

# **POSTGRESQL - EINFÜHRUNG UND SQL GRUNDLAGEN**

## 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](mailto:mail@softwaredesign-solution.de)
  - [github.com/softwaredesign-solution](https://github.com/softwaredesign-solution)

## ZEITPLAN

- 9 - 16 Uhr
- 1 Stunde Mittagspause

**04.12.25**

- Grundlagen und Prinzip von PostgreSQL
- Benutzer & Berechtigungen
- Installation im Überblick
- Datenbanken in PostgreSQL
- Schemas in PostgreSQL
- Arbeiten mit Tabellen: Tabellen erzeugen und löschen
- Views in PostgreSQL
- Daten eingeben, ändern, abfragen

**05.12.25**

- Arbeiten mit mehreren Tabellen, Tabellen verbinden
- Suchen, Sortieren, Gruppieren, Berechnen
- Umgang mit Datentypen
- Backup & Restore
- Erweiterte Auswertungen
- Komplexe Abfragen im Überblick

## ABLAUF DES KURSES

Theory & Live Coding



Exercices



Discuss solution

## 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/2025-12-04-postgresql-workshop>
- Im Repository enthalten:
  - 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
```

## **GRUNDLAGEN UND PRINZIP VON POSTGRESQL**

## PRINZIP VON POSTGRESQL

- Open-Source, objektrelationales Datenbankmanagementsystem (ORDBMS)
- Seit 1986 entwickelt (Postgres-Projekt an der University of California, Berkeley)
- Fokus auf Standards, Erweiterbarkeit und Stabilität
- Unterstützt relationale und semistrukturierte Daten (JSON, Arrays, XML)
- Sehr starke SQL-Konformität (ACID, MVCC, Transaktionen)
- Große Community + viele Erweiterungen
- Weit verbreitet in Unternehmen und Open-Source-Projekten
- Plattformunabhängig: Linux, Windows, macOS

## GRUNDPRINZIPIEN VON POSTGRESQL

- **Objektrelationales Modell**
  - Tabellen können komplexe Datentypen enthalten
  - Eigene Datentypen, Funktionen, Operatoren möglich
- **Erweiterbarkeit als Kernprinzip**
  - Erweiterungen installierbar (z. B. PostGIS, pgcrypto)
  - Benutzerdefinierte Funktionen (PL/pgSQL, Python, SQL u. a.)
- **MVCC (Multi-Version Concurrency Control)**
  - Verhindert Locks beim Lesen
  - Snapshot-basierte Isolation → sehr hohe Parallelität
- **Transaktionssicherheit (ACID)**
  - Garantiert zuverlässige Datenverarbeitung
- **Strikte Einhaltung von SQL-Standards**
- **Server-/Client-Architektur**
  - Clients kommunizieren über Protokoll mit PostgreSQL-Server
- **WAL (Write-Ahead Logging)**
  - Garantiert Datensicherheit
  - Grundlage für Replikation & Point-in-Time Recovery

## POSTGRESQL ARCHITEKTUR IM ÜBERBLICK

- PostgreSQL-Server (`postgres` Prozess)
- Backend-Prozesse
  - Ein Prozess pro Client-Verbindung
- Speicherverwaltung
  - Shared Buffers
  - Work Memory
  - WAL-Buffer
- Autovacuum Prozess
  - Entfernt „tote“ Zeilen (MVCC-Reste)
  - Optimierung der Datenbank
- Replikationsprozesse
  - Streaming Replication
  - Logical Replication

## DATENTYPEN IN POSTGRESQL

- Umfassende native Typen:
  - Text, Zahlen, Datum/Zeit, Bool
  - Arrays, ENUMs, Ranges (z. B. int4range)
- JSON/JSONB mit Index-Unterstützung
- Geodaten (über PostGIS)
- Custom Types möglich

## UNTERSCHIEDE POSTGRESQL VS. „NORMALES SQL“

- **SQL-Standard-Konformität**
  - PostgreSQL unterstützt SQL-Standards breiter und korrekter
  - MySQL weicht oft vom Standard ab (z. B. bei GROUP BY, Transaktionen)
- **Objektrelational vs. rein relational**
  - PostgreSQL: ORDBMS → Arrays, JSONB, Custom Types, eigene Operatoren
  - MySQL: klassisches RDBMS ohne komplexe Typisierung
- **MVCC ohne Read-Locks**
  - PostgreSQL: Kein Lesen blockiert Schreiben
  - MySQL (InnoDB): MVCC vorhanden, aber weniger konsistent implementiert
- **Indexpower**
  - PostgreSQL: GIN, GIST, BRIN, b-tree, RUM
  - MySQL: hauptsächlich B-Tree, weniger Typen
- **Erweiterbarkeit**
  - PostgreSQL: Erweiterungen (PostGIS, pgRouting, pg\_stat\_statements ...)
  - MySQL: viel eingeschränkter
- **JSON-Unterstützung**
  - PostgreSQL: JSONB (binary), Indexierung, mächtige Operatoren
  - MySQL: JSON erst später integriert, weniger performant & flexibel
- **Stored Procedures & Sprachen**
  - PostgreSQL: Multi-Language (PL/pgSQL, Python, JS, etc.)
  - MySQL: Nur wenige Sprachen
- **Replikation**
  - PostgreSQL: Logische & physische Replikation
  - MySQL: Master-Master möglich, andere Funktionsweise

## WARUM POSTGRESQL VERWENDEN?

- Hohe Standards & Datensicherheit
- Extrem flexibel & erweiterbar
- Sehr leistungsfähig (MVCC, Indexe, JSONB, Parallelisierung)
- Große Community und langlebige Open-Source-Technologie
- Ideal für Enterprise, GIS, Analytics, Web-Apps

## BENUTZER & BERECHTIGUNGEN

- PostgreSQL verwendet Roles als zentrales Sicherheitskonzept
- Standardmäßig gibt es den Superuser „postgres“
- Eine Role kann sein:
  - Benutzer (Login möglich)
  - Gruppe (Sammelrolle zur Rechtevergabe)
- Rollen können Rechte besitzen oder anderen Rollen Rechte erteilen
- Rollen können Mitglied anderer Rollen sein („Role Inheritance“)

## WAS SIND ROLES IN POSTGRESQL?

- Role = generisches Sicherheitsobjekt (User oder Gruppe)
- Rollen können:
  - Datenbanklogin erlauben / verbieten
  - Rechte auf Tabellen, Schemas, Datenbanken besitzen
  - Unterrollen haben (inherit)
- Vereinheitlicht Benutzer & Gruppen → einfacheres Berechtigungssystem
- Beispiele für Rollenarten:
  - Login Role
  - Group Role
  - Anwendungsrolle (z. B. read\_only)

## BENUTZER ANLEGEN (CREATE ROLE)

- Benutzer mit Login-Rechten erstellen:

```
CREATE ROLE alice LOGIN PASSWORD 'secret';
```

- Benutzer ohne Login (z. B. als Gruppe):

```
CREATE ROLE reporting;
```

- Weitere Attribute:

- CREATEDB (Darf Datenbanken erstellen)
- CREATEROLE (Darf Rollen erstellen)
- SUPERUSER (Superuser-Rechte)
- LOGIN (Darf sich anmelden)
- REPLICATION (Darf Replikation durchführen)
- INHERIT / NOINHERIT (Vererbung von Rechten)

- Passwort ändern:

```
ALTER ROLE alice WITH PASSWORD 'newpass';
```

## RECHTE VERGEBEN (GRANT)

- Rechte werden auf Objektebene vergeben:
  - Tabellen: SELECT, INSERT, UPDATE, DELETE
  - Schemas: USAGE, CREATE
  - Datenbanken: CONNECT, CREATE
- Beispiel: Lesezugriff auf Tabelle:

```
GRANT SELECT ON TABLE customer TO alice;
```

- Schreibrechte:

```
GRANT INSERT, UPDATE ON customer TO alice;
```

- Schema-Berechtigung:

```
GRANT USAGE ON SCHEMA public TO reporting;
```

## RECHTE ENTZIEHEN (REVOKE)

- Rechte entfernen:

```
REVOKE SELECT ON customer FROM alice;
```

- Alle Rechte entziehen:

```
REVOKE ALL PRIVILEGES ON customer FROM al
```

- Auch auf Rollenmitgliedschaft:

```
REVOKE reporting FROM alice;
```

# LOGIN-ROLE VS. GROUP-ROLE

## Login-Role (Benutzer)

- Kann sich anmelden
- Hat Passwort
- Beispiel:

```
CREATE ROLE alice LOGIN PASSWORD 'secret'
```

## Group-Role (Gruppe)

- Kein Login
- Dient zur Bündelung von Rechten
- Beispiel:

```
CREATE ROLE reporting;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO reporting;
```

- Benutzer einer Gruppe hinzufügen:

```
GRANT reporting TO alice;
```

## BEST PRACTICES

- Keine Rechte direkt an Benutzer vergeben  
→ immer über Gruppenrollen (z. B. read\_only, editor)
- Logins nur für echte Benutzer, nicht für technische Gruppen
- Standardrechte über:

```
ALTER DEFAULT PRIVILEGES ...
```

- Passwortrichtlinien beachten (z. B. PASSWORD ENCRYPTED)
- Regelmäßig prüfen:

```
\du -- Rollenübersicht in psql
```

## INSTALLATION IM ÜBERBLICK

- PostgreSQL kann auf verschiedenen Plattformen installiert werden
- Häufige Installationswege:
  - Native Installation (Linux/Windows)
  - Paketmanager (apt, homebrew)
  - Docker-Container
- Ziel: Schnell lauffähige Umgebung für Entwicklung & Produktion

## INSTALLATION UNTER WINDOWS

- Installer von der offiziellen Website
- Enthält: PostgreSQL Server, pgAdmin, StackBuilder
- Benutzerdefinierte Installation möglich
- Automatische Einrichtung eines Dienstes
- Passwort für Superuser postgres wird abgefragt
- Standardport: 5432

## INSTALLATION UNTER LINUX

- Installation per Paketmanager
  - Ubuntu/Debian: sudo apt install postgresql postgresql-contrib
  - CentOS/RHEL: sudo yum install postgresql-server postgresql-contrib
- Alternative: Offizielle PostgreSQL-Paketquellen (aktuellere Versionen)
- Initialisierung der Datenbank (falls erforderlich)
  - z.B. sudo postgresql-setup initdb (RHEL)
- Dienste starten/stoppen
  - sudo systemctl enable postgresql

## INSTALLATION VIA DOCKER

- Sehr beliebt für Entwickler & Schulungen
- Beispiel Docker-Befehl:

```
docker run --name psqltraining -e POSTGRES_PASSWORD=mysecretpassword -p 5432:5432 -d
```

- Vorteile:
  - Keine lokale Installation
  - Schnelle Einrichtung
  - Isolierte Umgebung
  - Leichtes Löschen/Neuaufsetzen
  - Einfaches Backup & Restore

## GRUNDKONFIGURATION

- Wichtige Konfigurationsdateien:
  - postgresql.conf – Servereinstellungen (Port, Speicher, Logging)
  - pg\_hba.conf – Zugriffs- und Authentifizierungsregeln
- Standardverzeichnis:
  - Linux: /var/lib/postgresql/<version>/main/
  - Windows: Im Datenordner des Installers
- Typische Anpassungen:
  - Speicherparameter (shared\_buffers, work\_mem)
  - Logging aktivieren
  - Listen Address ändern: listen\_addresses = '\*'

## DIENST STARTEN/STOPPEN

- Unter Linux (systemd):
  - Start: `sudo systemctl start postgresql`
  - Stopp: `sudo systemctl stop postgresql`
  - Status: `sudo systemctl status postgresql`
- Unter macOS (Homebrew):
  - Start: `brew services start postgresql`
  - Stopp: `brew services stop postgresql`
  - Status: `brew services list`
- Windows:
  - Als Windows-Dienst
  - Verwaltung über Dienste-Konsole
- Docker:
  - Start: `docker start psqltraining`
  - Stopp: `docker stop psqltraining`

## ZUGRIFF EINRICHTEN

- Standardbenutzer: postgres
- Zugriff über lokale Unix-Sockets oder TCP
- In pg\_hba.conf definieren:
  - Authentifizierungsmethode (md5, scram-sha-256, trust, peer)
  - Erlaubte IPs/Subnetze
- Beispiel:

```
host all all 0.0.0.0/0 md5
```

- Remote-Zugriff aktivieren:
  - listen\_addresses='\*'
  - Firewallports öffnen (5432)
- Tools für Zugriff:
  - psql, pgAdmin, DBeaver, DataGrip

# EINFÜHRUNG IN PSQL

- Offizielles CLI-Tool zur Verwaltung von PostgreSQL-Datenbanken
- Ermöglicht direkte Ausführung von SQL-Befehlen
- Ideal für Admins, Entwickler & Automatisierung
- Installiert automatisch mit PostgreSQL
- Alternative zu GUI-Tools wie pgAdmin

```
> psql -d mydb -c "SELECT * FROM users"
   id | name    |      email
-----+-----+
   1 | Alice   | alice@example.com
   2 | Bob     | bob@example.com
   3 | Charlie | charlie@example.com
(3 rows)

>
```

## PSQL STARTEN

- Verbindung zu einer Datenbank herstellen:
  - psql -U benutzername
  - psql -U benutzername -d datenbankname
  - psql -U benutzername -d datenbankname -h host -p port
- Direkt am Server einloggen:
  - sudo -u postgres psql
- Ohne Parameter: verbindet zu aktuellem User + DB gleichen Namens

## BASISBEFEHLE IN PSQL (BACKSLASH-KOMMANDOS)

- \l → Alle Datenbanken anzeigen
- \c dbname → Mit Datenbank verbinden
- \dt → Tabellen auflisten
- \d tablename → Tabellenstruktur anzeigen
- \du → Benutzer anzeigen
- \q → Beenden

## SQL AUSFÜHREN

- SQL-Befehle funktionieren wie gewohnt:
  - `SELECT * FROM customers;`
  - `INSERT INTO ...;`
- Mehrzeilige Befehle werden ausgeführt, wenn ein ; am Ende steht
- Auto-Vervollständigung für Tabellen/Spalten mit Tab
- Fehlermeldungen und Ergebnisse werden direkt angezeigt

## NÜTZLICHE EINSTELLUNGEN

- Ausgabe formatieren:
  - \x on → Erweiterte Ausgabe on/off
  - \a on → Align mode on
  - \H → Ausgabe als HTML
- Anzeige der aktuellen Einstellungen:
  - \set

## SKRIPTE AUSFÜHREN

- SQL-Dateien ausführen:
  - `\i /pfad/zur/datei.sql`
- Per Shell:
  - `psql -U benutzername -d datenbankname -f datei.sql`
- Ideal für Migrationen, Backups, Automatisierung

## DATEN EXPORTIEREN & IMPORTIEREN

- Export (COPY TO)
  - Exportiert Daten aus einer Tabelle in eine Datei
  - Beispiel: COPY table TO '/tmp/data.csv' CSV HEADER;
- Import (COPY FROM)
  - Importiert Daten aus einer Datei in eine Tabelle
  - Beispiel: COPY table FROM '/tmp/data.csv' CSV HEADER;
-  Erfordert Serverzugriff – alternativ:
- Clientseitig:
  - Beispiel: \copy table TO 'file.csv' CSV HEADER;
  - Beispiel: \copy table FROM 'file.csv' CSV HEADER;

## PSQL-HISTORY & AUTOCOMPLETE

- Historie durchsuchen mit Pfeiltasten
- Autocomplete:
  - Tab → Tabellen, Spalten, Befehle

## PRAKTISCHE KURZBEFEHLE

- `\df` → Funktionen anzeigen
- `\dn` → Schemas anzeigen
- `\dt *.*` → Tabellen in allen Schemas
- `\dv` → Views anzeigen
- `\watch 2` → Führt letzten Befehl alle 2 Sekunden aus (Monitoring!)
- `\timing` → Zeigt Ausführungszeit für SQL-Abfragen

## PGADMIN - WEBBASIERTES VERWALTUNGSTOOL FÜR POSTGRESQL

- Grafisches Verwaltungstool für PostgreSQL
- Installation meist zusammen mit PostgreSQL
- Ermöglicht Administration ohne Kommandozeile
- Open Source, plattformübergreifend (Windows, macOS, Linux)
- Ideal für Einsteiger und Schulungen
- Browserbasierte GUI mit komfortabler Bedienung

## HAUPTFUNKTIONEN VON PGADMIN

- Datenbank- und Serververwaltung
- Erstellen, Ändern, Löschen von Datenbanken, Tabellen, Schemata
- SQL-Abfragen schreiben und ausführen (Query Tool)
- Datenbankobjekte durchsuchen und verwalten
- Backup und Restore von Datenbanken
- Benutzer- und Rollenverwaltung
- Dashboard mit Monitoring (Sessions, Locks, Aktivitäten)

## AUFBAU DER OBERFLÄCHE

- Browser/Tree View: Navigation durch Server, DBs, Tabellen, Funktionen
- Dashboard: Überblick über Performance & DB-Aktivitäten
- Properties-Tab: Meta-Informationen zum gewählten Objekt
- SQL-Tab: Zeigt SQL-Befehl zu Aktionen (z. B. CREATE TABLE)
- Query Tool: Editor für SQL-Abfragen
- Data Output: Ergebnisse von SELECTs & Ausgaben

## ARBEITEN MIT DEM QUERY TOOL

- Öffnen über: Rechtsklick auf DB → Query Tool
- Syntax-Highlighting & Autocomplete
- Mehrere Query-Tabs parallel
- Ergebnisse als Tabelle im „Data Output“
- Abfragehistorie einsehbar
- Export von Ergebnissen (CSV, JSON, Excel)

## DATENBANKOBJEKTE VERWALTEN

- Tabellen erstellen / ändern per UI oder SQL
- Spalten, Indizes, Constraints visuell hinzufügen
- Beziehungen grafisch einsehbar
- Trigger & Funktionen verwalten
- Schemata anlegen / umbenennen / löschen

## BENUTZER- UND ROLLENVERWALTUNG

- Rollen erstellen und Berechtigungen setzen
- Passwörter ändern
- Rechte für Datenbanken, Tabellen & Funktionen einstellen
- Verwaltung auch über SQL möglich – UI zeigt SQL an

## BACKUP & RESTORE

- Backup
  - Komplett oder selektiv (Schema, Daten, Rollen)
  - Formate: custom, tar, directory
- Restore
  - Über Menü Restore oder pg\_restore im Hintergrund
  - Nützlich für Migrationen & Sicherungen
- Hinweis: Backup/Restore erfordert passende Rollenrechte

## VORTEILE & NACHTEILE VON PGADMIN

### Vorteile

- Kostenlos und Open Source
- Sehr einsteigerfreundlich
- Umfangreiche Verwaltungsoberfläche
- Gute Visualisierung komplexer DB-Strukturen
- Perfekt für Schulungen, Debugging & schnelle Tests

### Nachteile

- Teilweise träge bei großen Datenbanken
- Viele Funktionen tief verschachtelt
- Power-User bevorzugen oft psql für Automatisierung

## TIPPS FÜR DIE PRAXIS

- „SQL“-Tab nutzen, um zu sehen, welcher SQL-Code hinter UI-Operationen steckt
- Query Tool für alle Übungen verwenden
- Bei Fehlermeldungen: Messages-Tab lesen
- Ergebnisse aus „Data Output“ kopierfähig → gut für Dokumentation
- Für Performance-Analysen: Dashboard → „Activity“ & „Locks“

## INSTALLATION PGADMIN

- pgAdmin ist als:
  - Desktop-App
  - Web-Version (Server-Version)
  - Docker-Container
- Verbindung zu PostgreSQL erfolgt per Host + Port + Benutzer + Passwort

## INSTALLATION UNTER WINDOWS / MACOS

- Offizielle Website: <https://www.pgadmin.org/download/>
- Download des Installers für das jeweilige Betriebssystem
- Typische Installationsschritte:
  - Setup ausführen
  - Speicherort auswählen
  - Web-Browser starten (pgAdmin läuft im Browser)
  - Erstes Passwort für „pgAdmin Master Password“ vergeben
- Nach Start:
  - Neuen Server hinzufügen (Rechtsklick auf „Servers“ → „Create“ → „Server...“)
  - Verbindungsdaten eingeben (Host, Port, Benutzer, Passwort)

## INSTALLATION UNTER LINUX

- Paketmanager verwenden
- Beispiele:
  - Debian/Ubuntu:

```
sudo apt install pgadmin4
```

- Fedora/RHEL:

```
sudo dnf install pgadmin4
```

- Alternativ über Repository der pgAdmin-Projektseite
- Start:
  - Browser: <http://localhost:5050/>
  - Anmeldung mit Master Password

## VERBINDUNG ZU POSTGRESQL HERSTELLEN

- In pgAdmin: Register → Server
- Wichtige Felder:
  - Name: Beliebiger Name für die Verbindung
  - Connection Tab:
    - Host: IP oder Hostname des PostgreSQL-Servers
    - Port: Standard 5432
    - Username: z. B. postgres
    - Password: Passwort des Servers
- Bei Docker-Umgebungen:
  - Host meist nicht localhost, sondern Container-IP oder Docker-Netzwerkname

## HINWEIS: IP-ADRESSE DES POSTGRESQL-CONTAINERS ERMITTeln

- Falls der PostgreSQL-Container nicht per Docker-Netzwerk-Namen erreichbar ist
- In Powershell ausführen:

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

### Ergebnis:

- Gibt die interne Docker-IP des Containers zurück
- Beispiel: 172.18.0.4

Diese IP muss in pgAdmin als Host eingetragen werden

## ALTERNATIVE: VERBINDUNG PER DOCKER-NETZWERKNAME

- Wenn pgAdmin und PostgreSQL im gleichen Docker-Netzwerk laufen:
  - Als Host kann der Service-Name verwendet werden
  - Beispiel:
    - PostgreSQL-Service heißt: psqltraining
    - Host in pgAdmin: psqltraining
- Vorteil:
  - Keine IP-Abfrage notwendig
  - Container können sich automatisch erreichen

## TYPISCHE PROBLEME BEI DER VERBINDUNG

- Fehler: Connection refused
  - PostgreSQL-Server läuft nicht
  - Firewall blockiert Port 5432
- Fehler: could not translate host name
  - Hostname falsch (Docker-Name nur im gleichen Netzwerk gültig)
- Fehler: password authentication failed
  - Passwort aus Umgebungsvariablen überprüfen
- Lösung oft:
  - PostgreSQL-Logs prüfen: `docker logs psqltraining`
  - pg\_hba.conf Einstellungen checken
  - IP-Adresse erneut über `docker inspect` prüfen
  - Container-Neustart

## **LAB TIME**

**AUFGABE 1: EXERCISES/EXERCISE-1-INSTALLATION**

# EINFÜHRUNG IN DATENBANKEN

- Was ist eine Datenbank?
  - Strukturierte Sammlung von Daten
  - Organisiert zur effizienten Speicherung, Abfrage & Verwaltung
  - Grundlage fast aller modernen Anwendungen
- Warum Datenbanken nutzen?
  - Persistente Datenspeicherung
  - Hohe Datenintegrität
  - Paralleler Zugriff mehrerer Benutzer
  - Sicherheit & Zugriffskontrolle
  - Leistungsfähige Such- und Analysefunktionen
- Relationale Datenbanken (RDBMS)
  - Arbeiten mit Tabellen (Zeilen & Spalten)
  - Nutzen SQL als Abfragesprache
  - Bieten starke Konsistenz (ACID-Prinzip)
- PostgreSQL als Beispiel
  - Objekt-relational
  - Open Source
  - Sehr mächtig (JSONB, Window Functions, CTEs, Erweiterungen, ...)

## WIE ERSTELLE ICH EINE DATENBANK?

- Datenbank anlegen (SQL-Befehl):

```
CREATE DATABASE dbname;
```

- Datenbank löschen:

```
DROP DATABASE dbname;
```

- In psql verbinden
  - \l - Liste aller Datenbanken anzeigen
  - \c dbname - Verbindung zu einer Datenbank herstellen
- In pgAdmin
  - Rechtsklick auf „Databases“ → „Create“ → „Database...“
  - Datenbankname eingeben & speichern
- Praxis-Hinweis
  - Projekt-/Teamdatenbanken systematisch benennen (z. B. „crm\_dev“, „crm\_test“, „crm\_prod“)
  - Rechtevergabe gleich mit einplanen

## WAS BEINHALTET EINE DATENBANK?

- Schemas
  - Strukturierungseinheiten (Ordner/Namespaces)
  - Enthalten Tabellen, Views, Funktionen etc.
- Tabellen
  - Hauptspeicherort für Daten
  - Definiert durch Spalten & Datentypen
- Views (Sichten)
  - Virtuelle Tabellen basierend auf SELECT-Abfragen
- Sequences
  - Nummerngeber (z. B. für Auto-Increment)
- Funktionen & Prozeduren
  - Logik auf Datenbankebene (z. B. Berechnungen, Validierungen)
- Indices
  - Beschleunigen Abfragen
- Constraints
  - Regeln für Datenintegrität (PRIMARY KEY, FOREIGN KEY, UNIQUE...)
- Extensions (Erweiterungen)
  - Zusätzliche Features (z. B. uuid-ossp, postgis)

## SCHEMAS IN POSTGRESQL

- Ein Schema ist ein Namensraum in einer Datenbank
- Dient zur logischen Strukturierung von Tabellen, Views, Funktionen etc.
- Vergleichbar mit Ordnern in einem Filesystem
- Mehrere Objekte können denselben Namen haben – solange sie in verschiedenen Schemas liegen
- Schemas erleichtern Organisation, Ordnung und Sicherheit

## AUFBAU: DATABASE → SCHEMAS → TABLES

- Hierarchie in PostgreSQL:
  - Datenbank
    - enthält Schemas
      - enthalten Tabellen, Views, Indexe, Funktionen ...
- Ein Benutzer arbeitet innerhalb einer Datenbank, aber kann mehrere Schemas nutzen
- Klare Trennung von Strukturen – hilfreich bei großen Projekten

## DAS PUBLIC-SCHEMA

- Jede PostgreSQL-Datenbank hat standardmäßig ein Schema: public
- Ohne weitere Konfiguration werden Tabellen automatisch im public-Schema erstellt
- Alle Benutzer haben auf public standardmäßig CREATE-Rechte
- In produktiven Systemen oft empfehlenswert:
  - Rechte im public-Schema einschränken
  - Eigene Schemas pro Anwendung/Modul verwenden

# CREATE SCHEMA

Schema anlegen:

```
CREATE SCHEMA reporting;
```

Schema mit Owner:

```
CREATE SCHEMA sales AUTHORIZATION sales_user;
```

Objekt direkt im Schema erzeugen:

```
CREATE TABLE reporting.monthly_stats (
    id SERIAL PRIMARY KEY,
    total NUMERIC
);
```

## **SET SEARCH\_PATH**

- Der search\_path bestimmt, in welchen Schemas PostgreSQL zuerst nach Objekten sucht
- Standard:

```
"$user", public
```

- Beispiel:

```
SET search_path TO sales, public;
```

- Danach reicht:

```
SELECT * FROM kunden;
```

statt:

```
SELECT * FROM sales.kunden;
```

## WARUM SCHEMAS WICHTIG SIND

- Struktur & Ordnung
  - Module, Fachbereiche, Teams sauber trennen (z. B. sales, analytics, security)
- Namenskonflikte vermeiden
  - Zwei Tabellen dürfen denselben Namen haben → wenn sie in verschiedenen Schemas liegen
- Security & Rechteverwaltung
  - Berechtigungen pro Schema vergeben
  - Benutzer auf einzelne Schemas beschränken
- Multi-Tenant Architekturen
  - Jeder Kunde erhält eigenes Schema
- Deployment-Organisation
  - Datenmodelle verschiedener Services sauber trennen

## BEST PRACTICES

- Schemas bewusst als logische Module verwenden
- Pro Anwendung/Modul ein eigenes Schema
- Rechte systematisch vergeben:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
GRANT USAGE ON SCHEMA sales TO sales_user;
```

- search\_path gezielt setzen oder immer schema-qualified arbeiten (schema.table)

## ARBEITEN MIT TABELLEN

- Tabellen bilden die zentrale Struktur zur Speicherung von Daten
- Tabellen bestehen aus Spalten (Attribute) mit definierten Datentypen
- PostgreSQL bietet mächtige Möglichkeiten zur Strukturierung (Constraints, Keys, Datentypen)

## TABELLEN ERSTELLEN: GRUNDLAGEN (CREATE TABLE)

- Syntax:

```
CREATE TABLE tabellenname (
    spaltenname1 datentyp [CONSTRAINTS],
    spaltenname2 datentyp [CONSTRAINTS],
    ...
);
```

- Jede Spalte benötigt:
  - Spaltenname
  - Datentyp (z. B. VARCHAR, INT, BOOLEAN, DATE, UUID)
  - Optionale Constraints (z. B. NOT NULL, PRIMARY KEY)
- Optionale Eigenschaften:
  - DEFAULT-Werte
  - NOT NULL
  - PRIMARY KEY, FOREIGN KEY
  - UNIQUE, CHECK
- Best Practices:
  - Sinnvolle Datentypen wählen
  - Spaltennamen klar und konsistent halten
  - Möglichst früh Constraints definieren

## CREATE TABLE - BEISPIELE

- Einfaches Beispiel:

```
CREATE TABLE customers (
    id SERIAL PRIMARY KEY,
    firstname VARCHAR(50),
    lastname VARCHAR(50),
    email VARCHAR(100) UNIQUE
);
```

- Beispiel mit CHECK und DEFAULT:

```
CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    amount NUMERIC(10,2) CHECK (amount > 0)
    created_at TIMESTAMP DEFAULT NOW()
);
```

## PRIMÄRSCHLÜSSEL (PRIMARY KEY)

- Eindeutige Identifikation eines Datensatzes
- Muss eindeutig und nicht NULL sein
- Varianten:
  - Automatische Generierung mit SERIAL oder GENERATED ALWAYS AS IDENTITY
  - Zusammengesetzte Primary Keys möglich
- Beispiel:

```
id SERIAL PRIMARY KEY
```

- Vorteil: Schnelle Suche & klare Referenzen

## FREMDSCHLÜSSEL (FOREIGN KEY)

- Definiert eine Beziehung zwischen Tabellen
- Stellt referenzielle Integrität sicher
- Syntax:

```
spaltenname datentyp REFERENCES referenztabelle(referenzspalte)
customer_id INT REFERENCES customers(id)
```

- Unterstützt Aktionen:
  - ON DELETE (CASCADE, SET NULL, RESTRICT)
  - ON UPDATE (CASCADE, SET NULL, RESTRICT)
- Best Practices:
  - Fremdschlüssel immer definieren
  - Nur gültige Werte in relationalen Feldern zulassen

## WEITERE CONSTRAINTS

- UNIQUE – Wert darf in der Spalte nicht mehrfach vorkommen
- CHECK – Benutzerdefinierte Bedingungen
- NOT NULL – Wert muss vorhanden sein
- DEFAULT – Standardwert, wenn nichts angegeben wird
- Beispiel:

```
price NUMERIC CHECK (price >= 0)
```

## TABELLEN ÄNDERN (ALTER TABLE)

- Spalten hinzufügen:

```
ALTER TABLE customers ADD COLUMN phone VARCHAR(30);
```

- Spalten ändern:

```
ALTER TABLE customers ALTER COLUMN phone TYPE TEXT;
```

- Constraints hinzufügen:

```
ALTER TABLE customers ADD CONSTRAINT chk_email CHECK (email <
```

- Spalten löschen:

```
ALTER TABLE customers DROP COLUMN phone;
```

## TABELLEN LÖSCHEN (DROP TABLE)

- Tabelle komplett löschen:

```
DROP TABLE customers;
```

- Fehler vermeiden, wenn Tabelle nicht existiert:

```
DROP TABLE IF EXISTS customers;
```

- Abhängigkeiten berücksichtigen:

- DROP TABLE ... CASCADE (löscht abhängige Objekte)
- Vorsicht: kann Kaskadenlöschungen auslösen!

## BEST PRACTICES IM UMGANG MIT TABELLEN

- Primärschlüssel immer definieren
- Konsistente Namenskonventionen (snake\_case)
- Constraints möglichst früh einbauen
- Keine unnötigen Spalten („Data Normalization“)
- Struktur nicht zu früh optimieren – aber sauber gestalten
- Backup der Struktur vor Änderungen (pg\_dump --schema-only)

## **LAB TIME**

**AUFGABE 2: EXERCISES/EXERCISE-2-ARBEITEN-MIT-TABELLEN**

## EINFÜHRUNG IN VIEWS

- Ein View ist eine virtuelle Tabelle, basierend auf einer SELECT-Abfrage
- Speichert keine eigenen Daten (Ausnahme: Materialized Views)
- Dient zur Vereinfachung von komplexen Abfragen
- Ermöglicht Datenabstraktion
- Wird wie eine Tabelle abgefragt:

```
SELECT * FROM view_name;
```

## VORTEILE VON VIEWS

- Wiederverwendbarkeit häufig benötigter Abfragen
- Vereinfachung komplexer SQL-Statements
- Sicherheit:
  - Zugriff auf sensible Daten einschränken
  - Nutzer sehen nur relevante Spalten
- Strukturelle Flexibilitätstrong
  - Änderungen an Tabellen müssen nicht alle Anwendungen betreffen
- Lesbarkeit & Wartbarkeit erhöhen

## WIE ERSTELLT MAN EINEN VIEW?

- Syntax:

```
CREATE VIEW view_name AS  
SELECT spalten  
FROM tabellen  
WHERE bedingung;
```

- Beispiel:

```
CREATE VIEW active_customers  
SELECT id, name, email  
FROM customer  
WHERE is_active = true;
```

## VIEWS VERWENDEN

- Wie eine Tabelle abfragbar:

```
SELECT * FROM active_customers;
```

- Views können auch gefiltert werden:

```
SELECT name  
FROM active_customers  
WHERE id > 100;
```

## VIEWS AKTUALISIEREN / ERSETZEN

- View ändern (kompletter Neuaufbau):

```
CREATE OR REPLACE VIEW active_customers AS
SELECT id, name
FROM customer
WHERE is_active = true AND deleted = false;
```

- Bestehende Abhängigkeiten bleiben erhalten

## VIEW LÖSCHEN

- View löschen:

```
DROP VIEW view_name;
```

- Fehler vermeiden, wenn View nicht existiert:

```
DROP VIEW IF EXISTS view_name;
```

## BEST PRACTICES

- Views nutzen für:
  - Sicherheitsabstraktion
  - Vereinheitlichung von komplexen Reports
  - Logische Business-Schichten
- Views nicht zu stark verschachteln → Performance-Risiko
- Views klar und selbsterklärend benennen

## ÜBERBLICK: DATEN EINGEBEN, ÄNDERN, ABFRAGEN

- Grundlegende CRUD-Operationen in PostgreSQL
- Arbeiten mit Datensätzen in Tabellen
- SQL-Befehle:
  - INSERT – neue Daten einfügen
  - UPDATE – bestehende Daten ändern
  - DELETE – Daten löschen
- Wichtig: WHERE-Klausel richtig einsetzen  
(Vermeidung unerwünschter Änderungen/Löschen)

## DATEN EINGEBEN (INSERT)

- Verwendung von INSERT INTO ... VALUES ...
- Spalten explizit angeben (Best Practice)
- Mehrere Datensätze in einem INSERT möglich
- Beispiel:

```
INSERT INTO kunden (name, stadt, email)
VALUES ('Müller', 'Berlin', 'mueller@example.com');
```

- Alternative: INSERT ... RETURNING für Rückgabe neu erzeugter Werte (z. B. Primärschlüssel)

```
INSERT INTO bestellungen (kunde_id, betrag)
VALUES (1, 99.50)
RETURNING id;
```

## DATEN ABFRAGEN (SELECT)

- Grundstruktur

```
SELECT spalten FROM tabelle;
```

- Alle Spalten abfragen:

```
SELECT * FROM kunden;
```

- Ergebnisse filtern mit WHERE:

```
SELECT *
FROM kunden
WHERE stadt = 'Berlin';
```

- Ergebnisse sortieren (ORDER BY):

```
SELECT name, stadt
FROM kunden
ORDER BY name ASC/DESC;
```

- Ergebnisse begrenzen (LIMIT, OFFSET):

```
SELECT *
FROM kunden
LIMIT 10 OFFSET 20;
```

## DATEN ÄNDERN (UPDATE)

- Änderung bestehender Zeilen mit UPDATE
- Unbedingt WHERE-Bedingung setzen, da sonst alle Zeilen geändert werden
- Beispiel:

```
UPDATE kunden
SET stadt = 'Hamburg'
WHERE id = 3;
```

- Mehrere Spalten gleichzeitig ändern möglich
- Verwendung von RETURNING, um geänderte Daten auszugeben

```
UPDATE kunden
SET email = 'neu@example.com'
WHERE id = 5
RETURNING *;
```

## DATEN LÖSCHEN (DELETE)

- Daten entfernen mit DELETE FROM
- Auch hier: WHERE nicht vergessen
- Beispiel:

```
DELETE FROM kunden  
WHERE id = 10;
```

- Löschen ohne WHERE entfernt alle Datensätze der Tabelle
- Tipp: Vorher Daten prüfen

```
SELECT * FROM kunden WHERE id = 10;
```

- RETURNING verfügbar

```
DELETE FROM kunden  
WHERE id = 10  
RETURNING *;
```

## BEST PRACTICES BEIM DATENMANIPULIEREN

- Immer WHERE-Klausel prüfen
- Erst SELECT testen, dann UPDATE/DELETE ausführen
- Transaktionen nutzen (BEGIN, COMMIT, ROLLBACK)
- Eindeutige Primärschlüssel erleichtern Updates/Löschen
- Constraints sorgen für Datenintegrität
- Bei großen Datenmengen: Batch-Inserts verwenden

## WHERE-BEDINGUNGEN (BASICS)

- Gleich / Ungleich
  - = (Gleich)
  - <> (Ungleich)
  - != (Ungleich)
  - Beispiel: WHERE city = 'Berlin'
- Größer / Kleiner
  - < (Kleiner als)
  - > (Größer als)
  - <= (Kleiner gleich)
  - >= (Größer gleich)
  - Beispiel: WHERE price > 50
- BETWEEN (inklusiv)
  - Beispiel: WHERE price BETWEEN 50 AND 100
- IN (Values)
  - Beispiel: WHERE country IN ('DE', 'AT', 'CH')
- IS NULL / IS NOT NULL
  - Beispiel: WHERE phone IS NOT NULL

## **STRING-, MATHE- & DATUMSFUNKTIONEN**

# STRINGFUNKTIONEN

- LOWER(text) – wandelt in Kleinbuchstaben um

```
LOWER('HELLO') → 'hello'
```

- UPPER(text) – wandelt in Großbuchstaben um

```
UPPER('hello') → 'HELLO'
```

- TRIM(text) – entfernt Leerzeichen am Anfang & Ende

```
TRIM(' Test ') → 'Test'
```

- CONCAT(a, b, ...) – verkettet Strings

```
CONCAT('Post', 'greSQL') → 'PostgreSQL'
```

- || (String-Operator) – ebenfalls zum Verketten

```
'Hello' || ' ' || 'World' → 'Hello World'
```

## MATHEMATISCHE FUNKTIONEN

- ABS(x) – gibt den absoluten Wert zurück

```
ABS(-5) → 5
```

- ROUND(x [, n]) – runden auf n Nachkommastellen

```
ROUND(4.5678) → 5
```

- CEIL(x) – runden aufrundend auf Ganzzahl

```
CEIL(4.1) → 5
```

- FLOOR(x) – runden abrundend auf Ganzzahl

```
FLOOR(4.9) → 4
```

## DATUM- & ZEIT-FUNKTIONEN

- NOW() – aktuelles Datum + Uhrzeit (Timestamp)

```
NOW() → 2025-12-03 17:42:10
```

- CURRENT\_DATE – nur Datum

```
CURRENT_DATE → 2025-12-03
```

- AGE(date1, date2) – Zeitdifferenz berechnen

```
AGE('2025-12-03', '2020-01-01') → '5 years 11 mons 2 days'
```

```
AGE(NOW(), '2020-01-01') → '5 years ...'
```

- DATE\_PART(field, source) – extrahiert Anteile

```
DATE_PART('month', CURRENT_DATE) → 12
```

```
DATE_PART('year', CURRENT_DATE) → 2025
```

```
DATE_PART('dow', CURRENT_DATE) → Wochentag (0-6)
```

## **LAB TIME**

**AUFGABE 3: EXERCISES/EXERCISE-3-DATEN-EINGEBEN-AENDERN-ABFRAGEN**

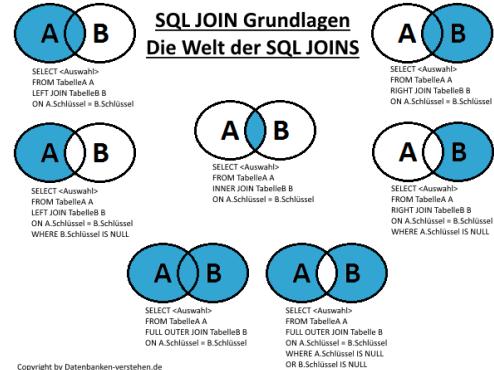
## ARBEITEN MIT MEHREREN TABELLEN – ÜBERBLICK

- Relationale Daten liegen in normalisierten Tabellen
- Verbindung über Schlüssel (Primär- und Fremdschlüssel)
- Nutzung von JOINs, um Daten aus mehreren Tabellen zusammenzuführen
- Typische Anwendungsfälle:
  - Kundendaten + Bestellungen
  - Artikel + Kategorien
  - Mitarbeiter + Abteilungen

## BEZIEHUNGEN ZWISCHEN TABELLEN

- 1:1 Beziehung
  - Selten, z. B. User + UserDetails
  - Beide Tabellen über eindeutigen Schlüssel verbunden
- 1:n Beziehung (häufigste)
  - Beispiel: Kunde → Bestellungen (ein Kunde, viele Bestellungen)
  - Fremdschlüssel in der „n“-Tabelle
- n:m Beziehung
  - Beispiel: Produkte ↔ Kategorien
  - Realisiert durch Zwischentabelle (Mapping-Tabelle)
  - Wichtig: Referentielle Integrität durch FOREIGN KEY Constraints

# JOINS – ÜBERBLICK



## INNER JOIN

- Gibt nur Datensätze aus beiden Tabellen zurück, die zueinander passen
- Wird am häufigsten eingesetzt
- Beispiel:

```
SELECT k.name, b.datum
FROM kunden k
INNER JOIN bestellungen b
ON k.id = b.kunden_id;
```

- Typischer Einsatz: Kunden mit Bestellungen

## LEFT JOIN

- Gibt alle Datensätze aus der linken Tabelle zurück
- Rechte Tabelle nur, wenn passende Einträge vorhanden sind
- Nicht gefundene Werte → NULL
- Beispiel:

```
SELECT k.name, b.datum
FROM kunden k
LEFT JOIN bestellungen b
ON k.id = b.kunden_id;
```

- Einsatzfall: Kunden ohne Bestellungen anzeigen

## RIGHT JOIN

- Gegenteil des LEFT JOIN
- Gibt alle Datensätze der rechten Tabelle zurück
- Wird selten verwendet (LEFT JOIN ist meist klarer)
- Beispiel:

```
SELECT k.name, b.datum
FROM kunden k
RIGHT JOIN bestellungen b
  ON k.id = b.kunden_id;
```

## FULL OUTER JOIN

- Gibt alle Datensätze beider Tabellen zurück
- Zeigt auch unverknüpfte Werte (Links- oder Rechtsseite) mit NULL
- Beispiel:

```
SELECT k.name, b.datum
FROM kunden k
FULL OUTER JOIN bestellungen b
ON k.id = b.kunden_id;
```

- Einsatzfall: Vollständige Datenanalyse mit potenziellen Lücken

## JOIN-FALLBEISPIELE

- Beispiel 1: Produkte mit Kategorien (INNER JOIN)
- Beispiel 2: Alle Produkte inkl. solche ohne Kategorie (LEFT JOIN)
- Beispiel 3: Kunden und Bestellungen analysieren (LEFT / INNER JOIN)
- Beispiel 4: Fehlende Zuordnungen aufdecken (FULL OUTER JOIN)
- Beispiel 5: n:m Beziehung (über Zwischentabelle verbinden)

## BONUS – SELF JOIN

- Tabelle wird mit sich selbst verbunden
- Typische Fälle:
  - Mitarbeiter und Vorgesetzte in derselben Tabelle
  - Kategorien mit übergeordneten Kategorien
  - Hierarchische Strukturen
- Beispiel:

```
SELECT e.name, v.name AS vorgesetzter
FROM mitarbeiter e
LEFT JOIN mitarbeiter v
  ON e.vorgesetzter_id = v.id;
```

## **LAB TIME**

**AUFGABE 4: EXERCISES/EXERCISE-4-ARBEITEN-MIT-MEHREREN-TABELLEN**

**SUCHEN, SORTIEREN, GRUPPIEREN, BERECHNEN**

## ORDER BY — DATEN SORTIEREN

- Sortiert das Ergebnis einer SELECT-Abfrage
- Standard: ASC (aufsteigend)
- Weitere Optionen: DESC, Sortieren nach mehreren Spalten
- Kombination möglich: ORDER BY preis DESC, name ASC
- Sortierung nach berechneten Werten oder Aliasen möglich
- NULL-Handling: NULLS FIRST / NULLS LAST
- Beispiel:

```
SELECT * FROM produkte ORDER BY preis DESC;
```

## LIKE, ILIKE, PATTERN MATCHING – DATEN SUCHEN

### LIKE

- Case-sensitive Mustererkennung
- Platzhalter:
  - % mehrere Zeichen
  - \_ genau ein Zeichen
- Beispiel: WHERE name LIKE 'A%'

### ILIKE

- Case-insensitive Variante von LIKE
- Beispiel: WHERE name ILIKE '%test%'

### Pattern Matching mit regulären Ausdrücken

- PostgreSQL unterstützt Regex-Operatoren
- ~ (case-sensitive), ~\* (case-insensitive)
- Beispiel: WHERE name ~ '^[A-Z].\*'

## GROUP BY — DATEN GRUPPIEREN

- Gruppert Zeilen basierend auf Spaltenwerten
- Wird benötigt, wenn Aggregatfunktionen und normale Spalten kombiniert werden
- Beispiel: SELECT kategorie, COUNT(\*) FROM produkte GROUP BY kategorie
- Alle nicht-aggregierten Spalten müssen im GROUP BY stehen
- Beispiel:

```
SELECT kategorie, AVG(preis)
FROM produkte
GROUP BY kategorie;
```

## HAVING – BEDINGUNGEN AUF GRUPPEN

- Filtert Gruppen nach der Aggregation
- Ergänzt WHERE (Filtern vor der Gruppierung)
- Beispiel:

```
SELECT kategorie, COUNT(*)  
FROM produkte  
GROUP BY kategorie  
HAVING COUNT(*) > 10;
```

## AGGREGATFUNKTIONEN — BERECHNUNGEN DURCHFÜHREN

- COUNT() – Anzahl der Zeilen
- SUM() – Summe von Werten
- AVG() – Durchschnitt
- MIN() / MAX() – Kleinster / größter Wert
- Sonderfall: COUNT(\*) zählt alle Zeilen, COUNT(spalte) ignoriert NULLs
- Aggregatfunktionen können mit DISTINCT verwendet werden
- Beispiel: COUNT(DISTINCT kunde\_id)

## **LAB TIME**

**AUFGABE 5: EXERCISES/EXERCISE-5-SUCHEN-SORTIEREN-GRUPPIEREN-BERECHNEN**

## ÜBERBLICK: DATENTYPEN IN POSTGRESQL

- PostgreSQL bietet eine sehr umfangreiche Typenbibliothek
- Kategorien:
  - Skalare Datentypen (TEXT, INTEGER, BOOLEAN, NUMERIC, DATE ...)
  - Komplexe Datentypen (JSONB, ARRAY, HSTORE ...)
  - Geodaten (PostGIS – Erweiterung)
  - Benutzerdefinierte Datentypen
- PostgreSQL unterstützt starke Typisierung  
→ verhindert viele Laufzeitfehler

## WICHTIGE DATENTYPEN (TEIL 1)

- TEXT / VARCHAR
  - TEXT: unbegrenzte Länge
  - VARCHAR(n): begrenzt auf n Zeichen
  - Beide sind ähnlich performant – TEXT wird meist empfohlen
- INTEGER / BIGINT
  - INTEGER: Ganzzahlen von -2.147.483.648 bis 2.147.483.647
  - BIGINT: Für größere Werte bis 9.223.372.036.854.775.807
- NUMERIC(p,s)
  - Exakte Dezimalwerte → ideal für Geldbeträge
  - Vermeidet Rundungsfehler
- BOOLEAN
  - TRUE / FALSE / NULL

## WICHTIGE DATENTYPEN (TEIL 2)

- DATE / TIME / TIMESTAMP
  - DATE: Nur Datum (YYYY-MM-DD)
  - TIME: Nur Uhrzeit (HH:MM:SS)
  - TIMESTAMP: Datum + Uhrzeit (YYYY-MM-DD HH:MM:SS)
  - TIMESTAMPTZ: Datum + Uhrzeit mit Zeitzone
- SERIAL / BIGSERIAL
  - Auto-Increment (intern Sequenzen)
  - Heute: Empfehlung → GENERATED AS IDENTITY
- UUID
  - Eindeutige IDs ohne Sequenzen
  - Gut für verteilte Systeme
- 
- JSON / JSONB
  - JSON: originalgetreu gespeichert
  - JSONB: binär, schneller für Abfragen & Indizes

## BESONDERHEITEN VON POSTGRESQL-DATENTYPEN

- ARRAY-Typen
  - Speicherung von Listen: INT[], TEXT[]
  - Gute Nutzung für kleinere Mengen, kein Ersatz für Tabellen
- ENUM-Typen
  - Definierte Auswahl fester Werte
  - Gut für Status-Felder
- RANGE-Typen (z. B. int4range, tsrange)
  - Speicherung von Wertebereichen
  - Perfekt für Zeiträume (z. B. Buchungsfenster)
- HSTORE
  - Key/Value als Text – leichtgewichtiges JSON
  - Unterstützung für Geodaten (PostGIS) durch Erweiterungen

## CASTING & TYPKONVERTIERUNG

- Explizites Casting
  - Syntax 1: SELECT '123'::INTEGER;
  - Syntax 2: SELECT CAST('2024-01-01' AS DATE);
- Implizites Casting
  - Automatische Typumwandlung bei kompatiblen Typen
  - Beispiel: Vergleich TEXT mit INTEGER
- Typfunktionen
  - TO\_CHAR(timestamp, 'DD.MM.YYYY')
  - TO\_NUMBER(text, '999.99')
  - TO\_DATE(text, 'YYYY-MM-DD')
  - TO\_TIMESTAMP(text, 'YYYY-MM-DD')
- Wichtig: Immer auf korrekte Formate achten  
→ sonst Fehler wie invalid input syntax
- Explizites Casting mit :: oder CAST()
- Beispiel: SELECT '123'::INTEGER; oder SELECT CAST('123' AS INTEGER);
- Automatisches Casting bei kompatiblen Typen
- Vorsicht bei nicht kompatiblen Typen (Fehler)
- Funktionen zur Typumwandlung (z. B. TO\_CHAR, TO\_DATE)

## BEST PRACTICES IM UMGANG MIT DATENTYPEN

- Passende Datentypen wählen:
  - Geldwerte → NUMERIC
  - Ganzzahlen → INTEGER/BIGINT
  - Wahrheitswerte → BOOLEAN
  - Timestamp → TIMESTAMPTZ
  - IDs → SERIAL/IDENTITY oder UUID
- Nicht unnötig VARCHAR-Längen begrenzen
  - TEXT ist meist die bessere Wahl
- JSONB sinnvoll nutzen:
  - Für semi-strukturierte Daten
  - Dennoch relationale Modellierung bevorzugen
- Casting sparsam einsetzen
  - vermeidet Performance-Verluste

## BACKUP & RESTORE

- Ziel: Daten sichern & im Fehlerfall wiederherstellen
- Zwei Backup-Arten:
  - Logische Backups (pg\_dump / pg\_restore)
  - Physische Backups (pg\_basebackup)
- Tools für Einsteiger:
  - pgAdmin (GUI für Backup/Restore)
  - pg\_dump (Kommandozeile für logische Backups)
- Wichtig:
  - Backups regelmäßig testen
  - Backup-Strategie dokumentieren
  - Automatisierung empfohlen

## LOGISCHES BACKUP MIT PG\_DUMP

- pg\_dump erzeugt ein Backup in Form von SQL oder Archivformat
- Backup einzelner Datenbanken, Tabellen oder Schemas
- Vorteile:
  - portabel, unabhängig von PostgreSQL-Versionen (aufwärts kompatibel)
  - flexibel (Einzelobjekte wiederherstellbar)
- Nachteile:
  - langsamer als physische Backups
  - Datenbank während des Dumps sollte wenig Last haben
- Wichtige Formate:
  - -Fc → Custom Format (für pg\_restore)
  - -Fp → Plain SQL
- Beispiel:

```
pg_dump -U postgres -Fc -f backup.dump mydb
```

## RESTORE MIT PG\_RESTORE

- Wird verwendet für Backups im Custom- oder Directory-Format
- Vorteile:
  - selektives Wiederherstellen (Schemas, Tabellen etc.)
  - paralleles Restore möglich
- Beispiel:

```
pg_restore -U postgres -d mydb_restored backup.dump
```

- Typischer Ablauf:
  1. Neue Datenbank anlegen
  2. Restore durchführen
  3. Berechtigungen und Extensions prüfen

## PHYSISCHE BACKUPS (NUR ERWÄHNEN)

- pg\_basebackup – Werkzeug für vollständige Dateisystem-Sicherungen
- Einsatz in der Praxis:
  - Replikation einrichten
  - Große Produktionsdatenbanken
- Eigenschaften:
  - sehr schnell
  - vollständiges PostgreSQL-Datenverzeichnis

## EXPORT / IMPORT IN PGADMIN

- Exportieren:
  - Rechtsklick auf Datenbank → „Backup...“
  - Auswahl von Format: SQL, Custom, Tar
  - Benutzerfreundliche Oberfläche
- Importieren / Wiederherstellen:
  - Rechtsklick auf Datenbank → „Restore...“
  - Auswahl von .backup / .dump
- Grenzen:
  - für große Datenbanken ungeeignet
  - weniger Kontrolle als CLI

## WICHTIG FÜR DIE PRAXIS

- Backups regelmäßig automatisieren (Cron, PowerShell, CI/CD)
- Restore regelmäßig testen (Backup ist nur so gut wie der Restore)
- Speicherung:
  - mehrere Sicherungsorte (Cloud, externe Drives)
  - Versionshistorie aufbewahren
- Dokumentation:
  - Wie, wann und wo wird gesichert?
  - Wer ist verantwortlich?
- Tipp:
  - Mindestens ein Backup im Custom Format (pg\_dump -Fc) erstellen

## BENUTZERDEFINIERTE DATENTYPEN IN POSTGRESQL

- Erlauben eigene Strukturdefinitionen neben Standardtypen
- Steigern Konsistenz, Wiederverwendbarkeit und Lesbarkeit
- Einsetzbar in Tabellen, Funktionen, Views und Stored Procedures
- Typen:
  - Composite Types (zusammengesetzte Strukturen)
  - Enum Types (vordefinierte Wertemengen)
  - Domain Types (Typen mit Constraints)

## PRAXISBEISPIELE

- Kundenadresse als Composite-Type

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

CREATE TABLE customer (
    id SERIAL PRIMARY KEY,
    name TEXT,
    address address_type
);
```

- Bestellstatus als ENUM

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

CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    status order_status NOT NULL
);
```

- Validierte E-Mail als Domain-Type

```
CREATE DOMAIN email_type AS TEXT CHECK (VALUE ~ '^\w+@\w+\.\w+$');

CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email email_type NOT NULL
);
```

## PRAXISBEISPIELE (FORTS.)

- Preis nur positiv – Domain-Type

```
CREATE DOMAIN positive_price AS NUMERIC CHECK (value >= 0);

CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    price positive_price
);
```

- Geokoordinaten als Composite-Type

```
CREATE TYPE geo_coord AS (lat NUMERIC, lng NUMERIC);

CREATE TABLE store (
    id SERIAL PRIMARY KEY,
    location geo_coord
);
```

## BEISPIEL: INSERT INTO MIT COMPOSITE TYPE

Tabelle

```
CREATE TABLE customer (
    id SERIAL PRIMARY KEY,
    name TEXT,
    address address_type
);
```

### BEISPIEL: EINFÜGEN EINES DATENSATZES

Variante 1 – Direkter INSERT mit ROW()

```
INSERT INTO customer (name, address)
VALUES (
    'Max Mustermann',
    ROW('Musterstraße 1', 'Musterstadt', '12345')
);
```

Variante 2 – Direkter INSERT ohne ROW (Kurzform)

```
INSERT INTO customer (name, address)
VALUES (
    'Erika Beispiel',
    ('Beispielweg 10', 'Beispielstadt', '98765')
);
```

Variante 3 – Insert mit benannten Feldern (klarer lesbar)

```
INSERT INTO customer (name, address)
VALUES (
    'Hans Format',
    (street := 'Hauptstraße 5', city := 'Berlin', zip := '10115')
);
```

# SELECT, UPDATE & WHERE MIT COMPOSITE TYPES

## SELECT-Beispiele

1. Gesamtes Objekt auslesen

```
SELECT c.id, c.name, c.address  
FROM customer c;
```

2. Auf einzelne Felder im Composite zugreifen

```
SELECT  
    c.name,  
    (c.address).street AS street,  
    (c.address).city   AS city,  
    (c.address).zip    AS zip  
FROM customer c;
```

## UPDATE-Beispiele

1. Gesamtadresse ersetzen

```
UPDATE customer  
SET address = ('Neue Straße 1', 'Dresden', '01067')  
WHERE id = 1;
```

2. Nur ein Feld im Composite aktualisieren

```
UPDATE customer  
SET address = address::address_type ||  
      ROW(NULL, NULL, '99999') -- zip aktualisieren  
WHERE id = 1;
```

## WHERE-BEISPIELE

1. Nach einem einzelnen Feld filtern

```
SELECT *
FROM customer
WHERE (address).city = 'Berlin';
```

2. Nach vollständiger Adresse filtern

```
SELECT *
FROM customer
WHERE address = ('Musterstraße 1', 'Musterstadt', '12345');
```

3. Teilvergleich über einzelne Spalten

```
SELECT *
FROM customer
WHERE (address).zip LIKE '10%';
```

## WARUM BENUTZERDEFINIERT TYPEN VERWENDEN?

- Wiederholbare Geschäftsregeln an einer Stelle
- Weniger Redundanz (z. B. kein mehrfaches Schreiben von Constraints)
- Klare Datenmodelle, leichter wartbar
- Starke Validierung direkt in der Datenbank

## **LAB TIME**

**AUFGABE 6: EXERCISES/EXERCISE-6-UMGANG-MIT-DATENTYPEN**

## ERWEITERTE AUSWERTUNGEN – ÜBERBLICK

- Erweiterte Techniken zur komplexeren Datenanalyse
- Einsatz bei statistischen Berechnungen, Hierarchieauswertungen, Auswertungen über Zeiträume
- Werkzeuge:
  - Subqueries
  - Window Functions
  - CTEs (WITH-Queries)
  - Ziel: Effizientere, lesbare und wiederverwendbare SQL-Abfragen

## SUBQUERIES (UNTERABFRAGEN)

- Abfragen innerhalb anderer SQL-Statements
- Können in SELECT, FROM, WHERE, HAVING stehen
- Typische Einsatzfälle:
  - Werte dynamisch bestimmen (z. B. max. Preis, durchschnittliche Werte)
  - Ergebnislisten filtern
  - Vergleiche von aggregierten Werten
- Arten:
  - Scalar Subquery (liefert genau 1 Wert)
  - Row Subquery (liefert 1 Zeile)
  - Table Subquery (liefert mehrere Zeilen)

## WINDOW FUNCTIONS – GRUNDLAGEN

- Arbeiten über „Fenster“ von Datensätzen, ohne Gruppierung
- Ermöglichen Analysen über Reihenfolgen, Zeiträume, Partitionen
- Wichtige Window Functions:
  - ROW\_NUMBER, RANK, DENSE\_RANK
  - SUM, AVG, COUNT (als Window-Version)
  - LAG, LEAD
- Syntax:

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

- Vorteil: Einzelzeilen bleiben erhalten → ideal für Trend- & Rankinganalysen

## WINDOW FUNCTIONS – BEISPIEL

```
SELECT
    customer_id,
    order_date,
    amount,
    SUM(amount) OVER (PARTITION BY customer_id ORDER BY order_date) AS running_total
FROM orders;
```

- PARTITION BY = Teilt die Daten je Kunde
- ORDER BY = Chronologische Sortierung
- SUM(... OVER) = Laufende Summe (Running Total)

## CTES (COMMON TABLE EXPRESSIONS – WITH)

- Temporäre Ergebnistabellen für die Query
- Verbessern Lesbarkeit & Struktur komplexer Abfragen
- Können mehrfach genutzt werden
- Unterstützen rekursive Abfragen
- Syntax:

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

## CTE-BEISPIEL

```
WITH high_value_orders AS (
    SELECT *
    FROM orders
    WHERE amount > 1000
)
SELECT customer_id, COUNT(*)
FROM high_value_orders
GROUP BY customer_id;
```

- Ziel: Kunden mit Bestellwert über 1000€ identifizieren
- Ergebnis mehrfach nutzbar
- Saubere, verständliche Struktur

## REKURSIVE CTES – ÜBERBLICK

- Ideal für hierarchische Daten (z. B. Kategorien, Mitarbeiterstrukturen)
- Bestehen aus zwei Teilen:
  1. Anchor Query (Startpunkt)
  2. Recursive Query (wiederholt sich)
- Beispiel:

```
WITH RECURSIVE category_tree AS (
    SELECT id, parent_id, name
    FROM categories
    WHERE parent_id IS NULL

    UNION ALL

    SELECT c.id, c.parent_id, c.name
    FROM categories c
    JOIN category_tree ct ON ct.id = c.parent_id
)
SELECT * FROM category_tree;
```

## **LAB TIME**

**AUFGABE 7: EXERCISES/EXERCISE-7-ERWEITERTE-AUSWERTUNGEN**

## EINFÜHRUNG – KOMPLEXE ABFRAGEN

- Ziel: Daten aus mehreren Tabellen effizient kombinieren und verarbeiten
- Typische Anforderungen:
  - Mehrstufige Filterlogik
  - Aggregationen über Teilmengen
  - Verschachtelte Berechnungen
  - Wiederverwendbare Zwischenergebnisse
- Werkzeuge:
  - CTEs (Common Table Expressions)
  - Subqueries
  - Joins
  - Window Functions
  - Aggregatfunktionen

## BEST PRACTICES FÜR KOMPLEXE SQL-ABFRAGEN

- Klare Struktur nutzen
  - Komplexe Logik in CTEs auslagern („Lesbarkeit vor Performance“)
- Aussagekräftige Aliase für Tabellen & CTEs
- Explizite Joins statt implizite (Vermeidet Cross Joins)
- Filter so früh wie möglich anwenden
- Aggregationen und Window Functions bewusst trennen
- Kommentare bei komplexer Logik setzen
- Schrittweises Vorgehen
  - Erst Grundabfrage
  - Dann Joins hinzufügen
  - Dann Filter anwenden
  - Dann Aggregationen oder Fensterfunktionen ergänzen

## PERFORMANCE-BASICS – INDEXES

- Index = Struktur zur schnelleren Suche (ähnlich wie Buchindex)
- Besonders wichtig bei:
  - WHERE-Bedingungen
  - JOIN-Bedingungen
  - ORDER BY / GROUP BY
- Häufige Index-Typen:
  - B-Tree (Standard, gut für Vergleichsoperatoren)
  - GIN / GiST (Ideal für JSONB, Full-Text-Suche, Arrays)
- Tipps:
  - Nur indizieren, was häufig genutzt wird
  - Zu viele Indexe → langsame INSERT/UPDATE
  - EXPLAIN und EXPLAIN ANALYZE verwenden, um Index-Nutzung zu prüfen

## KOMBINIEREN VON CTEs, JOINS UND AGGREGATEN

- CTEs (WITH) nutzen für:
  - Vorberechnung von Zwischenergebnissen
  - Lesbare, logisch getrennte Abfrageblöcke
- Joins verbinden Daten über Schlüssel:
  - Komplexe Datenmodelle → mehrere JOIN-Ebenen
  - Beispiel: Kunde → Bestellung → Bestellpositionen → Produkt
- Aggregationen einbinden:
  - SUM, COUNT, AVG etc. auf gruppierten Daten
  - HAVING für Filter auf aggregierten Ergebnissen
- Vorgehensweise:
  - CTE für gefilterte Basisdaten
  - Joins für Verknüpfung der relevanten Tabellen
  - Aggregation oder Window Functions auf Ergebnis

## BEISPIEL – KOMPLEXE ABFRAGE KOMBINIERT

- Ziel: Umsatz pro Kunde + Anzahl der Bestellungen + Durchschnitt pro Bestellung
- Nutzung von:
  - CTE für gefilterte Bestellungen (z. B. nur aus letztem Jahr)
  - JOINs für Kunden und Bestellungen
  - Aggregation für Umsatz, Anzahl, Durchschnitt
- Beispiel:

```
WITH current_year AS (
    SELECT * FROM orders
    WHERE EXTRACT(YEAR FROM order_date) = 2024
)
SELECT
    c.name,
    COUNT(o.id) AS bestellungen,
    SUM(o.total) AS gesamtumsatz,
    AVG(o.total) AS durchschnitt_pro_bestellung
FROM current_year o
JOIN customers c ON c.id = o.customer_id
GROUP BY c.name
ORDER BY gesamtumsatz DESC;
```

## **LAB TIME**

**AUFGABE 8: EXERCISES/EXERCISE-8-KOMPLEXE-ABFRAGEN**