Grundlegende Begrifflichkeiten

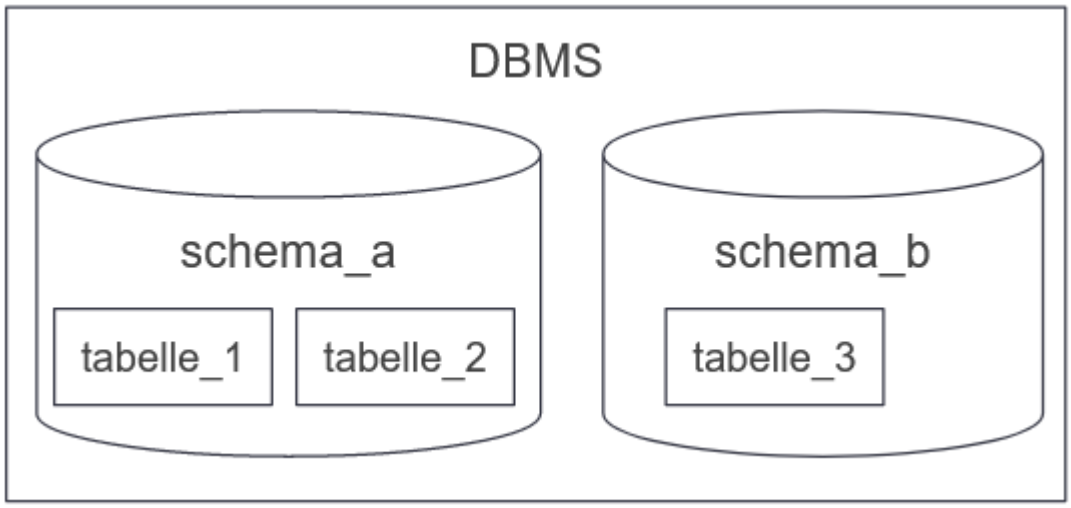
- **Datenbank** (DB)
- **Datenbankmanagementsystem** (DBMS) z.B. Oracle, SQL Server, DB2, MySQL, PostgreSQL und MariaDB
- **SQL** (Structured Query Language) als Datenbanksprache
    - SQL definiert wichtigste Befehle
    - Wird von allen großen DBMS unterstützt
    - Von Organisationen ANSI und ISO standardisiert
    - Da SQL historisch gewachsen haben viele Hersteller SQL Befehle auf ihre Produkte angepasst (bspw. Oracle oder PostgreSQL)
- **Befehlskategorien**
    - DDL (Data Definition Language)
    - DML (Data Manipulation Language)
    - DQL (Data Query Language)
    - TCL (Transaction Control Language)
    - DCL (Data Control Language)

Data Definition Language (DDL)

Referenz: https://www.postgresql.org/docs/13/ddl.html

Schema erstellen

- DBMS kann viele Datenmodelle enthalten
- Diese Bereiche enthalten Tabellen und werden Schemas genannt



Syntax

```
CREATE SCHEMA <Schemaname>;
```

Beispiel

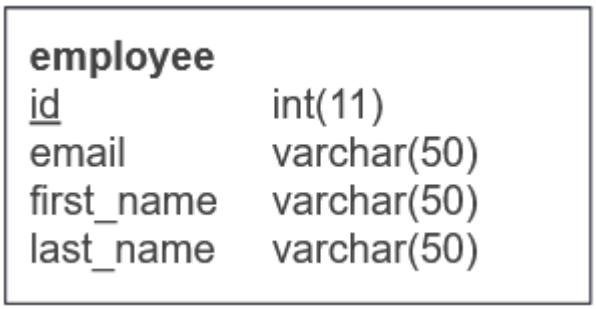
```
CREATE SCHEMA employees;
```

> *Verwenden Sie ein Schema mit* `SET SCHEMA 'employees'`.

Referenz: https://www.postgresql.org/docs/13/sql-createschema.html

Tabelle erstellen

- Schema enthält Tabellen
- Tabellen enthalten Definitionen der Spalten
- Spaltendefinition beinhaltet Name und Datentyp
- Datentypen unterscheiden sich bei DBMS



Syntax

```
CREATE TABLE <Tabellenname> (
    <Name>    <Datentyp>,
    ...
```

```
    [PRIMARY KEY <Spaltenname>]
);
```

Beispiel

```
CREATE TABLE emp_employee (
    emp_id           INTEGER,
    emp_email        VARCHAR(50),
    emp_first_name   VARCHAR(50),
    emp_last_name    VARCHAR(50),
    PRIMARY KEY (emp_id)
);
```

Referenz: https://www.postgresql.org/docs/13/sql-createtable.html

Typische Datentypen

- Datentypen variieren zwischen DBMS

Datentyp	Beschreibung
CHAR(n)	Zeichenkette mit fester Länge (belegt Speicher immer)
VARCHAR(n)	Zeichenkette mit variabler Länge von maximal n
INTEGER	Ganzzahl
DECIMAL(n,m)	Kommazahl mit n Stellen (m Nachkommastellen)
DATE	Datum
DATETIME	Datum und Uhrzeit
BOOL	Wahrheitswert (wahr, falsch)
TEXT, MEDIUMTEXT, LONGTEXT	Große Textdaten
BLOB, MEDIUMBLOB, LONGBLOB	Binärdaten (Binary Large Object)

Referenz: https://www.postgresql.org/docs/13/datatype.html

Auto Increment

- Automatische Generierung von Sequenzen
- MySQL unterscheidet sich zu SQL Standard
- Nur ein AUTO_INCREMENT pro Tabelle
- Muss auf einen Key (z.B. Primary Key) angewandt werden
- Startet standardmäßig bei 1
- Kann niemals < 0 sein

Syntax SQL Standard

```
CREATE TABLE <Tabellenname> (
    id    INTEGER GENERATED BY DEFAULT AS IDENTITY
   (START WITH 1000 INCREMENT BY 1),
    ...
```

```
    PRIMARY KEY id
);
```

Syntax PostgreSQL

```
CREATE TABLE <Tabellenname> (
    id    SERIAL,
    ...
    PRIMARY KEY id
);
```

Syntax MySQL

```
CREATE TABLE emp_employees (
    emp_id    INTEGER AUTO_INCREMENT,
    ...
    PRIMARY KEY (emp_id)
) AUTO_INCREMENT = 1000;
```

Referenz: https://mariadb.com/kb/en/auto_increment/, https://www.postgresql.org/docs/13/datatype-numeric.html#DATATYPE-SERIAL

Löschen und Ändern von Schemas und Tabellen

`DROP` um Schema oder Tabellen zu löschen

Syntax

```
DROP SCHEMA <Schemaname>;
DROP TABLE <Tabellenname>;
```

`ALTER`, um Schema oder Tabelle zu ändern

Syntax

```
ALTER SCHEMA <Schemaname> <Schema>;
```

Syntax Tabelle

```
ALTER TABLE <Tabellenname> <Optionen>;

<Optionen>:
    ADD <Spaltendefinition>
    MODIFY <Spaltendefinition>
    DROP <Spaltendefinition>
```

Beispiel: löschen einer Spalte

```
ALTER TABLE employee DROP COLUMN first_name;
```

Referenz: https://www.postgresql.org/docs/13/ddl-alter.html

Generelle Verwendung

- `CREATE` um Schemas und Tabellen anzulegen (Definition des Datenlayouts)
- NICHT für das Einfügen konkreter Daten - hier werden andere Befehle genutzt
- `CREATE` wird auch für sämtliche andere Objekte genutzt (zum Beispiel Anlegen von Benutzerrechten mit `CREATE USER`)
- `ALTER`, um die DB Definition zu Ändern
- `DROP`, um DB Definitionen zu löschen

Integritätsbedingungen

- Qualitätssicherung der Daten
- Mit Integritätsbedingungen stellt DB sicher, dass diese beim Einfügen, Ändern oder Löschen eingehalten werden
- Bei Verstoß werden Befehle nicht ausgeführt
- SQL Constraints (Bedingungen):
  - NOT NULL-Constraint
  - UNIQUE Constraint
  - CHECK Constraint

Referenz: https://www.postgresql.org/docs/13/ddl-constraints.html

NOT NULL Constraint

- Bei Nutzung von NOT NULL darf Spalte nicht leer sein
- Mit DEFAULT kann beim Leerlassen ein Standardwert gesetzt werden
- NOT NULL und DEFAULT können kombiniert werden

Beispiel NOT NULL und DEFAULT

```
CREATE TABLE emp_employees (
    emp_id    INTEGER,
    emp_email    VARCHAR(50) NOT NULL DEFAULT 'noemail@sth.de',
    PRIMARY KEY (emp_id)
);
```

`emp_email` muss immer gesetzt werden und bekommt ansonsten den Standardwert `noemail@sth.de`.

UNIQUE Constraint

- Werte einer Spalte muss eindeutig sein

Beispiel

```
CREATE TABLE emp_employees (
    emp_id      INTEGER,
    emp_email   VARCHAR(50) UNIQUE,
    PRIMARY KEY (emp_id)
);
```

- Alle Mitarbeiter benötigen eine eindeutige Email
- Keine Mitarbeiter können die selbe Email haben

Beispiel mit Syntaxvariante

```
CREATE TABLE emp_employees (
    emp_id      INTEGER,
```

```
    emp_email   VARCHAR(50),
    …
    CONSTRAINT eindeutig_email UNIQUE (emp_email),
    ...
    PRIMARY KEY (emp_id)
);
```

> *Constraint kann über einen Namen (hier `endeutig_email`) referenziert werden*

Mehrere Felder lassen sich mit `UNIQUE` verbinden:

```
CONSTRAINT name_unique UNIQUE (first_name, last_name)
```

CHECK-Constraint

Zusätzliche Regeln, welche ein Spalteneintrag erfüllen muss

Beispiel

```
CREATE TABLE emp_employees (
    emp_id      INTEGER,
    emp_email   VARCHAR(50) UNIQUE,
    emp_age     INTEGER CHECK (emp_age >= 12)
    PRIMARY KEY (emp_id)
);
```

> *eingetragene Mitarbeiter haben ein Alter von mindestens 12 Jahren*

Beispiel mit Syntaxvariante

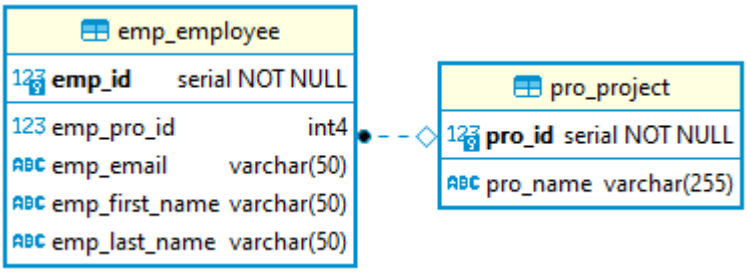
```
CREATE TABLE emp_employees (
    emp_id    INTEGER,
    emp_email    VARCHAR(50),
    emp_age    INTEGER,
    …
    CONSTRAINT altercheck CHECK (emp_age >= 16)
    ...
    PRIMARY KEY (emp_id)
);
```

> *Constraint kann über den Namen `altercheck` referenziert werden*

Referentielle Integrität

- Abbildung von Beziehungen
- Überprüfung ob Referenzen zu anderen Tabellen eingehalten werden

> *"Der Foreign-Key steht bei der N-Entity"*



Syntax

```
CREATE TABLE pro_project (
    pro_id        SERIAL,
    pro_name    VARCHAR(255),

    PRIMARY KEY (pro_id)
);

CREATE TABLE emp_employee (
    emp_id           SERIAL,
    emp_pro_id       INTEGER,
    emp_email        VARCHAR(50),
    emp_first_name   VARCHAR(50),
    emp_last_name    VARCHAR(50),

    PRIMARY KEY (emp_id),
    FOREIGN KEY (emp_pro_id) REFERENCES pro_project(pro_id)
);
```

- Mitarbeiter sind Projekten zugewiesen
- Ein Mitarbeiter **kann ein** Projekt haben
- Durch `NOT NULL` kann eine Referenz erzwungen werden (bspw. Foreign Key muss definiert werden)

Alternativsyntax

`REFERENCES` kann direkt hinter dem Datentyp verwendet werden.

```
CREATE TABLE emp_employee(
    ...
    emp_pro_id INTEGER REFERENCES pro_project (pro_id),
    ...
);
```

Verhalten von REFERENCES

Standardverhalten: Ein Projekt, welches durch Mitarbeiter referenziert wird kann nicht gelöscht werden!

Das Verhalten kann durch die Optionen `ON DELETE` oder `ON UPDATE` gesetzt werden.

Beispiel mit `ON DELETE`

```
CREATE TABLE emp_employee(
    ...
```

```
    emp_pro_id INTEGER REFERENCES pro_project (pro_id) ON DELETE <Verhalten>,
    ...
);
```

Verhalten:

- `RESTRICT` / `NO ACTION` : Löschen eines Projekts, welches einen Mitarbeiter hat ist nicht möglich
- `CASCADE` : Wird ein Projekt gelöscht, werden alle referenzierten Mitarbeiter gelöscht
- `SET NULL` : Wird ein Projekt gelöscht, wird `emp_pro_id` des Mitarbeiters auf `NULL` gesetzt
- `SET DEFAULT` : Wird ein Projekt gelöscht, wird die `emp_pro_id` auf einen Standardwert gesetzt

Zusammengesetzte Schlüssel

Engl: Composite Keys
Mehrere Spalten werden zu einem Schlüssel kombiniert

Beispiel

```
CREATE TABLE emp_employee (
    emp_first_name  VARCHAR(50) NOT NULL,
    emp_last_name   VARCHAR(50) NOT NULL,

    PRIMARY KEY (emp_first_name, emp_last_name)
);
```

> *Mitarbeiter werden über einen eindeutigen Schlüssel aus Vor- und Nachname referenziert.*

Data Modification Language (DML)

DML modifiziert Datensätze.

Daten einfügen

Datensätze werden mit INSERT eingefügt Syntax

```
INSERT INTO <Tabellenname> ( <Spaltennamen> )
VALUES ( <Werte> )
```

Die Werte sind in der gleichen Reihenfolge wie die Spaltennamen Auto-Increment Werte werden erhöht Fehler falls Primärschlüssel bereits existiert

Beispiel

```
INSERT INTO employee (id, email, first_name, last_name)
VALUES (1, 'worker@comp.de', 'Max', 'Muster');
```

Daten ändern

Datensätze werden mit UPDATE geändert Syntax

```
UPDATE <Tabellenname>
SET <Spaltenname>=<Neuer Wert>, ...
[WHERE <Auswahlbedingung>]
```

Bezieht sich immer nur auf eine Tabelle Mit WHERE können zu ändernde Daten gefiltert werden
Auswahlbedingungen können mit AND oder OR verknüpft werden

Beispiel

```
UPDATE employee
SET last_name = 'Mustermann', first_name = 'Maxi'
WHERE email = 'worker@comp.de'
AND last_name='Muster';
```

Daten löschen

Datensätze werden mit DELETE geändert Syntax

```
DELETE FROM <Tabellenname>
[WHERE <Auswahlbedingung>]
```

Wird keine Auswahlbedingung definiert, werden alle Daten der Tabelle gelöscht

Beispiel

```
DELETE FROM employee
WHERE email = 'worker@comp.de';
```

Alle Daten abfragen

Datensätze werden mit SELECT abgefragt. Komplexe Abfragen behandeln wir in der nächsten Vorlesung.

SELECT ist ein Teil der Data Query Language (DQL).

Syntax

```
SELECT * FROM <Tabellenname>;
```

> *Frägt alle Daten aus der Tabelle employee ab*

Beispiel

```
SELECT * FROM employee;
```

Sonstiges

Kommentare

Kommentare werden mit `--` beschrieben:

```
-- erstellt Tabelle
CREATE TABLE bla (...)
```

Skripte

Erstellung der Mitarbeiter-Projekt Beziehung

```
-- IF EXISTS fuehrt das Kommando nur aus, wenn die TABELLE existiert
DROP TABLE IF EXISTS emp_employee;
DROP TABLE IF EXISTS pro_project;
CREATE TABLE pro_project (
    pro_id        SERIAL,
    pro_name    VARCHAR(255),

    PRIMARY KEY (pro_id)
);

CREATE TABLE emp_employee (
    emp_id           SERIAL,
    emp_pro_id       INTEGER,
    -- email muss definiert werden und ist eindeutig
    emp_email        VARCHAR(50) NOT NULL UNIQUE,
    emp_first_name   VARCHAR(50),
    emp_last_name    VARCHAR(50),

    PRIMARY KEY (emp_id),
    FOREIGN KEY (emp_pro_id) REFERENCES pro_project(pro_id)
);
```

Datenmodifikation

```
-- Erstellen
INSERT INTO emp_employee (emp_email, emp_first_name, emp_last_name) VALUES
('nina@email.de', 'Nina', 'Haus');

-- Loeschen
DELETE FROM emp_employee WHERE emp_email = 'nina@email.de';

-- Keine Email nicht moeglich
INSERT INTO emp_employee (emp_first_name, emp_last_name) VALUES ('Nina', 'Haus');

-- Aktualisieren
UPDATE emp_employee
SET emp_last_name = 'Wohnung'
WHERE emp_email = 'nina@email.de';
```

Data Query Language (DQL)

SQL Befehle für die Abfrage von Daten.

Der SELECT Befehl

Datensätze werden mit `SELECT` abgefragt

Es lassen sich viele verschiedene Optionen definieren

Bestandteile

- **SELECT** Befehlsbeginn
- **FROM** Auswahl der Tabellen für den Abfragebefehl
- **WHERE** Auswahlbedingungen der auszuwertenden Datensätze
- **GROUP BY** Bedingungen nach welcher Gruppierungen vorgenommen werden
- **HAVING** Auswahl von Gruppen
- **ORDER BY** Auswahl der Sortierung

Syntax

```
SELECT <Spaltennamen>
  FROM <Tabellennamen>
  [WHERE <Auswahlbedingungen>]
  [GROUP BY <Gruppierung>
    [HAVING <Gruppierungsauswahl>]
  ]
  [ORDER BY <Sortierung>]
```

Einfache Abfragen

Für eine Abfrage sind lediglich `SELECT` und `FROM` Pflicht. Bei `SELECT` werden die rückzugebenden Spalten ausgewählt. Ein Stern (*) gibt alle Daten zurück. Nach FROM werden alle notwendigen Tabellen der Abfrage definiert.

Syntax

```
SELECT <Spaltennamen> FROM <Tabellennamen>;
```

Beispiele

```
SELECT * FROM emp_employee;
```

> *Gibt alle Datensätze der Tabelle emp_employee zurück*

```
SELECT emp_id FROM emp_employee;
```

> *Gibt alle `emp_id` Spalten der employee Tabelle zurück*

```
SELECT * FROM emp_employee, pro_project;
```

> *Gibt alle Datensätze der Tabelle emp_employee und pro_project zurück - Datensätze werden multipliziert!*

```
SELECT emp_id, pro_project FROM emp_employee, pro_project;
```

> *Gibt die `emp_id` und `pro_id` der Tabellen employee und mentor zurück*

Dopplungen

Mit `DISTINCT` werden Dopplungen entfernt

Beispiele

```
SELECT DISTINCT emp_pro_id FROM emp_employee;
```

> *Gibt alle Projekt Ids der Mitarbeitertabelle zurück, wobei alle doppelten ids zusammengefasst werden*

```
SELECT DISTINCT first_name FROM emp_employee;
```

> *Gibt alle Vornamen der Mitarbeiter zurück, wobei jeder Vorname eindeutig ist*

Limit

Mit `LIMIT <anzahl>` wird die Anzahl der Ergebnisse limitiert

Beispiel

```
SELECT * FROM emp_employee
LIMIT 1;
```

> *Gibt den ersten Mitarbeiter aus (sortiert nach der id)*

Aliasnamen

Mit `AS` können Aliasnamen definiert werden.

Diese geben der Spalte einen temporären Namen, welche für die Definition von weiteren Abfragen verwendet werden kann. Damit lassen sich Tabellennamen kürzen und eindeutige Tabellennamen definieren. Dies ist wichtig, falls zwei Tabellen den selben Spaltennamen enthalten.

Beispiele

```
SELECT first_name AS vorname FROM emp_employee;
```

> *Wählt alle Vornamen der Mitarbeiter aus und gibt die Datensätze als `vorname` zurück*

```
SELECT ee.first_name FROM emp_employee AS ee;
```

> *Definiert die employee Tabelle als `ee`, diese kann nun verkürzt im SELECT verwendet werden*

Auswahlbedingung

Basisoperatoren

Mit `WHERE` werden Datensätze ausgewählt, welche im Ergebnis berücksichtigt werden sollen. Die Syntax ist gleich wie bei `UPDATE` und `DELETE`.

> ***NULL ist nie gleich oder ungleich!*** > `=` *oder* `<>` *NULL liefert daher keine Ergebnisse*

Vergleichsoperator	Beschreibung
=	Ist gleich
<>	Ist ungleich
<, >, <=, >=	Kleiner (gleich), größer (gleich)

AND	Und Verknüpfung - beide Fälle müssen zutreffen
OR	ODER-Verknüpfung - mindestens ein Fall muss zutreffen

Beispiel

```
SELECT email FROM emp_employee
WHERE first_name <> 'Karl' AND age < 30;
```

> *Gibt alle Datensätze der Tabelle `emp_employee` zurück, bei welchen der Vorname ungleich `Karl` ist und das Alter geringer als 30*

BETWEEN

Abfrage von Datensätzen zwischen zwei Werten

Syntax

```
BETWEEN <Untere Grenze> AND <Obere Grenze>
```

Beispiel

```
SELECT * FROM emp_employee
WHERE age BETWEEN 16 AND 30;
```

> *Alle Mitarbeiter zwischen 16 und 30 (inklusive)*

Alternativsyntax

```
SELECT * FROM emp_employee
WHERE age >= 16 AND age <= 30;
```

IN

Filterung aller Werte, welche in einer gegebenen Liste enthalten sind.

Syntax

```
IN (<Werteliste>)
```

```
SELECT * FROM emp_employee
WHERE first_name IN ('Karl', 'Jan', 'Frieder');
```

> *Alle Mitarbeiter deren Vornamen Karl, Jan oder Frieder ist*

Alternativsyntax

```
SELECT * FROM emp_employee
WHERE first_name='Karl' OR first_name='Jan' OR first_name='Frieder';
```

LIKE

Vergleich von Strings % als Wildcard repräsentiert beliebige Charaktere von beliebiger Länge _ stellt einen beliebigen Buchstaben dar

```
SELECT * FROM emp_employee
WHERE first_name LIKE 'K%';
```

> *Alle Mitarbeiter deren Vornamen mit `K` startet*

IS NULL

- Basis-Vergleichsoperatoren (`=` , `<>`) können keine NULL Werte vergleichen
- Dies wird mit `IS NULL` und `IS NOT NULL` abgefragt

```
SELECT * FROM emp_employee
WHERE first_name IS NULL;
```

> *Alle Mitarbeiter deren Vornamen nicht gesetzt ist*

```
SELECT * FROM emp_employee
WHERE first_name IS NOT NULL;
```

> *Alle Mitarbeiter deren Vornamen gesetzt ist*

NOT

- Negiert die Bedingung
- Wird vor den Operator geschrieben mit Ausnahme von IS NULL - hier wird es zwischengestellt: IS NOT NULL

Beispiele

```
SELECT * FROM emp_employee
WHERE age NOT BETWEEN 16 AND 30;
```

```
SELECT * FROM emp_employee
WHERE first_name NOT IN ('Karl', 'Foo');
```

```
SELECT * FROM emp_employee
WHERE first_name NOT LIKE 'K%';
```

Komplexe Auswahlbedingungen

Auswahlbedingungen können kombiniert und mit Klammern () abgegrenzt und klar definiert werden:

```
SELECT * from emp_employee
WHERE first_name LIKE '%K' OR (age > 30 AND first_name IS NOT NULL);
```

> *Wähle alle Mitarbeiter aus, deren Vorname mit `K` endet und Mitarbeiter die Älter als 30 sind, falls ihr Vorname definiert ist*

Sortierung

Mit `ORDER BY` wird die Reihenfolge des Datenresultats gesetzt

Syntax

```
SELECT <Spalten> FROM <Tabellenname>
ORDER BY <Spaltennamen> [ASC | DESC];
```

- Mit **ASC** (eng. ascending) wir aufsteigend und mit **DESC** (eng. descending) absteigend sortiert
- **ASC** ist die standardmäßige Auswahl
- **NULL** Werte haben den kleinsten Wert und kommen bei **ASC** immer zu Beginn

Beispiel

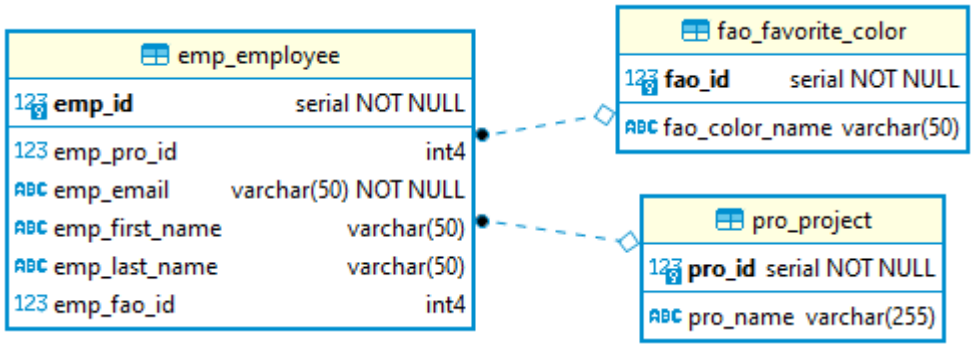
```
SELECT * FROM emp_employee
ORDER BY age;
```

> *Sortiert aufsteigend nach dem Alter*

```
SELECT * FROM emp_employee
ORDER BY first_name DESC, last_name ASC;
```

> *Sortiert absteigend nach dem Vornamen und, falls dieser gleich ist, absteigend nach dem Nachnamen*

Joins



JOIN

Um zwei verknüpfte Tabellen miteinander auszuwerten wird ein JOIN verwendet.

Syntax

```
<Tabelle1> [Alias1] <JoinTyp> JOIN <Tabelle2> [Alias2] [ON <JoinBedingung>]
```

Aliasnamen können für uneindeutige Spaltennamen verwendet werden. Ein Prefix, wie im Beispiel `emp`, `fao` oder `pro` können Uneindeutigkeiten verhindern.

Beispiel

```
SELECT emp_email
FROM emp_employee
INNER JOIN pro_project ON pro_id = emp_pro_id
WHERE pro_name LIKE 'tree_planting';
```

> *Finde alle Mitarbeiter, welche im Projekt `tree_planting` sind und gebe deren Email zurück*

Zusammenführung mehrerer Tabellen

Die Join Syntax kann beliebig erweitert werden. Aus Lesbarkeit bietet es sich an jedes Statement in eine Zeile zu schreiben.

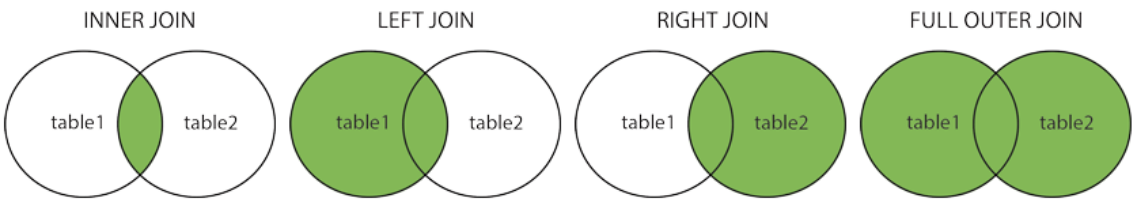
```
SELECT ...
FROM ...
INNER JOIN ... ON ...
INNER JOIN ... ON ...
```

Beispiel

```
SELECT e.emp_email as email, p.pro_name as project_name, f.fao_color_name as
favorite_color
FROM emp_employee e
INNER JOIN fao_favorite_color f ON f.fao_id = e.emp_fao_id
INNER JOIN pro_project p ON p.pro_id = e.emp_pro_id;
```

> *Gebe Email, Projektnamen und Lieblingsfarbe aller Mitarbeiter zurück*

Join-Arten



Quelle: https://www.w3schools.com/sql/sql_join.asp

- Mit dem `INNER JOIN` und dem `LEFT OUTER JOIN` lassen sich die meisten Joins abbilden
- `RIGHT OUTER JOIN`, `NATURAL JOIN`, `CROSS JOIN`, `FULL OUTER JOIN` sind weitere Arten, welche in der Praxis jedoch weniger Relevanz haben

```
LEFT OUTER JOIN
```

Gibt auch alle Datensätze zurück, welche nicht der Auswahlbedingung entsprechen

```
SELECT emp_email as email, fao_color_name as favorite_color
FROM emp_employee
LEFT OUTER JOIN fao_favorite_color ON fao_id = emp_fao_id
```

> *Selektiert auch alle Mitarbeiter-Emails, welche keine Lieblingsfarbe haben. Die Lieblingsfarbe (`favorite_color`) ist dann `NULL`.*

Alternativsyntax

Die Join Syntax lässt sich auch in einer `WHERE` Bedingung beschreiben.

Beispiel

```
SELECT emp_email
FROM emp_employee
INNER JOIN pro_project ON pro_id = emp_pro_id
WHERE pro_name LIKE 'tree_planting';
```

Ist gleich wie

```sql
SELECT emp_email
FROM emp_employee, pro_project
WHERE pro_id = emp_pro_id
AND pro_name LIKE 'tree_planting';
```

Aggregatfunktionen

Mit Aggregatfunktionen können Gesamtergebnisse des Resultatsets ermittelt werden. Sie können Bestandteil von `SELECT`, `HAVING` oder `ORDER BY` sein.

Beispiele

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

> *Anzahl der Mitarbeiter ermitteln*

```sql
SELECT COUNT(emp_first_name) FROM emp_employee;
```

> *Anzahl der Mitarbeiter, welche einen Wert für den Vornamen definiert haben*

Aggregatfunktion	Beschreibung
`COUNT(*)`	Anzahl der Datensätze
`COUNT(<Spalte>)`	Anzahl der Datensätze in der Spalte ohne NULL Werte
`SUM(<Spalte>)`	Numerische Summer der Spalte (nur für Zahlentypen gültig)
`AVG(<Spalte>)`	Durchschnittswert der Spaltenwerte
`MIN(<Spalte>)`	Minimaler Wert der Spaltenwerte
`MAX(<Spalte>)`	Maximaler Wert der Spaltenwerte

Gruppierung

GROUP BY

Mit `GROUP BY` können im `SELECT` Datensätze auch gruppiert werden.

Beispiel

```sql
SELECT emp_pro_id, count(*) as anzahl
FROM emp_employee
GROUP BY emp_pro_id;
```

> *Gib Anzahl der Mitarbeiter aus, welche einen Manager teilen*

HAVING

Mit HAVING können Gruppenergebnisse weiter eingeschränkt werden

Beispiel

```
SELECT emp_pro_id
FROM emp_employee
GROUP BY emp_pro_id
HAVING COUNT (emp_pro_id) >= 2;
```

> *Ids der Projekte, welche mindestens 2 Mitarbeiter haben*

Unterabfragen

In `SELECT` Abfragen können Unterabfragen verschachtelt werden, welche sich auch auf andere Tabellen beziehen können.

Beispiel

```
SELECT e.emp_email FROM emp_employee AS e
WHERE e.emp_id IN (
  SELECT m.emp_id FROM pro_project AS m
);
```

Liefert das gleiche Ergebnis wie

```
SELECT DISTINCT e.emp_email
FROM emp_employee AS e
INNER JOIN pro_project ON pro_id=emp_pro_id;
```

Beispiele

```
SELECT * FROM emp_employee
WHERE emp_pro_id = (
  SELECT pro_id FROM pro_project WHERE pro_name='tree_planting'
);
```

> *Abfrage aller Mitarbeiter, welche im Projekt `tree_planting` sind.*

```
SELECT * FROM pro_project AS p
WHERE NOT EXISTS (
  SELECT e.emp_pro_id FROM emp_employee AS e WHERE e.emp_pro_id = p.pro_id
);
```

> *Alle Projekte, welche von keinen Mitarbeitern besetzt sind*

Unterabfragen in anderen Befehlen

In der Praxis können Unterabfragen Beispielsweise verwendet werden, um die technische id (Primary Key) eines Business Keys (bspw. Email) zu erfragen.

Beispiel

```
INSERT INTO emp_employee (emp_pro_id, emp_email, emp_first_name, emp_last_name,
emp_fao_id)
VALUES (
  -- project id
  (SELECT pro_id FROM pro_project WHERE pro_name LIKE 'tree_planting'),
```

```
  -- email
  'pete@email',
  -- name
  'Pete', 'Eat',
  -- favorite color
  (SELECT fao_id FROM fao_favorite_color WHERE fao_color_name LIKE 'blue')
);
```

Zusätzliche Funktionen

Datenbanken stellen weitere Befehle zur Verfügung, welche jedoch oft auch abhängig vom DBMS implementiert sind.

Funktion	Beschreibung
NOW()	Aktuelles Datum und Zeit, kann bspw. für die Abfrage von Datensätzen nach oder vor dem jetzigen Zeitpunkt verwendet werden
UPPER()	String in Großbuchstaben umwandeln
LOWER()	String in Kleinbuchstaben umwandeln

Beispiel der Verwendung

```
SELECT UPPER(emp_first_name) FROM emp_employee;
```

> *Gibt alle Vornamen der Mitarbeiter in Großbuchstaben aus*

Skripte

```
DROP TABLE IF EXISTS emp_employee;
DROP TABLE IF EXISTS pro_project;
DROP TABLE IF EXISTS fao_favorite_color;

CREATE TABLE pro_project (
  pro_id        SERIAL,
  pro_name    VARCHAR(255),

  PRIMARY KEY (pro_id)
);

CREATE TABLE fao_favorite_color (
  fao_id          SERIAL,
  fao_color_name  VARCHAR(50),

  PRIMARY KEY (fao_id)
);

CREATE TABLE emp_employee (
  emp_id           SERIAL,
  emp_pro_id       INTEGER,
  emp_email        VARCHAR(50) NOT NULL UNIQUE,
  emp_first_name   VARCHAR(50),
```

```
  emp_last_name     VARCHAR(50),
  emp_fao_id        INTEGER,


  PRIMARY KEY (emp_id),
  FOREIGN KEY (emp_pro_id) REFERENCES pro_project(pro_id),
  FOREIGN KEY (emp_fao_id) REFERENCES fao_favorite_color(fao_id)
);

INSERT INTO fao_favorite_color(fao_color_name) VALUES
('red'),
('green'),
('yellow'),
('blue');

INSERT INTO pro_project(pro_name) VALUES
('tree_planting'),
('car_repair'),
('weather_forecast');

INSERT INTO emp_employee (emp_pro_id, emp_email, emp_first_name, emp_last_name,
emp_fao_id)
VALUES (
  -- project id
  (SELECT pro_id FROM pro_project WHERE pro_name LIKE 'tree_planting'),
  -- email
  'pete@email',
  -- name
  'Pete', 'Eat',
  -- favorite color
  (SELECT fao_id FROM fao_favorite_color WHERE fao_color_name LIKE 'blue')
);
INSERT INTO emp_employee (emp_pro_id, emp_email, emp_first_name, emp_last_name,
emp_fao_id)
VALUES (
  -- project id
  (SELECT pro_id FROM pro_project WHERE pro_name LIKE 'car_repair'),
  -- email
  'foo@email',
  -- name
  'Foo', 'Bar',
  -- favorite color
  (SELECT fao_id FROM fao_favorite_color WHERE fao_color_name LIKE 'yellow')
);
```

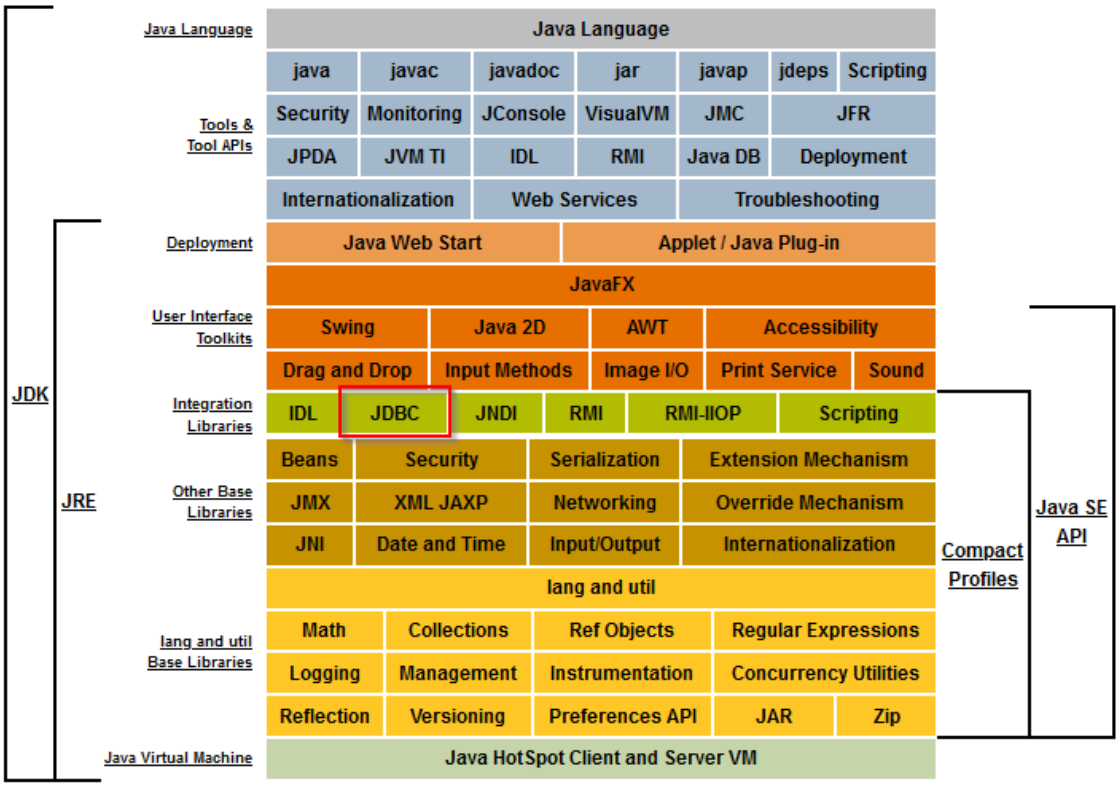
Was ist JDBC?

- JDBC (Java Database Connectivity) ist die Standard-Schnittstelle für den Zugriff auf DBs mittels SQL aus Java-Anwendungen.
- JDBC besteht aus einer Sammlung von Klassen und Interfaces in den Paketen java.sql / javax.sql
- JDBC enthält keinen datenbankspezifischen Code

- JDBC ist eine Abstraktionsschicht und ermöglicht eine Datenbankneutralität bzw. Austausch des DBMS

Eigenschaften

- Integrierter Bestandteil der Sprache Java
- Enthalten in J2SE- und J2EE-Releases
- Anwendung kann unabhängig vom DBS implementiert werden
  - *Write Once, Run Anywhere*
  - SQL-Anweisungen werden als Text (Strings) übertragen
  - JDBC-Treiber transformieren JDBC-SQL in DBMS-SQL
- DBMS-Anbieter implementieren und erweitern den Standard mit ihren eigenen JDBC-Treibern
  JDBC Driver API für die Implementieren von Treibern



From https://docs.oracle.com/javase/8/docs/

Treibertypen

Quelle abgerufen am 28.02.2021

Typ 1

- JDBC-ODBC (Open Database Connection) Bridge
- Ziel: unabhängiges Protokoll zwischen Datenbanken und Programm
- Deprecated in JDK 7 (JDBC 4.1)
- In JDK 8 (JDBC 4.2) entfernt

Quelle

Typ 2

- Native-API (thick)
- Spezielle Treiber des jeweiligen Datenbankherstellers
- Proprietär
- Betriebssystemabhängig
- Nicht alle Hersteller bieten native Treiber
- Beispiel: Oracle OCI Treiber

Typ 3

- Network-Protocol-Treiber / Middleware-Treiber
- Komplett in Java geschrieben
- Keine spezielle Installation erforderlich
- Treiber ist für die Kommunikation mit einer DB auf eine Middleware angewiesen
- DBMS kann problemlos ersetzt werden
- Three-Tier-Architektur

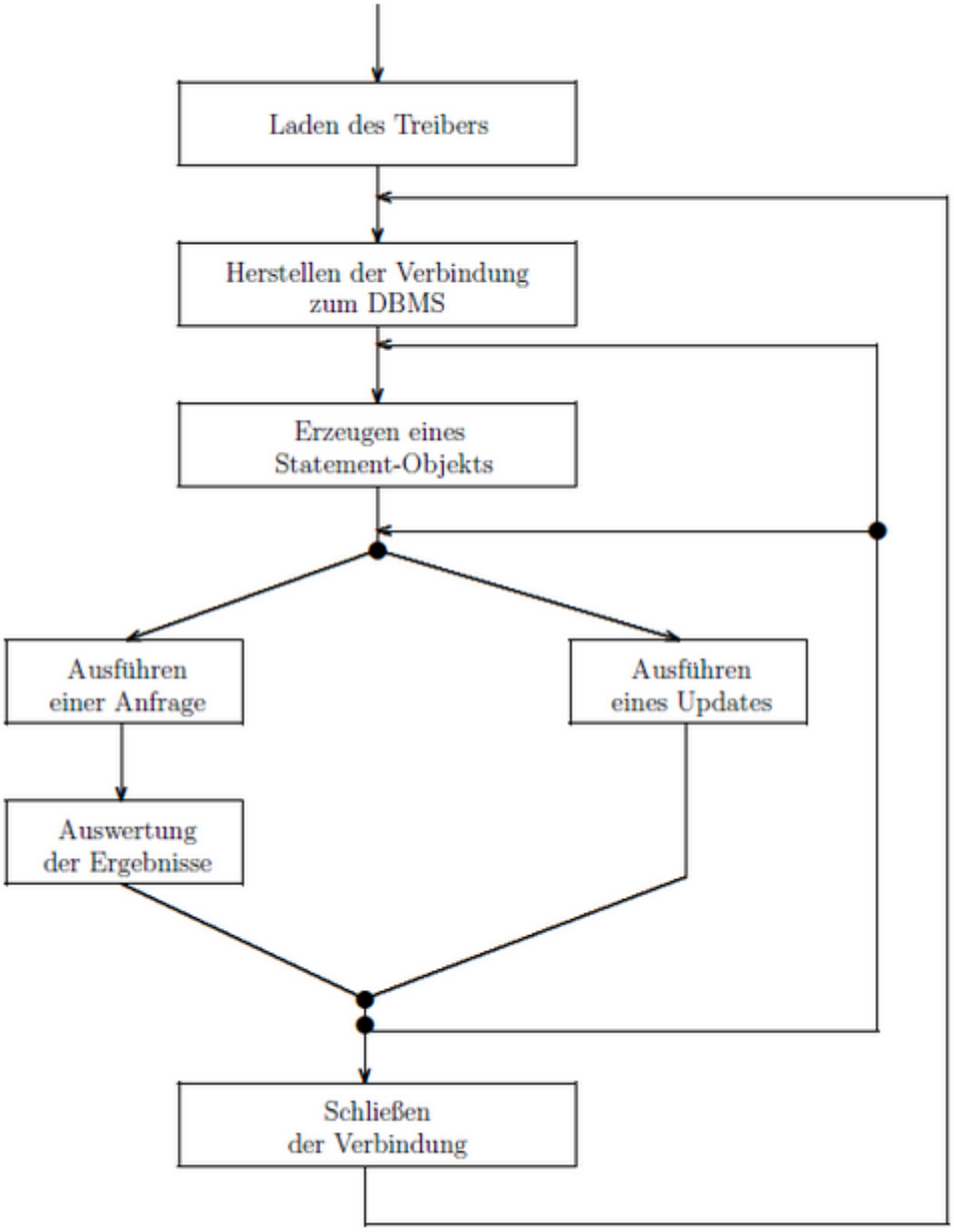
Typ 4

- Database-Protokoll-Treiber (Pure)
- Komplett in Java geschrieben
- Setzt die JDBC-Calls direkt in das erforderliche Protokoll der jeweiligen Datenbank um
- Plattformunabhängig
- DBMS-abhängig

Wichtige Klassen

```
import java.sql.*:
/*
  DriverManager
  Connection
  Statement, PreparedStatement
  ResultSet
  ResultSetMetadata
  SQLException
*/
```

Programmierung



Quelle

Verbindung herstellen

```
String url = String.format("jdbc:postgresql://localhost:5432/postgres?
currentSchema=%s", "schema_name");
Properties props = new Properties();
props.setProperty("user", "postgres");
props.setProperty("password", "1234");

// create the connection
Connection conn = DriverManager.getConnection(url, props);
```

```
// ... use the connection ...

// free the connection
conn.close();
```

Statement ausführen

```
// ...
Statement stmt = conn.createStatement();
String sql = "SELECT * FROM emp_employee";
ResultSet rs = stmt.executeQuery(sql);
// ...
```

- Abfragen mit executeQuery (SELECT)
- Änderungen mit executeUpdate (DELETE, INSERT, UPDATE)

Resultat auswerten

```
while (rs.next()) {
    Integer id = rs.getInt("emp_id");
    String email = rs.getString(2);
}
```

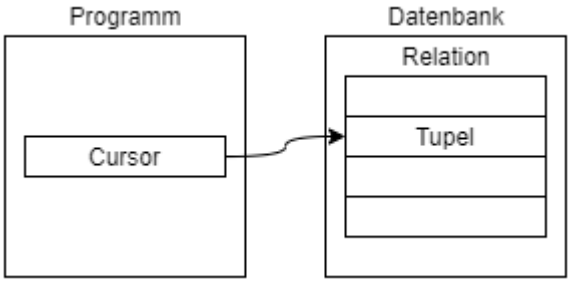
Abfragen der Datenwerte mit

> `getXXX(Position | Spaltenname)`, *wobei* `XXX` *ein passender Java Datentyp ist.*

getString(...) funktioniert für alle Spaltentypen.

Cursor Konzept

Das Resultat der DB wird mit einem Cursor durchlaufen.



Problem: Kopplung von SQL und Programmiersprache durch unterschiedliche Datenstrukturen (Relation vs. Tupel)

Lösung: Cursor als Iterator über die verschiedenen Tupel (Tupel enthält eine Liste an Elementen)

Beispiel: Rowcount

```
Statement stmt = conn.createStatement();
String sql = "SELECT COUNT(*) AS rowcount FROM emp_employee";
ResultSet rs = stmt.executeQuery(sql);
rs.next();
int count = rs.getInt(1);
// ODER: int count = rs.getInt("rowcount");
rs.close();
```

Fehlerbehandlung

Alle JDBC relevanten Funktionen können Fehler werfen und müssen entsprechend abgefangen werden.

`SQLException` wird für alle SQL und DBMS Fehler geworfen und muss entsprechend behandelt werden.

```
try {
    // JDBC Methoden
} catch (SQLException e) {
    e.printStackTrace();
}
```

Resourcenfreigabe

```
Connection conn = DriverManager.getConnection(url, user, password);
Statement stmt = conn.createStatement();
String sqj = "SELECT COUNT(*) AS rowcount FROM emp_employee";

ResultSet rs = stmt.executeQuery(sql);

// ... Abfrageauswertung

rs.close();
stmt.close();
conn.close();
```

> *Das Resultat Set, die Anweisung (Statement) und die Verbindung sollte immer am Ende einer Auswertung geschlossen werden. Ansonsten werden die Ressourcen nicht direkt freigegeben.*

SQL Injection

Mit `Statement.executeQuery(..)` kann bösartiger Code in die Query gelangen.

```
String id = "1 OR 1=1"; // 1=1 is injected code and the query will return all results

// ... id is a function parameter
ResultSet rs = stmt.executeQuery("SELECT * FROM favorite_number WHERE id = " + id);
// ...
```

Prepared Statements

Mit PreparedStatements können sichere Abfragen gestaltet werden.

- Parameter werden in der Query mit `?` gekennzeichnet
- diese werden nach der Erzeugung mit `setXXX()` gesetzt
- `XXX` ist ein passender Datentyp
- Es werden SQL Injections verhindert, da Parameter direkt an die DB geschickt werden und nicht wie bei einer einfachen Query geparsed werden

```
PreparedStatement preparedStmt =
conn.prepareStatement("SELECT * FROM emp_employee WHERE emp_email = ?");
preparedStmt.setString(1, employeeEmail);
ResultSet rs = preparedStmt.executeQuery();
```

Rechteverwaltung

- Zugriffskontrolle
- Prinzip “minimale Rechte” - nur Zugriff auf relevante Daten
- Benutzerbasierte Zugriffe
- Direkte Kopplung an einzelne Nutzer
- Schnell unübersichtlich
- Risikobehaftet
- Rollenbasierte Zugriffe
- eng. RBAC (Role Based Access Control)
- Nutzer werden in Rollen eingeteilt
- Rechte werden an Rollen abgetreten
- Nutzer erhalten Rechte indirekt über Gruppen
- Vereinfachte Konfiguration und Verteilung von Rechten

Referenz https://aws.amazon.com/de/blogs/database/managing-postgresql-users-and-roles/

Grundbefehle

```
CREATE ROLE
GRANT
REVOKE
```

```
CREATE USER
```

Rollen

Mit `CREATE ROLE` werden Rollen erstellt. Neu erstellte Rollen haben keine Rechte.

Syntax

```
CREATE ROLE <Name>;
```

Beispiel

```
CREATE ROLE sales;
```

Mit `DROP ROLE` werden Rollen gelöscht

Syntax

```
DROP ROLE <Name>;
```

Beispiel

```
DROP ROLE sales;
```

Rechtevergabe

Die Rechtervergabe wird mit `GRANT` und `REVOKE` geregelt.

Syntax

```
GRANT <Berechtigung> ON <Datenbankobjekt> TO <Rollenname/Nutzername>
```

Das Datenbankobjekt wird hier aus Schema und Tabellenname zusammengesetzt.

Beispiele

```
GRANT SELECT ON company.emp_employee TO human_resources;
GRANT ALL ON ALL TABLES IN SCHEMA company TO sales; -- erlaube zugriff auf alle tabellen im schema
GRANT SELECT (first_name, last_name) ON company.employee TO other_role;
```

> *`sales` und `employee` sind verschiedene Rollen*

Berechtigung	Beschreibung
`SELECT (<Spalten>)`	Verwendung von SELECT, Spalten sind optional
`INSERT (<Spalten>)`	Verwendung von INSERT, Spalten sind optional
`UPDATE (<Spalten>)`	Verwendung von UPDATE, Spalten sind optional
`DELETE`	Verwendung von DELETE
`CREATE`	Verwendung von CREATE TABLE
`ALL`	Verwendung aller gelisteten

Rollenattribute

Weitere Attribute koennen fuer die Erstellung einer Rolle angegeben werden. Wichtige Attribute sind:

`SUPERUSER / NOSUPERUSER (default)` : neue Rolle ist ein Superuser und hat keine Restriktionen

`CREATEDB / NOCREATEDB (default)` : Rolle kann neue Datenbanken anlegen

`CREATEROLE / NOCREATEROLE (default)` : Rolle kann andere Rollen verwalten

`INHERIT (default) / NOINHERIT` : Privilegien einer anderen Rolle werden geerbt

`LOGIN / NOLOGIN (default)` : gibt der Rolle die Moeglichkeit sich einzuloggen

`PASSWORD <password>` : wenn die Rolle LOGIN spezifiziert, kann mit PASSWORD das Passwort des Logins gesetzt werden

`IN ROLE <Rollenname>` : weisst den angegebenen Rollenname als Elternrolle hinzu. Mit INHERIT erbt die neue Rolle alle Attribute.

Beispiele

```
CREATE ROLE jonathan WITH LOGIN PASSWORD '1234';
```

```
CREATE ROLE superfred WITH LOGIN PASSWORD '1234' CREATEDB CREATEROLE;
```

```
CREATE ROLE emmy WITH LOGIN PASSWORD '1234' IN ROLE sales;
```

https://www.postgresql.org/docs/13/sql-createrole.html

https://www.postgresql.org/docs/13/role-attributes.html

Rechte aufloesen

Mit `REVOKE` werden vergebene Rechte zurückgenommen.

Syntax

```
REVOKE <Berechtigung> ON <Datenbankobjekt> FROM <Rollenname/Nutzername>
```

Beispiel

```
REVOKE SELECT ON company.employee FROM sales;
```

Nutzerverwaltung

Nutzer anlegen

Syntax

```
CREATE USER <Name> WITH PASSWORD BY <Passwort>
```

Beispiel

```
CREATE USER fred WITH PASSWORD 'Passwort1234!';
GRANT mentor TO fred;
```

`CREATE USER` ist an Alias fuer `CREATE ROLE + LOGIN PERMISSION`

Mit DROP USER werden Nutzerkonten gelöscht.

Syntax

```
DROP USER <Nutzername>
```

Stored Procedures

Mit gespeicherten Prozeduren kann Geschaeftslogik in der Datenbank implementiert werden. Bisher haben wir die Logik in Java / der Applikationsseite realisiert.

Motivation

> *"Security is a key reason. Banks commonly use stored procedures so that applications and users don't have direct access to the tables. Stored procedures are also useful in an environment where multiple languages and clients are all used to perform the same operations."*

Quelle

Programmierung

- SQL-Befehle können direkt in der Prozedur verwendet werden
- Mit `SELECT … INTO <Variable>` werden Abfragen in Variablen gespeichert
- Diese können zum Beispiel in IF Verzweigungen ausgewertet werden

Beispiel

```
DROP PROCEDURE IF EXISTS insert_data;
CREATE PROCEDURE insert_data(
  IN col_name varchar,
  IN pro_name varchar
)
LANGUAGE SQL
AS $$
  INSERT INTO fao_favorite_color(fao_color_name) VALUES (col_name);
  INSERT INTO pro_project(pro_name) VALUES (pro_name);
$$;

CALL insert_data('pink', 'candy_shop');
```

Fuegt Datensaetze in die beiden Tabellen hinzu.

Stored procedures wurden in PostgeSQL 11 hizugfuegt und folgen nicht komplett dem SQL-Standard.

Quelle

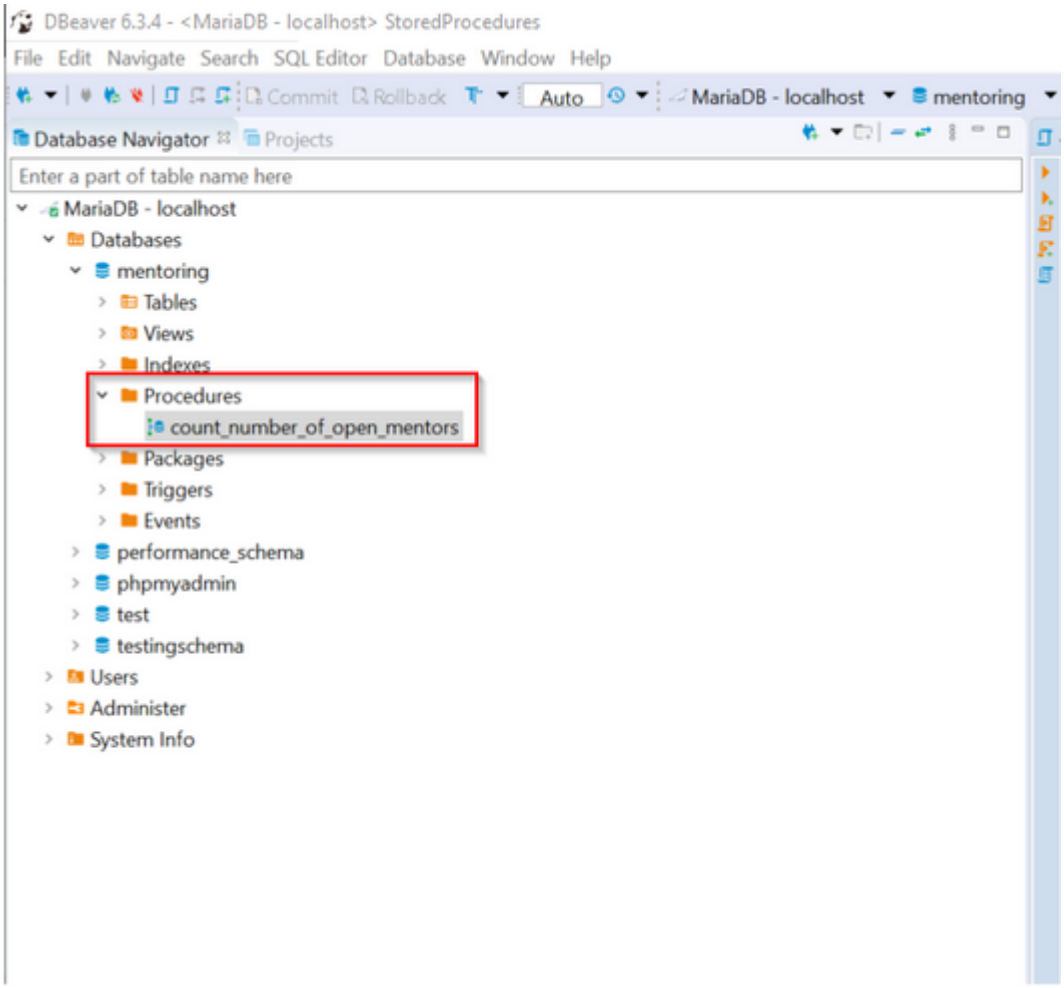
Stored Procedure in JDBC

- Stored Procedures werden in JDBC mit einem CallableStatement aufgerufen.
- Der Aufruf muss in geschweiften Klammern geschrieben werden
- "OUT"-Parameter müssen mit ihrem Typ registriert werden
- "IN"-Parameter werden wie in Vorlesung 3 durch setXXX() gesetzt wobei XXX der Datentyp ist

```
public void insertDataProcedure() throws SQLException {
    Connection conn = DriverManager.getConnection(...);

    String sql = "CALL insert_data(?,?)";
    CallableStatement stmt = conn.prepareCall(sql);

    stmt.setString(1, "brown");
    stmt.setString(2, "project x");

    stmt.executeUpdate();

    stmt.close();
```

```
    conn.close();
}
```

Stored Procedures in DBeaver



Functions

- Syntax ähnlich wie bei Stored Procedures
- Werden direkt in Abfragen oder anderen Statements verwendet
- Bekannte Beispiele sind zb. `UPPER`, `LOWER` (siehe Vorlesung 2)
- Mit `CREATE FUNCTION` werden neue Funktionen definiert
- Functions sind atomar

Beispielfunktion

```
CREATE FUNCTION anonymize(name varchar(100))
RETURNS varchar(4)
AS $$
BEGIN
    IF name IS NOT NULL THEN
```

```
        RETURN CONCAT(SUBSTRING(name,1,1), '***');
    ELSE
        RETURN NULL;
    END IF;
END;
$$
LANGUAGE plpgsql;
```

Beispielverwendung

```
SELECT anonymize(emp_email)
FROM emp_employee;
```

> *Gibt eine Liste aller Mitarbeiter zurück, wobei die Email nur den ersten Buchstaben und `***` enthält*

Quelle

Trigger

- Wird mit CREATE TRIGGER angelegt
- Methodenrumpf ähnlich wie Stored Procedures oder Functions
- Werden automatisch bei bestimmten Datensatzänderungen aufgerufen, z.B. beim Einfügen, Ändern und Löschen von Daten
- Bei UPDATE und INSERT kann mit NEW.* auf die neuen Daten zugegriffen werden
- Bei UPDATE und DELETE kann mit OLD.* auf die alten Daten zugegriffen werden (Beachte dass UPDATE neue und alte Daten enthält)

Beispiel

```
CREATE TRIGGER employee_beforeinsert
BEFORE INSERT ON emp_employee
FOR EACH ROW
BEGIN
-- … Methodenrumpf
END
```

- Der Methodenrumpf wird bei jedem Einfügen in die Employee Datenbank aufgerufen
- Hier kann wie bei Functions und Stored Procedures Geschäftslogik ausgeführt werden

Datenbankseitige Logik

Vorteile

- Einschränkung von Rechten vereinfacht
  - Rechte können auf Stored Procedures eingeschränkt werden
  - Durch Funktionen können Datenwerte gefiltert werden
- Bessere Performance
  - Datenbanken sind auf die Nutzung von Stored Procedures, Functions oder Triggern optimiert
  - Nur wirkliche Ergebnisse werden dem Client übertragen, keine Zwischenergebnisse
- Zentrale Anwendungslogik auf der Datenbank, welche in verschiedenen Anwendungen genutzt wird

Nachteile

- Unterschiedliche Implementierungen jedes DBMS
  - Muss auf DBMS angepasst werden
  - Komplex, aufwendig und daher teuer
- Entwicklungsteam benötigen sowohl tiefes SQL Verständnis als auch Verständnis der Anwendungssprache (zb Java)
  - Experten in beiden Gebieten sind schwierig zu finden
- Die Verwendung von Triggern kann schnell zu komplexen und unübersichtlichen Situationen führen, da Trigger weitere Trigger auslösen können

> *In der Praxis werden Stored Procedures, Funktionen und Trigger daher wenig verwendet*

Transaktionen

- Zusammenhängende SQL-Befehle ausführen
- Beispielsweise Banküberweisung
- Kontostand des Senders muss reduziert werden
- Gleichzeitig auch Kontostand des Empfängers
- Dieses gleichzeitige Ausführen nennt sich Transaktion
- Sollte während der Überweisung etwas schief gehen werden alle Änderungen rückgängig gemacht

Beispiel

```
UPDATE konto
SET kontostand = kontostand - 400
WHERE kontonr = 4;
UPDATE konto
SET kontostand = kontostand + 400
WHERE kontonr = 7;
```

- Standardmäßig arbeiten DBMS meistens mit AUTO COMMIT
- Hier wird jeder einzelne Befehl direkt auf der Datenbank ausgeführt
- Wird AUTO COMMIT deaktiviert können Transaktionen mit dem Befehl COMMIT abgeschickt werden

Beispiel

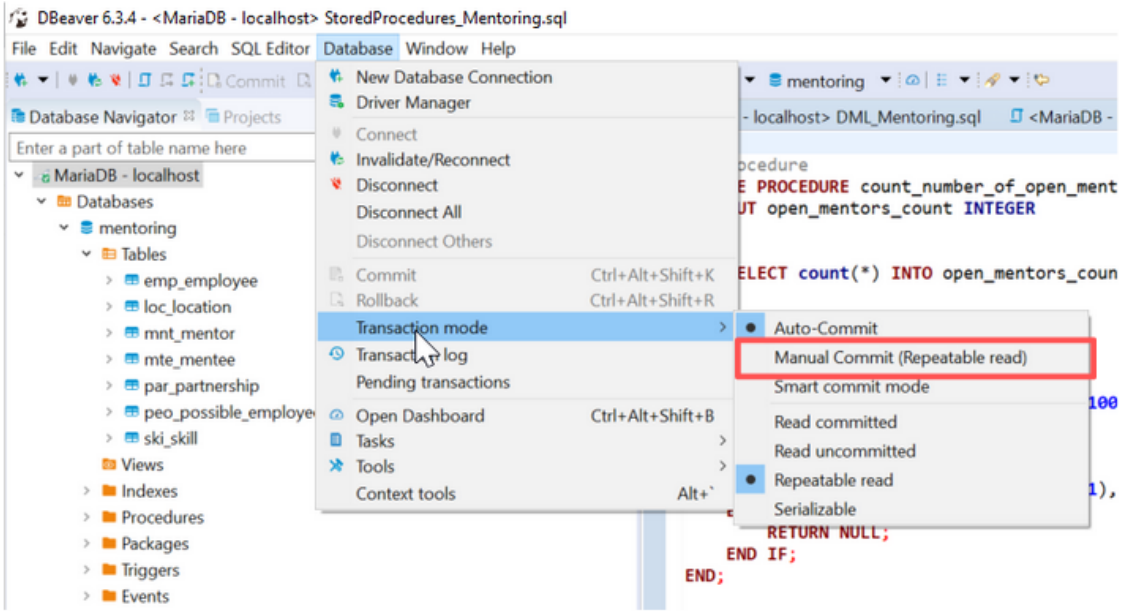
```
UPDATE konto
SET kontostand = kontostand - 400
WHERE kontonr = 4;
UPDATE konto
SET kontostand = kontostand + 400
WHERE kontonr = 7;
COMMIT;
```

> *Beide Befehle werden erst beim Aufruf von `COMMIT` auf der Datenbank ausgeführt*

Mit `ROLLBACK` wird die Transaktion abgebrochen.

Transaktionen in DBeaver

Um Transaktionen in DBeaver zu verwenden muss der Transaktionsmodus auf “Manual Commit (Repeatable read)” gestellt werden.



ACID

Bei der Ausführung von Transaktionen werden die ACID Eigenschaften erfüllt

Atomicity (Atomar): Transaktion wird entweder komplett oder gar nicht durchgeführt

Consistency (Konsistenz): Nach einer Transaktion befinden sich alle Daten in einem konsistenten Zustand

Isolation (Isolation): Gleichzeitig ausgeführte Transaktionen beeinflussen sich nicht gegenseitig

Durability (Dauerhaft): Änderungen von Transaktionen verbleiben dauerhaft in der Datenbank

Views

- Komplexe oder wiederholende Abfragen können in Views überführt werden
- Views virtuelle Tabellen dar
- Views werden durch SELECT Befehle definiert
- Views können wie normale Tabellen verwendet
- Mit DROP VIEW wird eine View wieder gelöscht

Syntax

```
CREATE VIEW <Viewname> AS <Selectabfrage>;
```

Beispiel

```
CREATE VIEW employee_projects AS
SELECT * FROM emp_employee
LEFT OUTER JOIN pro_project
ON emp_pro_id = pro_id;
```

Verwendung

```
SELECT * FROM employee_projects;
```

Skripte

```
CREATE ROLE sales;
DROP ROLE sales;

CREATE ROLE emmy WITH INHERIT LOGIN PASSWORD '1234' IN ROLE sales;

-- login with emmy

GRANT ALL ON ALL TABLES IN SCHEMA company TO sales;

DROP ROLE emmy;


-- stored procedure

DROP PROCEDURE IF EXISTS insert_data;
CREATE PROCEDURE insert_data(
    IN col_name varchar,
    IN pro_name varchar
)
LANGUAGE SQL
AS $$
INSERT INTO fao_favorite_color(fao_color_name) VALUES (col_name);
INSERT INTO pro_project(pro_name) VALUES (pro_name);
$$;

CALL insert_data('pink', 'candy_shop');


CREATE FUNCTION anonymize(name varchar(100))
RETURNS varchar(4)
AS $$
BEGIN
    IF name IS NOT NULL THEN
        RETURN CONCAT(SUBSTRING(name,1,1), '***');
    ELSE
        RETURN NULL;
    END IF;
END;
$$
LANGUAGE plpgsql;

SELECT anonymize(emp_email)
FROM emp_employee;


CREATE VIEW employee_projects AS
```

```sql
SELECT * FROM emp_employee
LEFT OUTER JOIN pro_project
ON emp_pro_id = pro_id;

SELECT * FROM employee_projects;
```

Java Persistence API (JPA)

- Object Relational Mapping (ORM) zwischen der Anwendung und der Datenbank
- Klassen und Attribute erhalten Annotationen, um Tabellen mit der Applikation zu verknuepfen

Grundlegende Verwendung

```java
@Entity
@Table(name = "favorite_number") // diese Klasse wird der Tabelle favorite_number zugeordnet
public class FavoriteNumber {

   @Id // definiert das Attribut als Typ id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   @Column // definiert das Mapping einer Spalte zu einem Attribut
   private Integer number

   // ...
}
```

> *Siehe Beispielprojekt im Ordner* `jpa`

Dokumentation Eclipse Link

Implementierungen

Es gibt verschiedene Implementierungen, wie

- Eclipse Link - Referenzimplementierung, in Jakarta EE enthalten
- Spring
- Hibernate

NoSQL

Relationale Datenbanken

- Beherrschten Markt für langen Zeitraum
- Relationenmodell nach Codd
- SQL als Datenbanksprache
- Transaktionsmodell (ACID)
- Im Normalfall ein zentraler DB-Server

NoSQL

- Oft als “Not only” SQL bezeichnet [2]
- Bezeichnung von schemafreien Datenbanken
- Datenbanksprache nicht standardisiert
- Horizontale Skalierbarkeit durch verteilte Datenbanken
- Schwache Garantie von Datenkonsistenz - BASE (Basically Available, Soft State, Eventual consistency) statt ACID

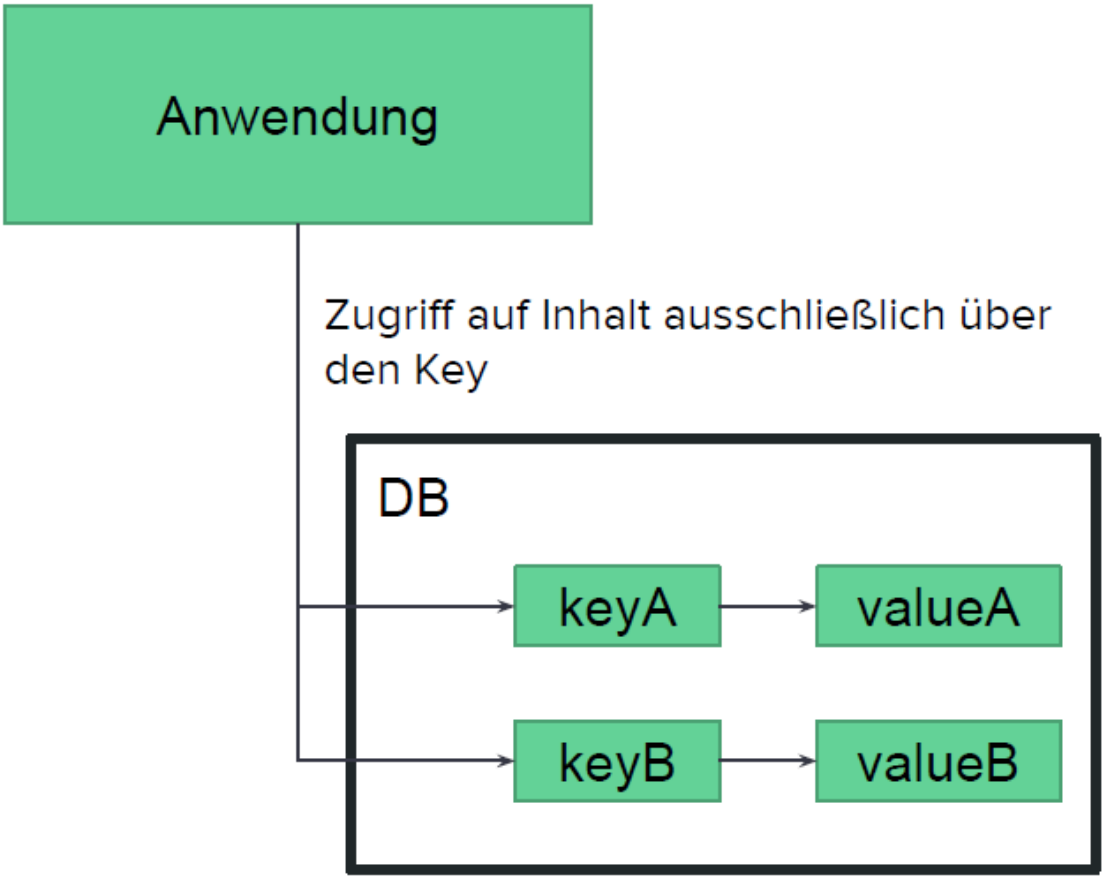
Quellen: Martin Fowler - NoSQL Definition, Grundlagen CAP Theorem

Key-Value Stores

- Daten werden ausschließlich in Key (Schlüssel) und Value (Inhalt) Paaren gespeichert
- Strukturlose Value - für die DB nur effizient zu speichernde Bits und Bytes
- Abfrage nur über Key möglich
- Nutzung der Datenbank obliegt der Anwendung
- Paare werden oft mit einer Lebensdauer ausgestattet, nach welcher diese gelöscht werden

Bekannte Datenbanken

Redis, Riak, Memcached



Dokumentenorientierte Datenbanken

- Speicherung von zusammengehörenden Daten in Dokumenten
- Eindeutiger Schlüssel für Dokument

- Dokumente können sowohl strukturierte Daten (zB. JSON oder XML), als auch unstrukturierte Daten enthalten
- In der Praxis bestehen strukturierte Dokumente aus Key-Value Paaren welche wiederum selbst strukturiert werden

Wichtig

- Es gibt keine Vorgabe zur Struktur
- Es können jederzeit neue Felder zu Dokumenten hinzugefügt werden

Beispiel

```
{
  mentorId: 4711
  vorname: "Jürgen",
  nachname: "Glas",
  mentees: [
    { vorname: "Gustav", nachname: "Anders"},
    { vorname: "Petra", nachname: "Rad"}
  ]
}
```

Bekannte Datenbanken

MongoDB, CouchDB, BaseX, eXist, HCL Notes, OrientDB, Apache Jackrabbit

Wide-Column Store

- Speicherung von Datensätzen mit flexibler Anzahl an Spalten
- Datensätze können unterschiedliche Spalten haben
- 2-Dimensionale Key-Value Stores

Bekannte Datenbanken

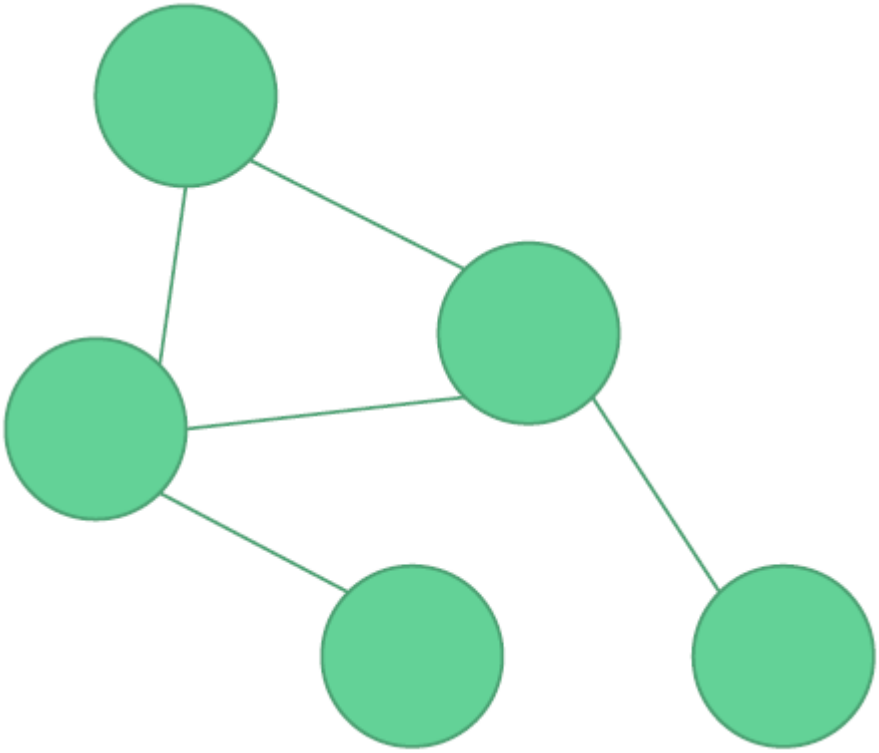
Cassandra, HBase

Graphdatenbanken

- Fokus auf Vernetzung von und Beziehungen von Objekten
- Speicherung als Knoten (Objekte) und Kanten (Beziehungen)
- Auswertung von Beziehungen und Navigation durch diese

Bekannte Datenbanken

Neo4j, OrientDB



Indexierung

- Optimierung von Datenabfragen
- KEY und INDEX sind gleichbedeutend
- PRIMARY KEY ist ein Index
    - im Regelfall mit AUTO_INCREMENT befüllt
    - immer einzigartig - jeder Eintrag wird eindeutig identifiziert
    - niemals NULL
    - nicht jede Tabelle benötigt einen primary key
- `EXPLAIN <Abfrage>` für Analyse von SELECT Abfragen: sinnvolle Indizes koennen hiermit identifizieren werden

Beispiel

ID	First_Name	Last_Name	Position	Home_Address	Home_Phone
1	Mustapha	Mond	Chief Executive Officer	692 Promiscuous Plaza	326-555-3492
2	Henry	Foster	Store Manager	314 Savage Circle	326-555-3847
3	Bernard	Marx	Cashier	1240 Ambient Avenue	326-555-8456
4	Lenina	Crowne	Cashier	281 Bumblepuppy Boulevard	328-555-2349
5	Fanny	Crowne	Restocker	1023 Bokanovsky Lane	326-555-6329
6	Helmholtz	Watson	Janitor	944 Soma Court	329-555-2478

> *Für Abfrage nach Nachnamen müssen alle Einträge durchlaufen werden*

`CREATE INDEX Last_Name` erstellt einen zusätzlichen Index, welcher alle Nachnamen, in einer separaten Tabelle sortiert speichert

> *Nachnamen werden schneller gefunden*

Last_Name	ID
Crowne	4
Crowne	5
Foster	2
Marx	3
Mond	1
Watson	6

Quelle

Verwendung

Create Index

- Einfacher Index für eine Spalte
- Einträge werden sortiert

Syntax

```
CREATE INDEX <Indexname> ON
<Tabellenname>(<Spaltenname>)
```

Unique Index

- Spaltenwert muss einzigartig sein
- Kann NULL sein! (NULL ist zu nichts gleich, auch nicht zu NULL)

Syntax

```
CREATE UNIQUE INDEX <Indexname> ON
<Tabellenname>(<Spaltenname>)
```

Verteilte Datenbanksysteme

Skalierung

Vertikale Skalierung

- Aufrüstung des DB-Servers
- Für RDBMS möglich Horizontale Skalierung
- Aufteilung der Daten auf verschiedene Server (Nodes)
- Lastenaufteilung
- Oft nicht für RDBMS möglich, da ACID nicht eingehalten werden kann
- Möglich für NoSQL Datenbanken

Failover-Cluster

- Daten werden auf Backup-Server gespiegelt
- Bei Ausfall des Hauptservers wird der Failover-Server verwendet

Replikation

Redundante Verteilung der Daten auf verschiedene Server

- Erhöhung der Performance durch Aufteilung der Zugriffe auf versch. Nodes
- Load Balancer reguliert Zugriffe auf Servernetz

Master-Slave-Replikation

- Lesezugriffe über alle Nodes
- Schreibzugriffe (Änderungen) nur über Master-Node
- Bei Ausfall des Masters, wird ein Slave zum Master

Master-Master-Replikation

- Alle Nodes haben Lese- und Schreibzugriff

Sharding

- Verteilung des Datenbestands nach bestimmten Kriterien auf verschiedene Knoten
- Beispielsweise alle Mitarbeiter mit den Nachnamen A-F auf Knoten 1, alle mit G-L Knoten 2, usw..
- Wichtig:
    - Kategorisierung muss auf Anwendungsfall und Abfrageoperationen abgestimmt sein
    - Kombinationen von Datensätzen über mehrere Nodes ist sonst aufwändig

CAP-Theorem

Nach Eric Brewer können in verteilten DBMS maximal zwei, jedoch nie drei der folgenden Eigenschaften garantiert werden:

Konsistenz: alle Knoten liefern identische Ergebnisse

Verfügbarkeit: auf jedem Knoten können Schreib- oder Lesezugriffe durchgeführt werden

Ausfalltoleranz: System kann bei Ausfällen weiterverwendet werden. Bei verteilten DBMS immer notwendig

Quelle

BASE

- Gegensatz zu den ACID Eigenschaften (starke Konsistenz)
- Einsatz in verteilten DBMS, da Verfügbarkeit höher gewichtet ist als Konsistenz
- BASE
    - Basically Available
    - Soft state
    - Eventual consistency

> *Eventual Consistency bedeutet, dass Schreibaktionen nicht unmittelbar auf allen Knoten durchgeführt werden müssen, sondern dass Aktualisierungen nach und nach im Knotennetz verteilt werden können.*

Literatur

- Alan Beaulieu: Learning SQL (2nd Edition). O'Reilly, 2009
- Lynn Beighley: Head First SQL, 2007. O'Reilly, 2007