance + p_amount WHERE id = p_to;

  COMMIT;
END;
$$;
```

Aufruf:

```
CALL transfer_money(1, 2, 50);
```

BEST PRACTICES FÜR FUNKTIONEN

- Möglichst IMMUTABLE oder STABLE nutzen für bessere Performance
- Nur PL/pgSQL verwenden, wenn SQL allein nicht reicht
- Funktionen ohne Seiteneffekte bevorzugen
- Für komplexe Rückgaben → RETURNS TABLE statt RECORD
- Funktion überarbeiten → Versionierung durch Overloading möglich
- Fehlerbehandlung mittels EXCEPTION-Blöcken

LANGUAGE IN FUNKTIONEN & PROZEDUREN

In PostgreSQL muss bei jeder FUNCTION und PROCEDURE angegeben werden, in welcher Sprache der Funktionscode geschrieben ist.

LANGUAGE SQL

- Reiner SQL-Code (eine einzelne SQL-Anweisung)
- Sehr performant, minimaler Overhead
- Ideal für:
  - einfache Berechnungen
  - reine SELECTs
  - Views-ähnliche Logik
- Keine:
  - Variablen
  - IF / LOOP
  - Exception-Behandlung

Beispiel:

```
CREATE FUNCTION product_count()
RETURNS INT
AS $$
  SELECT COUNT(*) FROM products;
$$ LANGUAGE sql;
```

LANGUAGE PLPGSQL

- Prozedurale Erweiterung von SQL
- Ermöglicht:
  - Variablen (DECLARE)
  - Kontrollstrukturen (IF, LOOP, CASE)
  - Exception Handling
  - Mehrere SQL-Statements
- Standard für komplexe Logik
- Etwas mehr Overhead als sql

Beispiel:

```
CREATE FUNCTION product_count_safe()
RETURNS INT
AS $$
DECLARE
  cnt INT;
BEGIN
  SELECT COUNT(*) INTO cnt FROM products;
  RETURN cnt;
END;
$$ LANGUAGE plpgsql;
```

BEST PRACTICES

- LANGUAGE sql bevorzugen, wenn:
  - eine einzige SQL-Anweisung ausreicht
  - keine Logik nötig ist
- LANGUAGE plpgsql verwenden, wenn:
  - Logik, Bedingungen oder Schleifen nötig sind
  - Fehler behandelt werden müssen
- **Nicht vorschnell plpgsql verwenden „weil man es immer so macht“**

BEGIN & END IN FUNKTIONEN & PROZEDUREN

LANGUAGE SQL → KEIN BEGIN … END

Bei LANGUAGE sql besteht der Funktionskörper aus genau einer SQL-Anweisung.

- Kein Block
- Kein prozeduraler Code
- Kein Kontrollfluss

Beispiel (ohne BEGIN / END)

```
CREATE FUNCTION product_count()
RETURNS INT
AS $$
  SELECT COUNT(*) FROM products;
$$ LANGUAGE sql;
```

LANGUAGE PLPGSQL → BEGIN … END FAST IMMER NÖTIG

plpgsql ist prozedural → es arbeitet mit Code-Blöcken.

- Variablen
- IF / LOOP / CASE
- Mehrere SQL-Statements
- Exception Handling

➡️ Dafür braucht PostgreSQL einen **Blockrahmen:**

```
BEGIN
  ...
END;
```

Beispiel:

```
CREATE FUNCTION product_count_safe()
RETURNS INT
AS $$
DECLARE
  cnt INT;
BEGIN
  SELECT COUNT(*) INTO cnt FROM products;
  RETURN cnt;
END;
$$ LANGUAGE plpgsql;
```

ZUSAMMENHANG MIT DECLARE

DECLARE darf nur vor einem BEGIN-Block stehen:

```
AS $$
DECLARE
  total NUMERIC;
BEGIN
  -- Logik
END;
$$;
```

❌ Nicht erlaubt:

```
BEGIN
  DECLARE total NUMERIC; -- Fehler
END;
```

PROCEDURE → BEGIN / END IMMER NÖTIG

Prozeduren sind immer plpgsql-artig:

- Mehrere Statements
- Transaktionen (COMMIT, ROLLBACK)

```
CREATE PROCEDURE increase_prices(...)
AS $$
BEGIN
  UPDATE products SET unit_price = unit_price * 1.1;
  COMMIT;
END;
$$ LANGUAGE plpgsql;
```

LAB TIME

AUFGABE N: EXERCISES/EXERCISE-2-FUNKTIONEN-PROZEDUREN

EINFÜHRUNG IN KONTROLLSTRUKTUREN

- PL/pgSQL ermöglicht prozedurale Logik in Funktionen
- Kontrollstrukturen steuern den Programmfluss
- Ermöglichen bedingte Ausführung und Wiederholungen
- Typische Einsatzgebiete:
  - Validierungen
  - Berechnungen
  - Dynamische SQL-Abfragen
  - Iteration über Datensätze
- Wichtige Strukturen:
  - IF / ELSIF / ELSE
  - CASE
  - Schleifen (LOOP, WHILE, FOR)
  - RETURN NEXT / RETURN QUERY

IF / ELSIF / ELSE

- Grundstruktur zur bedingten Ausführung
- Syntax:

```
IF condition THEN
    -- Code
ELSIF other_condition THEN
    -- Code
ELSE
    -- Code
END IF;
```

- Beispiele:
  - Prüfen von Eingabeparametern
  - Unterschiedliche Berechnungswege abhängig von Bedingungen

BEISPIEL: IF / ELSE IN EINER FUNKTION

- Demonstriert einfache Prüfungen innerhalb einer Funktion
- Ergebnis abhängig vom Parameterwert

```
CREATE OR REPLACE FUNCTION check_age(age INT)
RETURNS TEXT AS $$
BEGIN
  IF age < 18 THEN
    RETURN 'Minderjährig';
  ELSE
    RETURN 'Volljährig';
  END IF;
END;
$$ LANGUAGE plpgsql;
```

CASE-AUSDRÜCKE

- Alternative zu mehreren IF-Bedingungen
- Übersichtlicher, wenn viele Vergleiche benötigt werden
- Syntax:

```
CASE
  WHEN condition THEN result
  WHEN condition THEN result
  ELSE default_result
END
```

- Wird häufig verwendet für:
  - SELECT-Abfragen
  - Funktionen
  - Datenberechnungen

BEISPIEL: CASE IN PL/PGSQL

- Gut geeignet für Status- oder Kategorienlogik

```
CASE
  WHEN status = 'open' THEN 'Offen'
  WHEN status = 'closed' THEN 'Geschlossen'
  ELSE 'Unbekannt'
END;
```

SCHLEIFEN: LOOP, WHILE, FOR

LOOP

- Endlose Schleife, bis explizit beendet
- Verwendung von `EXIT WHEN` zum Abbruch

```
LOOP
  -- Code
  EXIT WHEN counter > 5;
END LOOP;
```

WHILE

- Schleife mit Bedingung am Anfang
- Führt Code aus, solange Bedingung wahr ist

```
WHILE counter < 10 LOOP
  counter := counter + 1;
END LOOP;
```

FOR

- Iteriert über einen Bereich oder eine Abfrage
- Automatische Zählvariable

```
FOR i IN 1..5 LOOP
  RAISE NOTICE 'Iteration %', i;
END LOOP;
```

RETURN NEXT VS. RETURN QUERY

RETURN NEXT

- Fügt eine Zeile zum Rückgabestrom hinzu
- Wird in Schleifen verwendet, um mehrere Zeilen zurückzugeben
- Beispiel:

```
FOR record IN SELECT * FROM my_table LOOP
  RETURN NEXT record;
END LOOP;
```

RETURN QUERY

- Fügt mehrere Zeilen per SELECT auf einmal hinzu
- Effizienter für größere Datenmengen
- Beispiel:

```
RETURN QUERY SELECT * FROM my_table WHERE condition;
```

BEISPIEL-FUNKTION: RETURN NEXT / RETURN QUERY

```
CREATE OR REPLACE FUNCTION get_active_customers()
RETURNS SETOF customers AS $$
DECLARE
  rec customers;
BEGIN
  -- mehrere Zeilen
  RETURN QUERY SELECT * FROM customers WHERE active = true;

  -- zusätzliche Einzelausgabe (optional)
  rec.id := 999;
  rec.name := 'Zusatz';
  rec.active := true;
  RETURN NEXT rec;
END;
$$ LANGUAGE plpgsql;
```

BEST PRACTICES BEI KONTROLLSTRUKTUREN

- Logik so einfach wie möglich halten
- Lange IF-Ketten → besser CASE verwenden
- Schleifen vermeiden, wenn SQL-Abfrage möglich ist (performanter!)
- RETURN QUERY bevorzugen statt viele RETURN NEXT
- Immer EXIT-Bedingungen in LOOPs nutzen
- Kommentare zur Erklärung komplexer Logik
- Fehler/Edge-Cases mit IF behandeln

LAB TIME

AUFGABE N: EXERCISES/EXERCISE-3-KONTROLLSTRUKTUREN

SET-BASIERTE VERARBEITUNG IN FUNKTIONEN

WARUM SET-BASIERT STATT ZEILENWEISE?

- Set-basierte Verarbeitung nutzt die Stärken von SQL
- PostgreSQL ist für Mengenoperationen optimiert
- Weniger Kontextwechsel zwischen SQL & PL/pgSQL
- Kürzerer, lesbarer Code
- Bessere Nutzung von Indizes & Query Planner

⚠️ Anti-Pattern:

```
FOR r IN SELECT * FROM orders LOOP
  UPDATE orders SET processed = true WHERE id = r.id;
END LOOP;
```

✅ Besser:

```
UPDATE orders SET processed = true WHERE processed = false;
```

INSERT / UPDATE / DELETE IN FUNKTIONEN

Grundidee

- DML-Statements funktionieren identisch wie im normalen SQL
- Mehrere Zeilen werden in einem Statement verarbeitet

Beispiel: Set-basiertes UPDATE

```
CREATE OR REPLACE FUNCTION mark_old_orders()
RETURNS void AS $$
BEGIN
  UPDATE orders
  SET status = 'archived'
  WHERE order_date < CURRENT_DATE - INTERVAL '2 years';
END;
$$ LANGUAGE plpgsql;
```

- Kein Loop
- Eine Transaktion
- Planbar & performant

INSERT … SELECT IN FUNKTIONEN

Typischer Use-Case

- Daten aus einer Tabelle aggregiert oder gefiltert übernehmen
- Sehr häufig in:
  - Reporting-Funktionen
  - Batch-Verarbeitung
  - Datenmigration

RETURNING IN PL/PGSQL

Warum RETURNING?

- Ergebnisse direkt aus DML-Statements zurückgeben
- Kein separates SELECT notwendig

Beispiel:

```
CREATE OR REPLACE FUNCTION close_order(p_id int)
RETURNS orders AS $$
DECLARE
  v_order orders;
BEGIN
  UPDATE orders
  SET status = 'closed'
  WHERE id = p_id
  RETURNING * INTO v_order;

  RETURN v_order;
END;
$$ LANGUAGE plpgsql;
```

- Ideal für APIs
- Spart Roundtrips
- Garantiert konsistente Daten

RETURNING MIT MEHREREN ZEILEN

- Funktion muss RETURNS TABLE(...) oder SETOF sein
- Sehr mächtig für Batch-Verarbeitung

Beispiel:

```
RETURN QUERY
UPDATE orders
SET status = 'processed'
WHERE status = 'new'
RETURNING id, customer_id;
```

- Alle aktualisierten Zeilen werden zurückgegeben
- Perfekt für Status-Updates mit Rückmeldung

MERGE IN FUNKTIONEN

MERGE = Set-basiertes UPSERT

- INSERT / UPDATE / DELETE in einem Statement
- Besonders geeignet für Synchronisationen

Beispiel:

```
MERGE INTO products p -- Ziel
USING product_import i -- Quelle
ON p.sku = i.sku
WHEN MATCHED THEN -- Datensatz existiert
  UPDATE SET price = i.price
WHEN NOT MATCHED THEN -- Datensatz existiert nicht
  INSERT (sku, price)
  VALUES (i.sku, i.price);
```

- Klarer als mehrere IFs
- Transaktional
- Sehr gut optimierbar

MERGE VS. PROZEDURALE LOGIK

❌ **Schleifen + IF:**

- schwer wartbar
- komplizierter Code
- fehleranfällig
- schlechtere Performance

✅ **MERGE:**

- deklarativ
- kompakt
- planner-freundlich

TRANSAKTIONSVERHALTEN IN FUNKTIONEN

Wichtig zu wissen

- Funktionen laufen immer innerhalb einer Transaktion
- COMMIT / ROLLBACK nicht erlaubt
- Fehler → komplette Funktion wird zurückgerollt

Beispiel

```
BEGIN
  UPDATE orders SET status = 'paid';
  -- Fehler hier → UPDATE wird zurückgerollt
END;
```

SAVEPOINTS? NICHT IN FUNKTIONEN

- SAVEPOINT nur in PROCEDURE, nicht in FUNCTION
- Funktionen sind:
  - atomar
  - deterministisch gedacht
  - ideal für set-basierte Logik

BEST PRACTICES

- ✅ SQL denkt in Mengen, nicht in Zeilen
- ✅ Schleifen nur, wenn fachlich nötig
- ✅ RETURNING statt zusätzlicher SELECTs
- ✅ MERGE für Synchronisationen bevorzugen
- ❌ Business-Logik nicht in FOR-Loops nachbauen
- ❌ Funktionen nicht als Mini-Transaktionsmanager missbrauchen

ZUSAMMENFASSUNG

- Set-basierte Verarbeitung ist der Schlüssel zu Performance
- Funktionen sind ideal für:
  - Bulk-Updates
  - Aggregationen
  - saubere Datenlogik
- Weniger Code, bessere Lesbarkeit, weniger Bugs

🧠 Merksatz:
Wenn du eine Schleife schreibst, prüfe zuerst, ob SQL das nicht alleine kann.

LAB TIME

AUFGABE N: EXERCISES/EXERCISE-4-SET-BASIERTE-VERARBEITUNG

DYNAMISCHES SQL

Was ist Dynamisches SQL?

- SQL-Befehle, die zur Laufzeit als Text zusammengesetzt und ausgeführt werden
- Ermöglicht flexible Abfragen (z. B. dynamische Tabellennamen, Spaltenlisten, WHERE-Bedingungen)
- Kommt hauptsächlich in PL/pgSQL-Funktionen zum Einsatz

Typische Anwendungsfälle

- Generische Funktionen (z. B. Tabellenbereinigung, Logging, Audit)
- Dynamische Filterbedingungen
- Tabellen/Spaltennamen zur Laufzeit bestimmen
- Migrationen / Automatisierungen

EXECUTE (GRUNDLAGEN)

Syntax:

```
EXECUTE query_string;
```

Hinweis:

- query_string muss ein Text (text) sein
- PostgreSQL interpretiert den String zur Laufzeit
- Ergebnisaufnahme mit INTO möglich

Beispiel:

```
DECLARE
  sql text;
  result int;
BEGIN
  sql := 'SELECT count(*) FROM kunden';
  EXECUTE sql INTO result;
  RAISE NOTICE 'Anzahl Kunden: %', result;
END;
```

EXECUTE … USING (SICHERER PARAMETER-ERSATZ)

Warum USING?

- Platzhalter $1, $2, … statt direkter Werte
- PostgreSQL setzt Werte sicher ein → Schutz vor SQL Injection
- Wichtig für dynamische WHERE-Bedingungen!

Syntax:

```
EXECUTE 'SELECT * FROM kunden WHERE id = $1'
USING kunden_id;
```

Beispiel:

```
DECLARE
  sql text := 'SELECT name FROM kunden WHERE id = $1';
  name text;
BEGIN
  EXECUTE sql INTO name USING 42;
END;
```

Vorteil: Werte müssen NICHT manuell gequotet werden.

GENERISCHE TABELLEN- UND SPALTENNAMEN

Problem:

Parameter in USING funktionieren nur für Werte – nicht für Tabellennamen oder Spaltennamen.

Daher:

- Namen müssen manuell in den SQL-String eingesetzt werden
- Dafür zwingend quote_ident() oder format() verwenden!

GENERISCHE TABELLEN- UND SPALTENNAMEN

Problem:

Parameter in USING funktionieren nur für Werte – nicht für Tabellennamen oder Spaltennamen.

Daher:

- Namen müssen manuell in den SQL-String eingesetzt werden
- Dafür zwingend quote_ident() oder format() verwenden!

QUOTE_IDENT() – SICHERES QUOTING VON IDENTIFIERS

- Syntax:

```
EXECUTE 'SELECT * FROM ' || quote_ident(tabellenname);
```

- Beispiel:

```
DECLARE
  t text := 'kunden';
  cnt int;
BEGIN
  EXECUTE 'SELECT count(*) FROM ' || quote_ident(t)
  INTO cnt;
END;
```

GENERISCHE TABELLEN- UND SPALTENNAMEN

Problem:

Parameter in USING funktionieren nur für Werte – nicht für Tabellennamen oder Spaltennamen.

Daher:

- Namen müssen manuell in den SQL-String eingesetzt werden
- Dafür zwingend quote_ident() oder format() verwenden!

QUOTE_IDENT() – SICHERES QUOTING VON IDENTIFIERS

- Syntax:

```
EXECUTE 'SELECT * FROM ' || quote_ident(tabellenname);
```

- Beispiel:

```
DECLARE
  t text := 'kunden';
  cnt int;
BEGIN
  EXECUTE 'SELECT count(*) FROM ' || quote_ident(t)
  INTO cnt;
END;
```

FORMAT() – LESBARER & SICHERER SQL-AUFBAU

- Platzhalter:
  - %I → Identifier (Tabellen-/Spaltennamen)
  - %L → Literal (Werte)
  - %s → roher String ohne Quoting
- Beispiel:

```
DECLARE
  t text := 'kunden';
  c text := 'name';
  v text;
BEGIN
  EXECUTE format('SELECT %I FROM %I WHERE id = $1', c, t)
  INTO v USING 1;
END;
```

SQL INJECTION VERMEIDEN – BEST PRACTICES

❌ Gefährlich: Ungequotete String-Konkatenation

```
EXECUTE 'SELECT * FROM kunden WHERE name = ''' || input || '''';
```

Problem:

input = '''; DROP TABLE kunden; -- → schwere Schäden möglich.

SQL INJECTION VERMEIDEN – BEST PRACTICES

❌ Gefährlich: Ungequotete String-Konkatenation

```
EXECUTE 'SELECT * FROM kunden WHERE name = ''' || input || '''';
```

Problem:

input = '''; DROP TABLE kunden; -- → schwere Schäden möglich.

✅ Sicher: USING + format/quote_ident

Werte → USING

Tabellen-/Spaltennamen → format('%I')

```
DECLARE
  t text := 'kunden';
  sql text := format('SELECT * FROM %I WHERE id = $1', t);
  rec record;
BEGIN
  EXECUTE sql INTO rec USING 5;
END;
```

ZUSAMMENFASSUNG

- EXECUTE führt SQL-Strings zur Laufzeit aus
- EXECUTE … USING schützt vor SQL Injection bei Werten
- Tabellen-/Spaltennamen müssen gequotet werden!
  - Nutze quote_ident() oder format('%I')
- Werte NIEMALS direkt in Strings einbauen → IMMER USING
- Dynamisches SQL ist mächtig, aber erfordert sehr sorgfältigen Umgang

LAB TIME

AUFGABE N: EXERCISES/EXERCISE-5-DYNAMISCHES-SQL

ARBEITEN MIT CURSORN

- Cursor ermöglichen zeilenweises Verarbeiten von Abfrageergebnissen
- Besonders nützlich in PL/pgSQL-Funktionen & Prozeduren
- Typische Einsatzfälle:
  - Komplexe Geschäftslogik pro Datensatz
  - Verarbeitung großer Ergebnismengen
  - Rückgabe mehrerer Resultsets (REFCURSOR)
- Alternative Ansätze:
  - Set-basierte SQL-Abfragen
  - Window Functions (oft performanter)

CURSOR: STRUKTUR & LEBENSZYKLUS

Typischer Ablauf:

1. Cursor deklarieren
2. Cursor öffnen
3. Daten lesen (FETCH)
4. Cursor schließen

Beispiel:

```
DECLARE cur_users CURSOR FOR
SELECT id, name FROM users;

OPEN cur_users;
FETCH cur_users INTO v_id, v_name;
CLOSE cur_users;
```

- Cursor existieren nur innerhalb der Session / Funktion
- Ressourcenbindung → immer schließen!

CURSOR ÖFFNEN, LESEN, SCHLIESSEN

Öffnen

```
OPEN cur_users;
```

Lesen (FETCH)

```
FETCH cur_users INTO v_id, v_name;
```

Schließen

```
CLOSE cur_users;
```

- FETCH-Varianten:
  - NEXT / PRIOR
  - FIRST / LAST
  - ABSOLUTE / RELATIVE
- FETCH gibt keine Fehlermeldung, wenn kein Datensatz mehr vorhanden ist
- Typischer Fehler: FETCH ohne OPEN

CURSOR IN SCHLEIFEN (EINSATZ IN SCHLEIFEN)

Klassisches LOOP-Pattern

```
OPEN cur_users;

LOOP
  FETCH cur_users INTO v_id, v_name;
  EXIT WHEN NOT FOUND;

  -- Geschäftslogik
END LOOP;

CLOSE cur_users;
```

- NOT FOUND signalisiert: keine weitere Zeile
- Sehr gut geeignet für:
  - Validierungen
  - Protokollierungen
  - Datenmigrationen

FOR-SCHLEIFE MIT IMPLIZITEM CURSOR

- PostgreSQL erstellt intern einen Cursor
- Vorteile:
  - Weniger Boilerplate-Code
  - Automatisches Öffnen/Schließen
  - Weniger Fehleranfällig
- Best Practice, wenn kein expliziter Cursor benötigt wird

Vereinfachte Syntax:

```
FOR rec IN
  SELECT id, name FROM users
LOOP
  -- Verarbeitung pro Zeile
END LOOP;
```

SCROLL VS. NO SCROLL (BLÄTTERN)

NO SCROLL (Standard)

- Nur vorwärts lesen
- Bessere Performance
- Weniger Speicherbedarf

```
DECLARE cur_users NO SCROLL CURSOR FOR
SELECT * FROM users;
```

SCROLL

- Vorwärts und rückwärts lesen möglich
- Ermöglicht FETCH PRIOR, FIRST, LAST, ABSOLUTE
- Etwas höhere Ressourcenanforderung
- Nur einsetzen, wenn wirklich nötig

```
DECLARE cur_users SCROLL CURSOR FOR
SELECT * FROM users;
```

RÜCKGABE VON REFCURSOR (RÜCKGABEN)

Funktionsbeispiel:

```
CREATE FUNCTION get_users()
RETURNS REFCURSOR AS $$
DECLARE
  ref REFCURSOR;
BEGIN
  OPEN ref FOR SELECT * FROM users;
  RETURN ref;
END;
$$ LANGUAGE plpgsql;
```

Verwendung:

```
BEGIN;
SELECT get_users();
FETCH ALL FROM <cursorname>;
COMMIT;</cursorname>
```

- Mehrere Resultsets möglich
- Häufig genutzt bei:
  - Reporting
  - Legacy-Schnittstellen
  - Stored-Procedure-APIs

CURSOR VS. WINDOW FUNCTIONS

Cursor – sinnvoll wenn:

- Komplexe Logik pro Datensatz
- Mehrere Schritte je Zeile notwendig
- Abhängigkeiten zwischen Datensätzen

Window Functions – besser wenn:

- Aggregationen / Rangfolgen
- Performance kritisch
- Rein deklarative SQL-Logik möglich

BEST PRACTICES & TYPISCHE FEHLER

Best Practices

- Cursor immer schließen
- NO SCROLL bevorzugen
- FOR-Schleife statt explizitem Cursor nutzen, wenn möglich
- Kleine Resultsets bevorzugen

Typische Fehler

- Cursor vergessen zu schließen
- Unnötige Cursor statt einfacher SQL-Abfragen
- SCROLL ohne echten Nutzen

ZUSAMMENFASSUNG

- Cursor ermöglichen kontrollierte, zeilenweise Verarbeitung
- Einsatz in Schleifen ist der häufigste Anwendungsfall
- REFCURSOR erlaubt flexible Rückgaben
- Window Functions sind oft die bessere Alternative
- Cursor = Werkzeug für Spezialfälle, nicht für Standardabfragen

LAB TIME

AUFGABE N: EXERCISES/EXERCISE-6-ARBEITEN-MIT-CURSORN

TRIGGER & TRIGGERFUNKTIONEN

- Trigger = automatische Aktionen in der DB, ausgelöst durch Ereignisse
- Ereignisse: `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`
- Können vor oder nach einer Operation ausgeführt werden
- Nutzen: Validierung, Audit, Kaskadierungen, technische Automatisierung
- Implementierung immer mit einer Triggerfunktion (PL/pgSQL)

BEFORE VS. AFTER TRIGGER

BEFORE

- Wird vor der Operation ausgeführt
- Kann Datensätze verändern (NEW manipulieren)
- Kann Operation abbrechen (RAISE EXCEPTION)
- Typische Nutzung: Validierung, Defaults, Berechtigungslogik

AFTER

- Wird nach der Operation ausgeführt
- Sieht endgültige Werte
- Gut für: Logging, Folgeaktionen, Benachrichtigungen
- Kann Operation nicht mehr verhindern

ROW-LEVEL VS. STATEMENT-LEVEL

Row-Level Trigger

- Wird für jede betroffene Zeile ausgeführt
- Zugriff auf NEW und OLD
- Beispiel: Audit-Log
- Syntax: FOR EACH ROW

Statement-Level Trigger

- Wird einmal pro DML-Anweisung ausgeführt
- Kein Zugriff auf NEW/OLD
- Beispiel: Aggregationen, Cache-Refresh
- Syntax: FOR EACH STATEMENT

NEW UND OLD VERWENDEN

- NEW: Die neue Zeile (INSERT/UPDATE)
- OLD: Die ursprüngliche Zeile (UPDATE/DELETE)
- Beispiele:
  - INSERT: Nur NEW verfügbar
  - UPDATE: Beide verfügbar
  - DELETE: Nur OLD verfügbar

Beispiel:

```
IF NEW.amount < 0 THEN
  RAISE EXCEPTION 'Amount darf nicht negativ sein!';
END IF;
```

WICHTIGE TRIGGER-PARAMETER

Parameter	Beschreibung	Verfügbar bei	Typische Nutzung
NEW	Neue Version der Zeile	INSERT, UPDATE (Row-Level)	Validierung, Defaults setzen, Audit
OLD	Alte Version der Zeile	UPDATE, DELETE (Row-Level)	Change-Logs, Vergleiche
TG_OP	Auslösende Operation	Alle Trigger	Unterscheidung von INSERT / UPDATE / DELETE
TG_WHEN	Zeitpunkt der Ausführung (BEFORE / AFTER)	Alle Trigger	Logik abhängig vom Ausführungszeitpunkt
TG_LEVEL	Trigger-Ebene (ROW / STATEMENT)	Alle Trigger	Verzweigung je nach Trigger-Typ
TG_TABLE_NAME	Name der Tabelle	Alle Trigger	Generische Triggerfunktionen
TG_TABLE_SCHEMA	Schema der Tabelle	Alle Trigger	Multi-Schema-Setups
TG_NARGS	Anzahl der Trigger-Argumente	Trigger mit Argumenten	Validierung von Parametern
TG_ARGV[]	Trigger-Argumente (Array)	Trigger mit Argumenten	Konfigurierbare Trigger
current_user	Aktueller DB-User	Überall	Audit-Logs, Security
current_database()	Aktuelle Datenbank	Überall	Mandanten- oder System-Logs

Quelle: https://www.postgresql.org/docs/current/functions-info.html

UPDATE OF TRIGGER

- Auslösung nur bei Änderung bestimmter Spalten
- Vermeidet unnötige Triggerläufe

Beispiel:

```
CREATE TRIGGER price_update_trigger
AFTER UPDATE OF price ON product
FOR EACH ROW
EXECUTE FUNCTION log_price_change();
```

CONSTRAINT TRIGGER

- Trigger, die wie Constraints agieren
- Werden AFTER ausgeführt
- Können deferrable sein
  - Ausführung erst am Ende einer Transaktion
- Typische Nutzung:
  - referenzielle Integrität über mehrere Tabellen
  - komplexere Validierungsregeln

Beispiel:

```
CREATE CONSTRAINT TRIGGER check_status
AFTER UPDATE ON orders
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW
EXECUTE FUNCTION validate_order_status();
```

AUDIT-LOG MIT TRIGGERN (PRAXISBEISPIEL)

Audit-Tabelle

```sql
CREATE TABLE customer_audit (
  id SERIAL PRIMARY KEY,
  customer_id INT,
  old_name TEXT,
  new_name TEXT,
  changed_at TIMESTAMP DEFAULT NOW(),
  operation TEXT
);
```

Triggerfunktion

```sql
CREATE OR REPLACE FUNCTION audit_customer_changes()
RETURNS TRIGGER AS $$
BEGIN
  IF TG_OP = 'UPDATE' THEN
    INSERT INTO customer_audit(customer_id, old_name, new_name, operation)
    VALUES (OLD.id, OLD.name, NEW.name, 'UPDATE');
  END IF;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Trigger erstellen

```sql
CREATE TRIGGER customer_audit_trigger
AFTER UPDATE ON customer
FOR EACH ROW
EXECUTE FUNCTION audit_customer_changes();
```

BEST PRACTICES

- Trigger sparsam und zielgerichtet einsetzen
- Business-Logik nicht vollständig in Triggern verstecken
- Immer kommentieren, da Trigger schwer zu finden/debuggen sind
- Trigger sollten idempotent sein
- RAISE NOTICE in Entwicklungsphase nutzen
- Keine langen oder komplexen Operationen im Trigger
- Bei vielen Triggern → Abhängigkeiten dokumentieren

LAB TIME

AUFGABE N: EXERCISES/EXERCISE-7-TRIGGER

FEHLER- & AUSNAHMEBEHANDLUNG

RAISE: MELDUNGEN UND FEHLER AUSGEBEN

PostgreSQL stellt mit RAISE ein zentrales Werkzeug zur Laufzeitkommunikation bereit.

Varianten

- RAISE NOTICE
- RAISE WARNING
- RAISE EXCEPTION

Syntax

```
RAISE level 'Nachricht %', variable;
```

Typische Einsatzfälle

- Debugging
- Protokollierung
- Abbruch von Operationen

RAISE NOTICE

Eigenschaften

- Rein informativ
- Wird in Logs und psql-Ausgabe angezeigt
- Hat keinen Einfluss auf den Ablauf der Funktion
- Beendet die Funktion nicht

Beispiel

```
RAISE NOTICE 'Aktueller Wert: %', v_counter;
```

Typische Verwendung

- Debug-Ausgaben
- Nachvollziehbarkeit von Abläufen
- Schulungs- und Testzwecke

RAISE WARNING

Eigenschaften

- Warnung, aber kein Abbruch
- Signalisiert ungewöhnliche Situationen
- Transaktion läuft weiter

Beispiel

```
IF v_stock < 0 THEN
    RAISE WARNING 'Negativer Lagerbestand: %', v_stock;
END IF;
```

Typische Verwendung

- Ungewöhnliche, aber nicht kritische Situationen
- Monitoring von Grenzwerten
- Hinweise für Administratoren

RAISE EXCEPTION

Eigenschaften

- Bricht die aktuelle Transaktion ab
- Rollback bis zum letzten Savepoint
- Entspricht einem „harten Fehler“

Beispiel

```
IF v_price <= 0 THEN
    RAISE EXCEPTION 'Preis muss größer als 0 sein';
END IF;
```

Typische Verwendung

- Validierungsfehler
- Unzulässige Zustände
- Abbruch von Operationen bei kritischen Fehlern

EIGENE FEHLERMELDUNGEN & SQLSTATE

PostgreSQL erlaubt das Setzen eigener Fehlercodes.

Syntax:

```
RAISE EXCEPTION
    USING MESSAGE = 'Ungültiger Status',
          ERRCODE = 'P0001';
```

Vorteile:

- Einheitliche Fehlercodes
- Bessere Auswertung in Anwendungen
- Klare Trennung von technischen & fachlichen Fehlern

EXCEPTION HANDLING MIT WHEN

Fehler können gezielt abgefangen werden.

Syntax:

```
BEGIN
    -- Logik
EXCEPTION
    WHEN exception_name THEN
        -- Fehlerbehandlung
END;
```

Ziel:

- Fehler kontrollieren
- Alternative Logik ausführen
- Abbruch vermeiden (wenn sinnvoll)

BEISPIEL: EXCEPTION HANDLING

```
BEGIN
    INSERT INTO orders(id) VALUES (1);
EXCEPTION
    WHEN unique_violation THEN
        RAISE NOTICE 'Bestellung existiert bereits';
END;
```

Ergebnis

- Fehler wird abgefangen
- Funktion läuft weiter
- Kein kompletter Transaktionsabbruch

HÄUFIGE WHEN-KLAUSELN

Typische Exceptions

- unique_violation
- foreign_key_violation
- not_null_violation
- division_by_zero
- others (Catch-All)

Beispiel:

```
EXCEPTION
    WHEN division_by_zero THEN
        RAISE EXCEPTION 'Division durch 0 nicht erlaubt';
    WHEN others THEN
        RAISE EXCEPTION 'Unbekannter Fehler';
```

⚠️ WHEN others sparsam einsetzen!

SAVEPOINTS: TEIL-ROLLBACKS

SAVEPOINTS ermöglichen punktuelles Rollback innerhalb einer Transaktion.

Grundidee

- Fehler betrifft nur einen Teil
- Rest der Logik soll weiterlaufen

Syntax:

```
SAVEPOINT sp1;
-- Operation
ROLLBACK TO SAVEPOINT sp1;
```

SAVEPOINTS IN FUNKTIONEN

In PL/pgSQL werden SAVEPOINTS implizit über BEGIN … EXCEPTION erzeugt.

Beispiel:

```
BEGIN
    INSERT INTO payments VALUES (...);
EXCEPTION
    WHEN foreign_key_violation THEN
        RAISE NOTICE 'Zahlung übersprungen';
END;
```

➡️ PostgreSQL setzt intern einen Savepoint vor dem BEGIN.

KOMBINATION: SAVEPOINT + EXCEPTION HANDLING

Typische Anwendungsfälle

- Batch-Verarbeitung
- Schleifen mit vielen Inserts
- Fehlerhafte Datensätze überspringen

Beispiel:

```
FOR rec IN SELECT * FROM staging_orders LOOP
    BEGIN
        INSERT INTO orders VALUES (rec.*);
    EXCEPTION
        WHEN others THEN
            RAISE WARNING 'Fehler bei Order %', rec.id;
    END;
END LOOP;
```

BEST PRACTICES

✅ Empfohlen

- Fachliche Fehler mit RAISE EXCEPTION melden
- Technische Fehler gezielt abfangen
- Aussagekräftige Fehlermeldungen verwenden
- SQLSTATE für APIs nutzen

⚠️ Vermeiden

- WHEN others ohne Logging
- Exceptions als Steuerungslogik
- Übermäßige RAISE NOTICE im Produktivbetrieb

ZUSAMMENFASSUNG

- RAISE steuert Kommunikation & Fehlerverhalten
- NOTICE & WARNING sind nicht destruktiv
- EXCEPTION beendet Transaktionen kontrolliert
- WHEN ermöglicht sauberes Exception Handling
- SAVEPOINTS schützen Teiloperationen

Merksatz

Gute Fehlerbehandlung macht Funktionen stabil, verständlich und wartbar.

TRANSAKTIONEN: GRUNDLAGEN

- Eine Transaktion = logische Einheit mehrerer SQL-Befehle
- Eigenschaften: ACID
  - Atomicity – alles oder nichts
  - Consistency – Daten bleiben in validem Zustand
  - Isolation – parallele Transaktionen beeinflussen sich nicht
  - Durability – Änderungen bleiben gespeichert
- Typische Anwendungsfälle:
  - Finanztransaktionen
  - Bestellungen
  - Mehrere Updates, die logisch zusammengehören

BEGIN / COMMIT / ROLLBACK

- BEGIN - Startet eine neue Transaktion
- COMMIT - Bestätigt alle Änderungen in der aktuellen Transaktion
- ROLLBACK - Macht alle Änderungen in der aktuellen Transaktion rückgängig
- Beispiel:

```
BEGIN;

UPDATE konto SET saldo = saldo - 100 WHERE id = 1;
UPDATE konto SET saldo = saldo + 100 WHERE id = 2;

COMMIT;
```

- Achtung: Ohne explizites BEGIN arbeitet PostgreSQL im Autocommit-Modus

SAVEPOINTS

- Teiltransaktionen innerhalb einer Transaktion
- Ermöglichen partielles Zurückrollen:
- Beispiel:

```
SAVEPOINT sp1;

UPDATE produkte SET menge = menge - 10 WHERE id = 5;

ROLLBACK TO sp1;  -- nur bis sp1 zurück
```

- Typische Anwendungsfälle:
  - Komplexe Transaktionen mit mehreren Schritten
  - Fehlerbehandlung in Teilprozessen
  - Große Skripte, bei denen nur einzelne Teile rückgängig gemacht werden sollen
  - Iterative Prozesse

LAB TIME

AUFGABE N: EXERCISES/EXERCISE-N-TRANSAKTIONEN

INDEXING DEEP DIVE

ÜBERBLICK: INDEX-TYPEN IN POSTGRESQL

- B-Tree (Standard)
- GIN
- GiST
- BRIN
- Spezialfälle: Partial & Expression Indexe
- Wartung & interne Optimierungen (HOT, REINDEX)

B-TREE INDEX

Eigenschaften

- Sortierte Baumstruktur
- Sehr effizient für:
  - =
  - <, <=, >, >=
  - BETWEEN
  - ORDER BY
- Unterstützt UNIQUE

Typische Use-Cases

- Primärschlüssel
- Fremdschlüssel
- Datums- & ID-Spalten

```
CREATE INDEX idx_orders_order_date
ON orders(order_date);
```

EXPLAIN & EXPLAIN ANALYZE