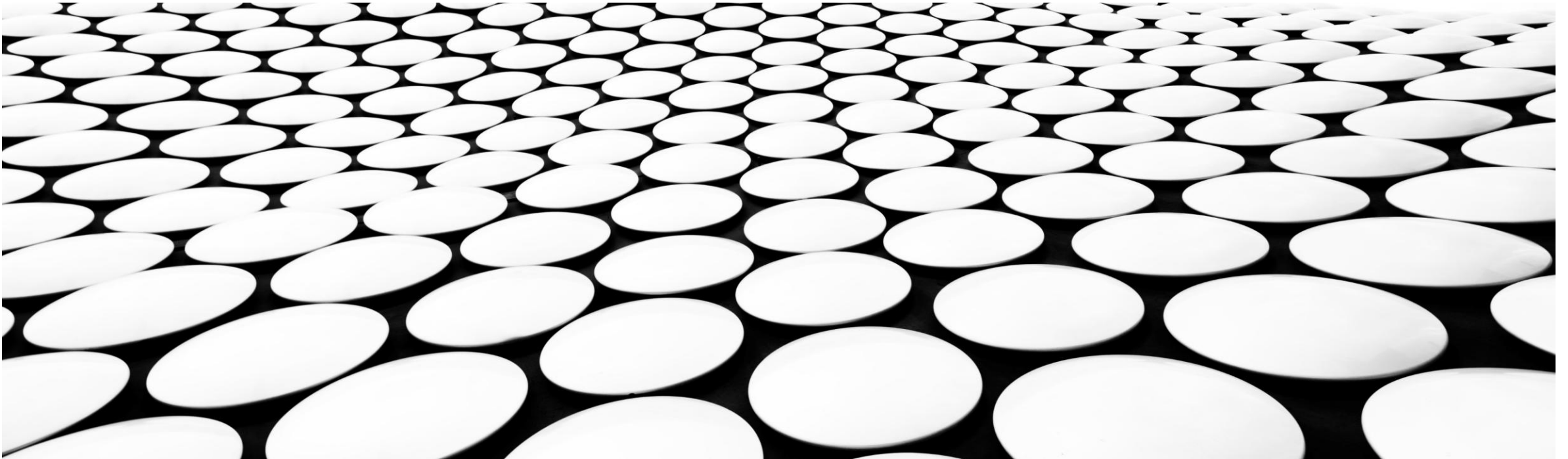


POSTGRESQL

FÜR FORTGESCHRITTENE



ARBEITSUMGEBUNG

- Verzeichnis C:\POSTGRESQL
- books_sik.csv
- customers.csv
- customer_sik.csv
- unibook.csv
- tester.csv
- bookstore_backup
- Ordner dvdrental
- sample.sql
- data_books_union.sql

Bookstore-DB genauer ansehen

Visual Studio Code installiert

SELECT ... | SELECT DISTINCT

Varianten:

```
SELECT 'Hallo'
```

```
SELECT ('Hallo Welt')
```

```
SELECT 'Hallo' || 'Welt'
```

```
SELECT 4/2
```

```
SELECT * FROM books
```

```
SELECT title FROM books
```

```
SELECT title, issued FROM books
```

```
SELECT DISTINCT amount FROM orders ORDER BY amount
```

```
SELECT version();
```

```
SELECT CURRENT_USER;
```

```
SELECT NOW();
```

```
SELECT CURRENT_TIMESTAMP;
```

```
SELECT orders FROM orders;
```

SELECT CONCAT()

– quick and dirty

```
SELECT (firstname, lastname) FROM customers
```

– mit Pipe-Verbindung

```
SELECT firstname || ", " || lastname FROM customers -- KEINE Anführungszeichen !!!
```

– MIT EINFACHEM Hochkomma !!!

```
SELECT firstname || ', ' || lastname FROM customers
```

– Konkatenierte Spalten (firstname & lastname) mit Alias "FullName" versehen

```
SELECT firstname || ', ' || lastname AS "FullName" FROM customers -- MIT Anführungszeichen für ALIAS
```

– mit CONCAT-Funktion

```
SELECT CONCAT(firstname, ', ', lastname) AS "FullName" FROM customers
```

– wie werden NULL-Werte mit CONCAT verarbeitet?

```
SELECT creator, issued FROM books ORDER BY creator
```

```
SELECT creator || ' --- ' || issued FROM books ORDER BY creator NULLS FIRST
```

```
SELECT CONCAT(creator, ' --- ', issued) FROM books ORDER BY creator NULLS LAST
```

SELECT ... CASE

Ausgabe eines Spaltenwertes, der infolge einer Überprüfung wechselt:

```
SELECT firstname, age,
CASE
    WHEN age >= 70 THEN 'Alt'
    WHEN age >= 60 THEN 'Älter'
    WHEN age >= 50 THEN 'BestAger'
    WHEN age >= 30 THEN 'Mittel'
    WHEN age >= 18 THEN 'Jung'
END
FROM customers
ORDER BY age, firstname
```

Nach dem ermittelten Spaltenwert filtern:

```
SELECT firstname, age,
CASE
    WHEN age >= 70 THEN 'Alt'
    WHEN age >= 60 THEN 'Älter'
    WHEN age >= 50 THEN 'BestAger'
    WHEN age >= 30 THEN 'Mittel'
    WHEN age >= 18 THEN 'Jung'
END
FROM customers
WHERE
CASE
    WHEN age >= 70 THEN 'Alt'
    WHEN age >= 60 THEN 'Älter'
    WHEN age >= 50 THEN 'BestAger'
    WHEN age >= 30 THEN 'Mittel'
    WHEN age >= 18 THEN 'Jung'
END = 'Mittel'
ORDER BY age, firstname
```

SELECT ... LIMIT ... OFFSET ...

Die Anzahl der auszugebenden Datensätze kann festgelegt werden durch
LIMIT Anzahl auszugebender Datensätze und
OFFSET Angabe auszulassenden Datensätze

Beispiel:

`SELECT * FROM customers LIMIT 10` → gibt nur 10 Datensätze aus

`SELECT * FROM customers LIMIT 10 OFFSET 5` → gibt 10 Datensätze ab Datensatz 6 aus

Einsatz:

- Bei großen Datenbeständen beschleunigt sich hierdurch die Performance der Abfrage
- Herausfinden von größten / kleinsten Werte einer Spalte in Kombination mit ORDER BY

SELECT ... WHERE

Vergleichsoperatoren: = , < , <= , > , >= , <> , !=

Vergleich mit Teilzeichen:

```
SELECT firstname FROM customers WHERE firstname LIKE 'A%'
```

```
SELECT firstname FROM customers WHERE firstname LIKE 'Anne%'
```

```
SELECT firstname FROM customers WHERE firstname LIKE 'Anne1'
```

ILIKE

Vergleich mit Bereichswerten:

```
SELECT firstname, age FROM customers WHERE age >= 50 AND age <= 80
```

```
SELECT firstname, age FROM customers WHERE age BETWEEN 50 AND 80
```

```
SELECT SUBSTR(lastname,1,1),lastname  
FROM customers  
WHERE SUBSTR(lastname,1,1) BETWEEN 'A' AND 'D'  
ORDER BY SUBSTR(lastname,1,1)
```

SELECT ... WHERE ... IN()

Grundaufbau

```
SELECT spalte FROM tabelle WHERE spalte IN('Suchwert', 'Suchwert2', 'Suchwert3')
```

Beispiel:

Zeige mir alle <lastname>, <firstname> aus [customers], die in <lastname> den Wert 'Albers', 'Bader' oder 'Brand' haben.

```
SELECT lastname, firstname
```

```
FROM customers
```

```
WHERE lastname IN ('Albers', 'Bader', 'Brand')
```

```
ORDER BY lastname, firstname
```

NOT IN

SELECT ... GROUP BY ... HAVING

Grundaufbau:

```
SELECT spalte(n), berechnete Spalten FROM tabelle  
[WHERE ...]  
GROUP BY spalte(n)  
HAVING ...
```

Einsatz:

- GROUP BY notwendig, wenn Spalte mit Aggregatfunktion kombiniert wird
- Alle Nicht-aggregierten Spaltenwerte müssen auch im GROUP BY erscheinen
- HAVING filtert die bereits gruppierten Zeilen nach Bedingungen
- In HAVING-Klausel können auch Aggregatfunktionen benutzt werden

SELECT ... GROUP BY ... HAVING

Aufgabe:

Zeige mit <lastname> aus [customers] und Anzahl gleicher <lastname>, wenn Anzahl gleicher <lastname> > 1

```
SELECT lastname, COUNT(*)  
FROM customers  
WHERE COUNT(*)>1  
GROUP BY lastname  
ORDER BY lastname  
-- funktioniert nicht
```

```
SELECT lastname, COUNT(*)  
FROM customers  
GROUP BY lastname  
HAVING COUNT(*) > 1  
ORDER BY lastname  
-- funktioniert
```

SELECT ... AS ALIAS

Spalten oder berechnete Spalten oder Tabellen können mit einem ALIAS benannt werden.

Grundaufbau:

```
SELECT Spalte AS ALIAS FROM Tabelle AS ALIAS
```

Vorteile:

- Erspart Schreibarbeit, wenn Herkunft der Spalten angegeben werden muss (bei verknüpften Tabellen und gleichen Spaltennamen)
- Macht den Code übersichtlicher und leichter zu erfassen
- Kann in ORDER BY verwendet werden – insbesondere bei komplexen Anweisungen vereinfacht das den Code erheblich

Achtung:

- Reihenfolge der Abarbeitung der Anweisung entscheidet, ob ALIAS berücksichtigt wird oder nicht.
In SELECT-Statement benannte Spalte kann in WHERE-Anweisung z.B. NICHT benutzt werden

SELECT ... AS ALIAS

```
SELECT lastname AS nachname
FROM customers AS c
ORDER BY nachname
-- funktioniert
```

```
SELECT lastname AS nachname
FROM customers AS c
WHERE c.nachname = 'Albers'
ORDER BY nachname
-- Spaltenalias in WHERE funktioniert nicht
-- Tabellenalias funktioniert
```

```
SELECT lastname AS nachname
FROM customers AS c
GROUP BY lastname
HAVING c.nachname = 'Albers'
ORDER BY nachname
-- Spaltenalias funktioniert nicht
```

```
SELECT lastname AS nachname
FROM customers AS c
GROUP BY lastname
HAVING c.lastname = 'Albers'
ORDER BY nachname
-- Tabellenalias funktioniert im HAVING !
```

SELECT ... DATE_PART()

Grundaufbau:

```
DATE_PART('Zeiteinheit', Spalte)
```

Einsetzbar bei:

- Ausgabe zusätzlicher informativer Spalten
- als Bedingung in WHERE

Beispiel:

Zeige mir alle <title> und Jahreszahlen der Datumsspalte <issued> aus der Tabelle [books].

```
SELECT title, DATE_PART('year', issued)
FROM books
```

SELECT ... WHERE DATE_PART()

Aufgabe:

Zeige mir alle <title>, <issued> aus [books], wenn in <issued> der Monat 6 angegeben ist.

```
SELECT issued, title  
FROM books  
WHERE DATE_PART('month', issued) = 6  
ORDER BY issued, title
```

SELECT ... GROUP BY DATE_PART()

Aufgabe: Zeige mir die Jahre der Veröffentlichungen der erfassten Bücher ([books] – Spalte <issued>) mit Anzahl der Veröffentlichungen und abgestuftem Kommentar entsprechend der Anzahl an.

Regel für abgestuftem Kommentar:

<= 1000 → Schlechte Zeiten

> 1000 → Bessere Zeiten

> 2000 → Gute Zeiten

> 3000 → Spitzen Zeiten

```
SELECT DATE_PART('year', issued) AS jahr, COUNT(*),  
       CASE  
         WHEN COUNT(*) > 3000 THEN 'Spitzen Zeiten'  
         WHEN COUNT(*) > 2000 THEN 'Gute Zeiten'  
         WHEN COUNT(*) > 1000 THEN 'Bessere Zeiten'  
       ELSE  
         'Schlechte Zeiten'  
       END  
FROM books  
GROUP BY DATE_PART('year', issued)  
ORDER BY jahr
```



SUBQUERIES

DIE ABFRAGE IN DER ABFRAGE

SELECT <SPALTE>, (SUBQUERY) ...

In eine Select-Abfrage können auch Unterabfragen eingebaut werden, die Daten z.B. aus einer anderen Tabelle holen und in einer zusätzlichen Spalte ausgeben.

- Beispiel: Bücher sind einem Gebiet zugeordnet – identifiziert über <subject_id>

```
SELECT subject_id, SUBSTR(title, 1, 10),  
      (SELECT title FROM books_subjects  
       WHERE books_subject.id = books. subject_id)  
FROM books
```

- ACHTUNG: ES KANN IMMER NUR 1 SPALTENWERT MIT EINER ZEILE ÜBER DIE SUBQUERY ZURÜCKGEGEBEN WERDEN!!!

(Trick: Mehrere Spalten konkatinieren oder mehrere Subqueries– geht aber zulasten der Performance)

WO KÖNNEN SUBQUERIES GENUTZT WERDEN?

```
SELECT lastname, SUBQUERY  
FROM SUBQUERY  
WHERE lastname = SUBQUERY  
GROUP BY lastname  
HAVING lastname = SUBQUERY  
ORDER BY SUBQUERY
```

Mit Subqueries können auch zusätzliche Vergleichsoperatoren genutzt werden:

- ANY
- ALL
- EXISTS

SUBQUERIES IN SELECT-KLAUSEL

- Ein Subselect ist quasi eine Query in der Query:
- Grundaufbau für Ausgabe einer zusätzlichen Spalte mit Hilfe einer Subquery (kann andere aber auch dieselbe Tabelle sein):

SELECT

(SELECT ...) [AS something]

FROM tabelle

- Die innere Query wird einmal pro Datensatz der äußeren Tabelle ausgeführt.
- Hierdurch können Daten aus mehreren Tabellen genutzt werden und komplexere Abfragen ausgeführt werden.

SUBQUERIES IN SELECT-KLAUSEL

```
SELECT c.id, c.lastname,  
       (SELECT COUNT(o2.customer_id)  
        FROM orders AS o2  
        WHERE c.id = o2.customer_id) AS bestellungen  
FROM customers AS c  
ORDER BY bestellungen DESC, c.lastname
```

Abfrage liefert Anzahl der Bestellungen aus Tabelle [orders] der Kunden aus Tabelle [customers]

SUBQUERIES IN WHERE-KLAUSEL & ANY

```
SELECT c.id, c.lastname,  
       (SELECT COUNT(o2.customer_id)  
        FROM orders AS o2  
        WHERE c.id = o2.customer_id) AS bestellungen  
FROM customers AS c  
WHERE c.id = ANY  
      (SELECT o.customer_id  
       FROM orders AS o  
       GROUP BY o.customer_id  
       HAVING COUNT(customer_id) > 10)  
GROUP BY c.id, c.lastname  
ORDER BY c.lastname;
```

Ausgabe aller Kunden, die mehr als 10 Bestellungen getätigt haben. Es werden durch das Subselect in der WHERE-Klausel eine Liste von customer_id erstellt, in der die jeweilige customer_id der äußeren Abfragen vorkommen muss.

SUBQUERIES IN WHERE-KLAUSEL & EXISTS

Aufgabe:

Zeige alle Kunden, die bestellt haben.

```
SELECT DISTINCT (c.lastname || ', ' || c.firstname) AS ln
  FROM customers AS c
   WHERE EXISTS
      (
        SELECT o.customer_id
          FROM orders AS o
         WHERE o.customer_id = c.id
      )
 ORDER BY ln;
```

SUBQUERIES IN HAVING-KLAUSEL

```
SELECT c.id, c.lastname,  
       (SELECT COUNT(o2.customer_id) FROM orders AS o2  
        WHERE c.id = o2.customer_id) AS bestellungen  
FROM customers AS c  
GROUP BY c.id, c.lastname  
HAVING  
       (SELECT COUNT(o2.customer_id) FROM orders AS o2  
        WHERE c.id = o2.customer_id) > 10  
ORDER BY c.lastname;
```

SUBQUERIES IN WHERE-KLAUSEL & ALL

```
SELECT c.id, c.lastname
FROM customers AS c
WHERE c.id = ALL
      (SELECT o.customer_id
       FROM orders AS o
       GROUP BY o.customer_id
       HAVING COUNT(customer_id) > 10);
```

-- Keine Ausgabe, weil c.id **nicht mit ALLEN**
Datensätzen übereinstimmt

```
SELECT c.id, c.lastname
FROM customers AS c
WHERE c.id = 47 and c.id = ALL
      (SELECT o.customer_id
       FROM orders AS o
       WHERE customer_id = 47
       GROUP BY o.customer_id
       HAVING COUNT(customer_id) > 10);
```

-- Übereinstimmung der gefilterten c.id
mit allen Subquery-Ausgabewerten – daher Ausgabe
eines Datensatzes

SUBQUERIES IN ORDER BY

Aufgabe:

Ausgabe von <lastname>, <firstname>, <anzahl_bestellungen>, <kundenkategorie>
aus [customers] und [orders]
absteigend sortiert nach <anzahl_bestellungen>

Kriterien für Kundenkategorie:

Bestellungen	kundenkategorie
0	KEIN KUNDE
<= 3	Gelegenheitskunde
<= 5	Guter Kunde
< 10	Zum Topkunden machen
Ab	TOPKUNDE

SUBQUERIES IN ORDER BY

```
SELECT c.lastname, c.firstname,  
       (SELECT COUNT(customer_id) FROM orders AS o2 WHERE c.id = o2.customer_id) AS  
bestellungen,  
       CASE  
         WHEN (SELECT COUNT(id) FROM orders AS o2  
              WHERE c.id = o2.customer_id) = 0 THEN 'KEIN KUNDE'  
         WHEN (SELECT COUNT(id) FROM orders AS o2  
              WHERE c.id = o2.customer_id) <= 3 THEN 'Gelegenheitskunde'  
         WHEN (SELECT COUNT(id) FROM orders AS o2  
              WHERE c.id = o2.customer_id) <= 5 THEN 'Guter Kunde'  
         WHEN (SELECT COUNT(id) FROM orders AS o2  
              WHERE c.id = o2.customer_id) < 10 THEN 'Zum Topkunden machen'  
       ELSE  
         'TOPKUNDE'  
       END  
FROM customers AS c  
ORDER BY  
       (SELECT COUNT(id) FROM orders AS o2  
        WHERE c.id = o2.customer_id) DESC;
```

ORDER BY bestellungen

SUBQUERIES IN FROM-KLAUSEL

Aufgabe:

Zeige mir <lastname>, <firstname>, <age>, <diff> aus [customers] für die Kunden an, die mehr als 60 Jahre Differenz zum maximalen Kundenalter haben absteigend nach diff sortiert, dann nach Namen aufsteigend

```
SELECT lastname, firstname, age, diff
```

```
FROM
```

```
    (SELECT lastname, firstname, age,
```

```
        (SELECT MAX(age) FROM customers)-age AS diff
```

```
        FROM customers) AS sub
```

```
WHERE diff>60
```

```
ORDER BY diff DESC, lastname ASC, firstname ASC
```

BISHERIGE (modifizierte) Lösung funktioniert NICHT, da äußere Abfrage keinen Zugriff auf innere Abfrage hat und Aggregatfunktion nicht in WHERE-Klausel benutzt werden darf

```
SELECT lastname, firstname, age,  
       (SELECT MAX(age) FROM customers) AS maxage,  
       (SELECT MAX(age) FROM customers)-age AS diff  
FROM customers  
   WHERE diff>60  
ORDER BY diff DESC, lastname ASC, firstname ASC
```

(HAVING funktioniert auch nicht)
Wiederholung der Subquery in ORDER BY funktioniert
!!! Performanceverlust !!!

Äußere Abfrage hat Zugriff auf innere Abfrage – ALIAS kann genutzt werden.

SUBQUERIES IN FROM-KLAUSEL

Aufgabe:

Zeige mir alle unterschiedlichen <creator> aus [books] an, die für mehr als 10 unterschiedliche Kategorien <subject_id> veröffentlicht haben, absteigend nach <anzsubject> sortiert, danach nach <creator> aufsteigend 'Anonymus', 'Various' und 'Unknown' sollen ausgeschlossen werden.

```
SELECT DISTINCT(creator), anzsubject
FROM
(
    SELECT creator, COUNT(DISTINCT subject_id) AS anzsubject
    FROM books
    GROUP BY creator
    HAVING creator NOT IN ('Anonymous', 'Various', 'Unknown')
) AS sub
WHERE anzsubject > 10
ORDER BY anzsubject DESC, creator ASC
```

SUBQUERIES IN FROM-KLAUSEL

Aufgabe:

Zeige alle <creator> und deren Summe der Downloads mit Spalte <sumdown> für die gilt, dass die Summe der Downloads in der Spalte <downloads> größer 10000 und kleiner 15000 betragen.
Es soll absteigend nach der Summe der Downloads sortiert werden.

```
SELECT creator, sumdown
FROM
    (SELECT creator, SUM(downloads) AS sumdown
     FROM books
     GROUP BY creator)
WHERE sumdown > 10000 AND sumdown < 15000
ORDER BY sumdown DESC
```

SUBQUERIES IN FROM-KLAUSEL

Aufgabe:

Ausgabe einer Liste der <creator> mit durchschnittlichen Downloadzahlen >80 und <90, absteigend nach Durchschnittszahl, dann aufsteigend nach creator sortiert - ohne 'Anonymous', 'Various', 'Unknown'

```
SELECT creator, ROUND(avgx,0) as durchschnitt
FROM
(
    SELECT creator, AVG(downloads) AS avgx
    FROM books
    GROUP BY creator
) AS s
GROUP BY creator, avgx
HAVING creator NOT IN('Anonymous', 'Various', 'Unknown')
AND avgx >80 AND avgx <90
ORDER BY avgx DESC, creator ASC
```

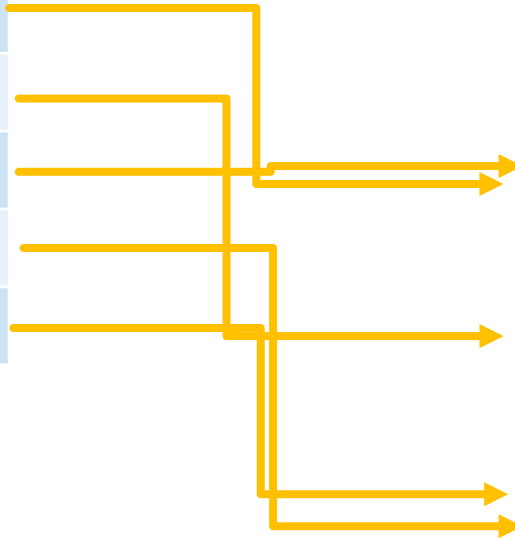


JOINS

VERKNÜPFEN VON MEHREREN TABELLEN

DATENABFRAGE MIT SUBSELECT

id	title	...	subject_id
1	Buch1		8
2	Buch2		17
3	Buch3		8
4	Buch4		20
5	Buch5		20



id	title
1	"History, Modern – 19th century – Juvenile literature – Periodicals"
...	
8	Poetry
...	
17	Art - Korea
...	
20	Essays

id	buch	...	kategorie
1	Buch1		Poetry
2	Buch2		Art - Korea
3	Buch3		Poetry
4	Buch4		Essays
5	Buch5		Essays

```
SELECT A.id, A.title AS titel,  
       (SELECT title AS kategorie FROM books_subjects AS B  
        WHERE A.subject_id = B.id)  
  
FROM  
books AS A
```

Zeilenweise Ausführung der Subselect, um Kategorie in 2. Tabelle zu finden. ALIAS, um Schreibarbeit zu reduzieren und Verwechslung von Spalten mit gleichen Namen in beiden Tabellen zu vermeiden.

DATENABFRAGE MIT GROUP BY

id	title	...	subject_id
1	Buch1		8
2	Buch2		17
3	Buch3		8
4	Buch4		20
5	Buch5		20

id	title
1	"History, Modern -- 19th century -- Juvenile literature -- Periodicals"
...	
8	Poetry
...	
17	Art - Korea
...	
20	Essays

```
SELECT A.id, A.title AS buchtitel, B.title AS kategorie
FROM
books AS A, books_subjects AS B
WHERE A.subject_id = B.id
group by A.id, A.title, B.title
```

id	buch	...	kategorie
1	Buch1		Poetry
2	Buch2		Art - Korea
3	Buch3		Poetry
4	Buch4		Essays
5	Buch5		Essays

Erst gruppieren, dann Auslesen der zugehörigen Daten aus der 2. Tabelle.

DATENABFRAGE MIT CROSS-JOIN

id	title
1	"History, Modern – 19th century – Juvenile literature – Periodicals"
...	
8	Poetry
...	
17	Art - Korea
...	
20	Essays

id	title	...	subject_id
1	Buch1		8
2	Buch2		17
3	Buch3		8
4	Buch4		20
5	Buch5		20

```
SELECT A.title AS kategorie, COUNT(B.id) as anzahl1
FROM
books_subjects AS A, books AS B
WHERE A.id = B.subject_id
group by A.title
```

id	kategorie	...	anzahl
1	„History, ...“		0
8	Poetry		2
17	Art - Korea		1
20	Essays		2

GROUP BY ermöglicht Kombination aus Spalten & Aggregatfunktionen im SELECT.

JOIN-ARTEN

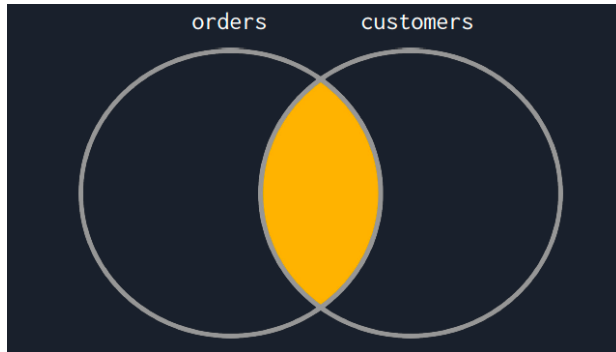
PostgreSQL ist ein RELATIONALES DATENBANKMANAGEMENTSYSTEM.

M.a.W.: Die Tabellen in einer Datenbank haben Beziehungen untereinander.

Die Beziehungen können unterschiedlich hergestellt / genutzt werden – das geschieht über die JOINS.

- ES GIBT VERSCHIEDENE TYPEN VON JOINS
 - CROSS JOIN
 - LEFT JOIN / RIGHT JOIN
 - INNER JOIN
 - FULL JOIN
 - UNION

INNER JOIN



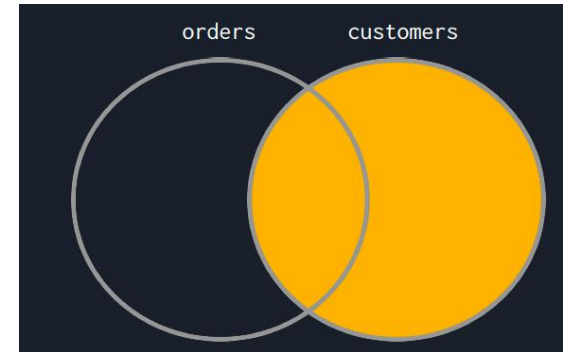
Nur Daten, die in beiden Tabellen vorkommen.

LEFT JOIN



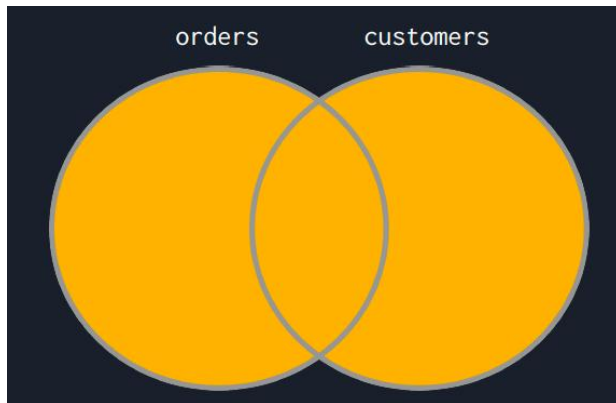
Alle Daten aus orders und die passenden aus customers

RIGHT JOIN



Alle Daten aus customers und die passenden aus orders

FULL JOIN



Daten aus beiden Tabellen

CROSS JOIN

Jeder DS aus orders wird mit jedem DS aus customers kombiniert.

CROSS JOIN

registration

	id integer	name character varying (100)
1	1	Anna
2	2	Max
3	3	Moritz

login

	id integer	name character varying (100)
1	1	Anna
2	2	Max
3	3	Silvia
4	4	Herbert
5	5	Peter

SELECT * FROM registration CROSS JOIN login

	id integer	name character varying (100)	id integer	name character varying (100)
1	1	Anna	1	Anna
2	1	Anna	2	Max
3	1	Anna	3	Silvia
4	1	Anna	4	Herbert
5	1	Anna	5	Peter
6	2	Max	1	Anna
7	2	Max	2	Max
8	2	Max	3	Silvia
9	2	Max	4	Herbert
10	2	Max	5	Peter
11	3	Moritz	1	Anna
12	3	Moritz	2	Max
Total rows: 15 of 15 Query complete 00:00:00.160				

INNER JOIN

```
SELECT * FROM registration
INNER JOIN login
ON registration.name = login.name;
```

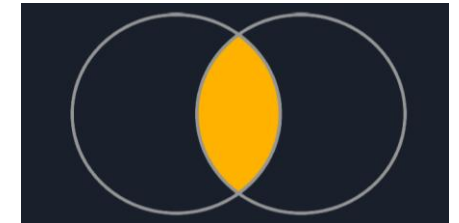
registration

	id integer	name character varying (100)
1	1	Anna
2	2	Max
3	3	Moritz

login

	id integer	name character varying (100)
1	1	Anna
2	2	Max
3	3	Silvia
4	4	Herbert
5	5	Peter

	id integer	name character varying (100)	id integer	name character varying (100)
1	1	Anna	1	Anna
2	2	Max	2	Max



LEFT JOIN

registration

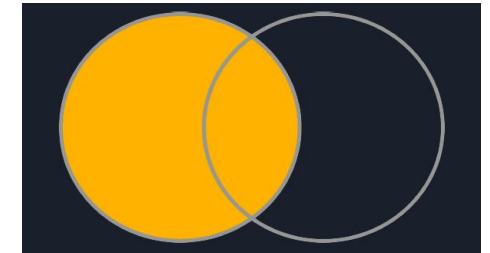
	id integer	name character varying (100)
1	1	Anna
2	2	Max
3	3	Moritz

login

	id integer	name character varying (100)
1	1	Anna
2	2	Max
3	3	Silvia
4	4	Herbert
5	5	Peter

```
SELECT * FROM registration LEFT JOIN login  
ON registration.name = login.name;
```

	id integer	name character varying (100)	id integer	name character varying (100)
1	1	Anna	1	Anna
2	2	Max	2	Max
3	3	Moritz	[null]	[null]



RIGHT JOIN

```
SELECT * FROM registration
RIGHT JOIN login
ON registration.name = login.name;
```

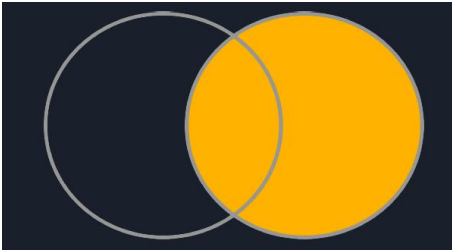
registration

	id integer	name character varying (100)
1	1	Anna
2	2	Max
3	3	Moritz

login

	id integer	name character varying (100)
1	1	Anna
2	2	Max
3	3	Silvia
4	4	Herbert
5	5	Peter

	id integer	name character varying (100)	id integer	name character varying (100)
1	1	Anna	1	Anna
2	2	Max	2	Max
3	[null]	[null]	3	Silvia
4	[null]	[null]	4	Herbert
5	[null]	[null]	5	Peter



FULL JOIN

```
SELECT * FROM registration
FULL OUTER JOIN login
ON registration.name = login.name;
```

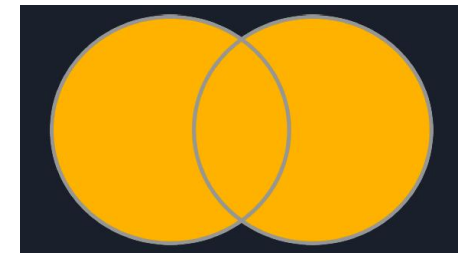
registration

	id integer	name character varying (100)
1	1	Anna
2	2	Max
3	3	Moritz

login

	id integer	name character varying (100)
1	1	Anna
2	2	Max
3	3	Silvia
4	4	Herbert
5	5	Peter

	id integer	name character varying (100)	id integer	name character varying (100)
1	1	Anna	1	Anna
2	2	Max	2	Max
3	3	Moritz	[null]	[null]
4	[null]	[null]	4	Herbert
5	[null]	[null]	3	Silvia
6	[null]	[null]	5	Peter



JOIN-SYNTAX – CROSS JOIN

Jeder DS aus Tabelle1 wird mit jedem DS aus Tabelle2 kombiniert. (Kartesischer Produkt).

Anschließendes Herausfiltern der zusammengehörigen DS aus beiden Tabellen.

```
SELECT <spalte1>, <spalte2> ...  
    FROM [tabelle1], [tabelle2]  
    WHERE [tabelle1].<id> = [tabelle2].<id>
```

ODER

```
SELECT * FROM [tabelle1] CROSS JOIN [tabelle2]  
    WHERE [tabelle1].<id> = [tabelle2].<id>
```

JOIN-SYNTAX – CROSS JOIN

```
SELECT c.lastname, c.firstname, COUNT(*)  
FROM customers AS c, orders AS o  
WHERE c.id = o.customer_id  
GROUP BY c.lastname, c.firstname
```

```
SELECT c.lastname, c.firstname, COUNT(*)  
FROM customers AS c CROSS JOIN orders AS o  
WHERE c.id = o.customer_id  
GROUP BY c.lastname, c.firstname
```

ODER

Kartesisches Produkt:

Tab1: 500 DS

Tab2: 1000 DS → 500.000 DS

JOIN-SYNTAX – LEFT JOIN

```
SELECT c.lastname, c.firstname, COUNT(*)  
FROM customers AS c  
LEFT JOIN orders AS o  
ON c.id = o.customer_id  
GROUP BY c.lastname, c.firstname
```

Wo ist der Fehler?

```
SELECT c.lastname, c.firstname, COUNT(*)  
FROM customers AS c CROSS JOIN orders AS o  
WHERE c.id = o.customer_id  
GROUP BY c.lastname, c.firstname
```

Wo ist der Unterschied?

JOIN-SYNTAX – RIGHT JOIN

```
SELECT c.lastname, c.firstname, COUNT(*)  
FROM customers AS c  
RIGHT JOIN orders AS o  
ON c.id = o.customer_id  
GROUP BY c.lastname, c.firstname
```

Warum werden nur 187 DS ausgegeben?

Warum werden nicht 186 DS ausgegeben?

Wie müsste der Code lauten, um das gleiche Ergebnis zu bekommen wie beim LEFT JOIN?

Wo ist der Fehler?

AUFGABEN ZU JOINS

1. Wie viele Kunden haben noch nichts bestellt?
2. Zu wie vielen Bestellungen fehlen uns die Kundenzuordnungen?
3. Wie viele durch Kundenzuordnungen verifizierte Bestellungen haben wir?

LÖSUNGEN FÜR AUFGABEN ZU JOINS

1. Wie viele Kunden haben noch nichts bestellt?

```
SELECT * FROM orders RIGHT JOIN customers ON orders.customer_id = customers.id  
WHERE timestamp IS NULL  
order by lastname  
--> 14
```

2. Zu wie vielen Bestellungen fehlen uns die Kundenzuordnungen?

```
SELECT * FROM orders LEFT JOIN customers ON orders.customer_id = customers.id  
WHERE lastname IS NULL  
order by lastname  
ODER: SELECT * from orders where customer_id IS NULL or customer_id = 0  
--> 51
```

3. Wie viele durch Kundenzuordnungen verifizierte Bestellungen haben wir?

```
SELECT * FROM orders INNER JOIN customers ON orders.customer_id = customers.id  
→ 1001
```



DATA DEFINITION LANGUAGE – DDL

ANWEISUNGEN, UM TABELLEN ZU DEFINIEREN

DEFINITION

Die **Data Definition Language (DDL)** umfasst alle **Befehle in SQL**, die zur **Erzeugung und Bearbeitung von Tabellen** in der Datenbank genutzt werden.

TABELLEN ERSTELLEN . CREATE TABLE

AUFBAU:

```
CREATE TABLE name (  
    spalte1 datatype,  
    spalte2 datatype  
)
```

DATENTYPEN . DIE WICHTIGSTEN

- VARCHAR(n) - Text mit maximaler Länge
- TEXT - (Große) Textmenge ohne feste Länge
- CHAR(n) - Text mit fester Länge
- INTEGER - Ganzzahlen
- NUMERIC(10,2) - Gerundete Zahlen
- DATE / TIMESTAMP - Datumswerte

ZAHLEN SPEICHERN

- Wir können verschieden viele Bytes pro Zahl verwenden
 - 1 Byte = 8 Bit, d.h. 8 Bit pro Zahl: 0 0 0 0 0 0 0 1
Bei 1 Byte = 8 Bit können wir also $2^8 = 256$ verschiedene
- Zahlen speichern
 - 2 Byte = 16 Bit = 2^{16}
 - 65.536 verschiedene Zahlen
 - 4 Byte = 32 Bit = 2^{32}
 - 4.294.967.296 verschiedene Zahlen
 - 8 Byte = 64 Bit = 2^{64}
 - $1,8 \cdot 10^{19}$ verschiedene Zahlen!

EXAKTE ZAHLEN SPEICHERN

- 2 Byte, $2^{16} = 65.536$ Zahlen:

MySQL & PostgreSQL:

- `SMALLINT`, -32768 bis +32767

Nur MySQL:

- `UNSIGNED SMALLINT`: 0 bis 65535

- 4 Byte, 2^{32} Zahlen:

- PostgreSQL: `INTEGER`

- MySQL: `INT`

- 8 Byte, 2^{64} Zahlen:

- `BIGINT`

KOMMAZAHLEN SPEICHERN

- Exakte Kommazahlen:
 - `DECIMAL (NUMERIC)`: Hier können wir selbst bestimmen, wie viele Ziffern vor dem Komma oder nach dem Komma unterstützt werden sollen
 - Beispiel: 123,45€
 - Anzahl Ziffern (precision): 5
 - Anzahl Ziffern hinter dem Komma (scale): 2
=> `DECIMAL(5, 2)`
 - Exakte Kommazahl (`DECIMAL,...`): Business-Logik
- Gerundete Kommazahlen (`DOUBLE / FLOAT`):
 - Wissenschaftliche Berechnungen / ggf. Messwerte,...

DATUMSANGABEN SPEICHERN

- PostgreSQL unterstützt verschiedene Typen:
- `TIMESTAMP / TIMESTAMP WITH TIME ZONE`
 - Datumsangabe (inkl. Uhrzeit)
- `DATE`:
 - Datumsangabe (ohne Uhrzeit)
- `TIME / TIME WITH TIME ZONE`
 - Uhrzeitangabe

DATUMSANGABEN SPEICHERN

- **TIMESTAMP:**
 - Wird 1:1 so abgespeichert wie angegeben
 - Es findet keine Umrechnung statt
- **TIMESTAMP WITH TIME ZONE:**
 - Wird beim Abspeichern in UTC umgewandelt
 - Wird beim Auslesen von UTC zurück in die lokale Zeitzone umgewandelt

MIT DATUMSWERTEN RECHNEN

- `CURRENT_TIMESTAMP`, `LOCALTIMESTAMP`:
 - Gibt die aktuelle Uhrzeit aus
 - `DATE_PART(part, timestamp)`: Gibt einen Teil vom Datum aus
z.B. das Jahr, den Monat,...
- Mit Datumswerten rechnen:
 - `SELECT timestamp '2020-01-01 00:00:00' - timestamp '2019-08-20 00:00:00'`
 - `SELECT timestamp '2019-01-01 00:00:00' + interval '2 days'`

PRIMÄRSCHLÜSSEL & AUTO INCREMENT

- MySQL:
 - Primärschlüssel:
 - Ist Eindeutig pro Tabelle
 - Darüber können Einträge angesteuert werden
 - Auto Increment:
 - Wenn wir einen neuen Eintrag einfügen, wird automatisch eine neue ID vergeben
- PostgreSQL:
 - Datentyp `SERIAL` / `BIGSERIAL`

PRIMÄRSCHLÜSSEL & AUTO INCREMENT

MySQL:

```
■ CREATE TABLE t (  
    id INT NOT NULL AUTO INCREMENT,  
    ...,  
    PRIMARY KEY (id)  
)
```

PostgreSQL:

```
■ CREATE TABLE t (  
    id SERIAL PRIMARY KEY,  
    ...  
)
```

```
CREATE TABLE t (  
    id SERIAL PRIMARY KEY,  
    name VARCHAR(100),  
    vorname VARCHAR(100)  
)
```

```
CREATE TABLE t2 (  
    id SERIAL,  
    name VARCHAR(100),  
    vorname VARCHAR(100),  
    PRIMARY KEY(id)  
)
```




TABELLE ÄNDERN . ALTER TABLE

- HINZUFÜGEN einer Spalte
 - ALTER TABLE table ADD COLUMN column datatype
- LÖSCHEN einer Spalte
 - ALTER TABLE table DROP COLUMN column
- VERÄNDERN einer Spalte
 - ALTER TABLE table ALTER COLUMN column TYPE datatype
- LÖSCHEN einer TABELLE
 - DROP TABLE table

```
ALTER TABLE t ADD COLUMN info TEXT
```

```
ALTER TABLE t ALTER COLUMN info  
TYPE VARCHAR(300)
```

```
ALTER TABLE t ADD COLUMN delcol  
TEXT
```

```
ALTER TABLE t DROP COLUMN delcol
```

```
DROP TABLE t2
```

FINETUNING . ALTER TABLE U.A.

– NOT NULL nachträglich setzen

```
ALTER TABLE t
```

```
    ALTER COLUMN name SET NOT NULL;
```

– Nachkommenfeld hinzufügen

```
ALTER TABLE t ADD COLUMN preis NUMERIC(5,2)
```

– DEFAULT-WERT setzen

```
ALTER TABLE t
```

```
    ALTER COLUMN preis SET DEFAULT 0.00;
```

– Umbenennen von Spalten

```
ALTER TABLE t
```

```
    RENAME COLUMN name TO last_name;
```

– ID löschen/hinzufügen

```
ALTER TABLE t
```

```
    DROP COLUMN id
```

```
ALTER TABLE t ADD COLUMN ID SERIAL
```

– ! FUNKTIONIERT NICHT !

```
ALTER TABLE t ALTER COLUMN ID TYPE SERIAL
```

– Über Properties verändern - sonst aufwendig, mit Sequenz

– Kopieren von Tabellen

```
CREATE TABLE table-neu AS TABLE t-alt
```

ERSTELLEN VON TABELLEN – VARIATIONEN

- KLASSISCH

```
CREATE TABLE tab (  
    id SERIAL PRIMARY KEY,  
    nachname VARCHAR(150),  
    vorname VARCHAR(150)  
);
```

- Tabelle aus Tabelle

```
CREATE TABLE books2  
    AS TABLE books;
```

Constraints etc.
gehen verloren

```
CREATE TABLE books2 AS (SELECT * FROM books)  
WITH NO DATA;
```

Constraints etc. gehen verloren
- ohne Daten

```
CREATE TABLE books2  
    (LIKE books INCLUDING DEFAULTS INCLUDING CONSTRAINTS INCLUDING INDEXES);
```

Constraints bleiben erhalten – aber ohne Daten

TABELLENDATEN – VARIATIONEN

- Elterntabelle vererbt Festlegungen an Kindtabelle

```
CREATE TABLE t2 (  
    color varchar(60) NOT NULL  
) INHERITS ( t1 );
```

Änderungen in Parenttabelle werden auch in Childtabelle wirksam. Zusätzliche Spalten können angelegt werden. Daten werden nicht übernommen.

- DATEN WIEDER HERSTELLEN AUS SICHERUNG (Tabelle muss vorhanden sein)

```
COPY t3 FROM 'C:/POSTGRESQL/tester.csv' WITH CSV HEADER;
```

- Daten aus vorhandener Tabelle in vorhandene Tabelle schreiben

```
INSERT INTO t4 (firstname,lastname)  
    SELECT firstname,lastname FROM customers;
```

ERSTELLEN VON TABELLEN – BEST PRACTISE

1. Erstellen der Tabellenstruktur

```
CREATE TABLE books2  
  (LIKE books INCLUDING DEFAULTS INCLUDING CONSTRAINTS INCLUDING INDEXES);
```

➔ Standardwerte, Constraints und Indexes bleiben erhalten – Daten werden nicht kopiert.

2. Hinzufügen der Daten

```
INSERT INTO books2  
  SELECT * FROM books;
```

➔ Daten werden vollständig in die neue Tabelle übernommen.

LÖSCHEN VON TABELLEN

Grundaufbau:

```
DROP TABLE tabelle;
```

Besser:

```
DROP TABLE IF EXISTS tabelle;
```

Noch besser:

```
DROP TABLE IF EXISTS tabelle CASCADE;
```

WIEDERHOLUNG

1. Erstellen einer neuen Tabelle [[Testen](#)] mit Autoinkrement und Primary Key mit Spalten: [<erfassdat>](#), [<updatedat>](#), [<textfeld>](#)
2. Hinzufügen einer Spalte mit Datentyp INTEGER mit Standardwert 0
3. Ändern dieser Spalte auf NUMERIC mit Standardwert 0.00
4. Umbenennen der Spalte [<textfeld>](#) in [<mytextfeld>](#)
5. Die Tabelle soll komplett mit Daten kopiert werden [[Testen02](#)]
6. Sichern der Daten ins POSTGRESQL-Verzeichnis mit Dateinamen [[Testen02.csv](#)]
7. Sichern der Tabelle als Backup [[Testen02_backup](#)]

ÜBUNGEN ZU TABELLEN

1. Schauen Sie sich den Aufbau der Sicherungsdatei tester.csv an.
2. Erstellen Sie eine Tabelle [`ttester`], die die Struktur der Sicherungsdatei umfasst.
3. Sichern Sie die Daten aus tester.csv zurück.
4. Erstellen Sie eine Tabelle mit Namen [`t1`], die folgende Spalten `<id>`, `<title>`, `<firstname>`, `<lastname>`, `<street>`, `<city>`, `<email>`, `<age>` enthält.
5. Sichern Sie die Daten aus customers.csv in die Tabelle [`t1`] zurück,
6. Erstellen Sie eine Sicherungskopie der Daten aus [`t1`] in eine Tabelle [`t2`] mit allen Key Constraints etc.



CONSTRAINTS

OHNE EINSCHRÄNKUNGEN GEHT ES NICHT - SORRY

CONSTRAINT-TYPEN

Typ	Beschreibung
NOT NULL	Stellt sicher, dass eine Spalte keinen NULL-Wert haben kann
UNIQUE	Stellt sicher, dass alle Werte in einer Spalte unterschiedlich sind
PRIMARY KEY	Einzigartige Identifizierung jeder Zeile in einer Tabelle
FOREIGN KEY	Verknüpft Daten zwischen Tabellen
CHECK	Stellt sicher, dass alle Werte in einer Spalte eine bestimmte Bedingung erfüllen
EXCLUSION	Stellt sicher, dass, wenn zwei Zeilen anhand der angegebenen Spalten mit den angegebenen Operatoren verglichen werden, nicht alle Vergleiche TRUE zurückgeben

VORARBEITEN

- Erstellen Sie eine neue Datenbank [myprojekt]
- Verbinden Sie sich mit der Datenbank
- Öffnen Sie ein neues Abfragefenster
- Laden Sie das Skript “myProdukte_Bestellungen_Kunden.sql” und lassen Sie es laufen
- Überprüfen Sie vorhandene Keys, Sequenzen und Constraints
- Rufen Sie das ERD-Diagramm auf
- Identifizieren Sie mögliche vorhandene Verbindungen zwischen den Tabellen

NOT NULL | CHECK

- Sicherstellen, dass Kundenname angegeben wird und Email auch @ enthält

```
DROP TABLE IF EXISTS myKunden;
```

```
CREATE TABLE myKunden (
```

```
    kunden_id SERIAL PRIMARY KEY,
```

```
    name text NOT NULL, -- ist eine Einschränkung, wird aber nicht durch gesondertes CONSTRAINT geregelt
```

```
    email_d VARCHAR(200) CHECK (POSITION('@' IN email_d) > 0),
```

```
    email_p VARCHAR(200) CHECK (POSITION(('@'::text) IN (email_p)) > 0)
```

```
);
```

LÖSCHEN VON CONSTRAINTS

```
ALTER TABLE myProdukte  
    DROP CONSTRAINT myprodukte_price_check;
```

```
ALTER TABLE myProdukte  
    DROP CONSTRAINT IF EXISTS myprodukte_price_check;
```


ERSTELLEN VON CONSTRAINTS

```
ALTER TABLE myProdukte ADD CONSTRAINT myProducte_price_check  
CHECK (price > 0);
```

```
ALTER TABLE myProdukte ADD CONSTRAINT myProducte_price  
CHECK (discounted_price > 0 AND price > discounted_price);
```

```
ALTER TABLE myKunden ADD CONSTRAINT myKunden_emailknr_unique_key  
UNIQUE (email, kundennummer); --Multicolumn UNIQUE
```

NULL DEFINIEREN

- Nachträglich NOT NULL festlegen

```
ALTER TABLE studenten ALTER COLUMN vorname SET NOT NULL;
```

- NOT NULL rückgängig machen

```
ALTER TABLE studenten ALTER COLUMN vorname DROP NOT NULL;
```

DEFAULT-WERT

- Zahl als Standard

```
ALTER TABLE orders ALTER COLUMN preis SET DEFAULT 0.00;
```

- String als Standard

```
ALTER TABLE kunde ALTER COLUMN status SET DEFAULT 'Neu';
```

- Datumswert als Standard

```
ALTER TABLE kunde ALTER COLUMN erfassdat SET DEFAULT CURRENT_TIMESTAMP;
```

DER PRIMARY KEY & SEQUENZEN

```
ALTER TABLE myKunden DROP CONSTRAINT IF EXISTS mykunden_pkey;
```

Löschen PK

```
-- ALTER TABLE myKunden DROP CONSTRAINT mykunden_pkey CASCADE;
```

```
ALTER TABLE myBestellungen DROP CONSTRAINT mybestellungen_kunden_id_fk;
```

Löschen FK

```
DROP SEQUENCE IF EXISTS myKunden_kunden_id_seq CASCADE;
```

Löschen Sequenz

```
ALTER TABLE myKunden ADD PRIMARY KEY (kunden_id);
```

Erstellen PK

```
CREATE SEQUENCE myKunden_kunden_id_seq;
```

Erstellen Sequenz

```
ALTER TABLE myKunden ALTER COLUMN kunden_id  
SET DEFAULT nextval('myKunden_kunden_id_seq');
```

Zuordnen Sequenz

ERSTELLEN VON FREMDSCHLÜSSELN

```
ALTER TABLE myBestellungen  
DROP CONSTRAINT IF EXISTS mybestellungen_product_id_fk;
```

```
ALTER TABLE myBestellungen  
ADD CONSTRAINT mybestellungen_product_id_fk  
FOREIGN KEY (product_id)  
REFERENCES myProdukte (product_id);
```

WIEDERHOLUNG

1. Erstellen einer neuen Tabelle [nurZumTesten] mit Autoinkrement und Primary Key mit Spalten: `<erfassdat>`, `<updatedat>`, `<textfeld>`
2. Löschen des Primary Keys und des Autoinkrements
3. Nachträgliches Erstellen eines Primary Key
4. Nachträgliches Erstellen einer Sequenz
5. Hinzufügen einer Spalte mit Datentyp INTEGER mit Standardwert 0
6. Ändern dieser Spalte auf NUMERIC mit Standardwert 0.00
7. Spalten `<erfassdat>` und `<updatedat>` sollen Standardwert „Aktueller Timestamp mit Zeitzone“ erhalten
8. Spalte `<textfeld>` muss ausgefüllt sein und mindestens 4 Zeichen enthalten
9. Spalte `<textfeld>` darf doch unausgefüllt bleiben
10. Die Tabelle soll komplett mit Daten kopiert werden [nurZumTesten02]
11. Sichern der Daten ins POSTGRESQL-Verzeichnis mit Dateinamen [nurZumTesten02.csv]

ÜBUNGEN TABELLEN & CONSTRAINTS

1. Erstellen Sie eine neue Datenbank [**unibook**]. Sichern Sie das Backup zurück.
2. Erstellen Sie eine neue Tabelle [studenten100] nur mit Datentypen mit folgender Struktur:
`student_id, erfass_dat, nachname, vorname, email, letzte_zahlung_dat, letzte_zahlung`
3. Legen Sie den PRIMARY KEY für die Tabelle [studenten100] fest und sorgen Sie dafür, dass bei jedem neuen Datensatz die <student_id> um 1 hochgezählt wird.
4. <erfass_dat> und <letzte_zahlung_dat> erhalten den Standardwert des aktuellen TIMESTAMP
5. <letzte_zahlung> erhält den Standardwert 0.00
6. Ergänzen Sie die Spalte <fon>.
7. <nachname> und <vorname> müssen angegeben werden.
8. Die <email> muss ein gültiges Format aufweisen und muss einmalig sein.

ÜBUNGEN TABELLEN & CONSTRAINTS

1. Schauen wir uns die unibook.sql an.
2. Lassen Sie das Skript unibook.sql laufen.
3. Da ist einiges durcheinander geraten. Welche Daten hängen miteinander zusammen, welche sind doppelt, welche müssen wir mit Foreign Keys verbinden?
Erarbeiten Sie einen Vorschlag. Lassen Sie uns Ihren Vorschlag besprechen bevor Sie die richtigen Tabellen mit Foreign Keys verbinden.
4. Der Datentyp von <letzte_zahlung> in der Tabelle [studenten100] soll noch auf NUMERIC(10,2) geändert werden.
5. Es soll außerdem ausgeschlossen werden, dass ein Student mehr als 100 € an Gebühren zahlt.
6. <nachname> und <vorname> dürfen jetzt doch NULL sein. Entfernen Sie die Bedingung NOT NULL.

ÜBUNGEN TABELLEN & CONSTRAINTS

1. Beseitigen Sie jetzt die überflüssigen Tabellen. Vorher sollten wir über TRANSAKTIONEN reden.
2. Verbinden Sie die verbleibenden Tabellen sinnvoll.
3. Lassen Sie sich anzeigen, wie viele Buchungen für welche Kurse vorliegen.
4. Welche Kurse werden von welchen Dozenten durchgeführt?
5. Es soll eine Liste für die Dozentin mit `<doz_id> 4` und `<kurs_id> 9` mit Thema, Dozentin und allen zugehörigen Student:innen ausgegeben werden.
6. Dieselbe Dozentin möchte auch wissen, für wie viele und welche Student:innen sie verantwortlich ist.

ÜBUNGEN FREMDSCHLÜSSEL [MYPROJEKT]

1. Wechseln Sie in die Datenbank [myProjekt].
2. Wir benötigen einen Fremdschlüssel zwischen <product_id> in [myBestellungen] und <product_id> in [myProdukte].
3. Kontrollieren Sie, ob dieser sowohl in der GUI als auch im ERD angezeigt werden.
4. Finden Sie heraus, welche Fremdschlüssel in der Datenbank [bookstore] festgelegt wurden.
5. Schreiben Sie einen Code, der
 - a. die vorhandenen Fremdschlüssel löscht
 - b. diese wieder anlegt
6. Löschen Sie in der Tabelle [orders] den Primary Key.
7. Legen Sie diesen wieder an.



TRANSAKTIONEN

AUF NUMMER SICHER GEHEN

AUFGABE VON TRANSAKTIONEN

- Transaktionen sichern ab, dass mehrere Operationen ausgeführt werden – auch wenn es zu einem Systemcrash oder ähnlichen Schwierigkeiten kommt. („Alles-oder-Nichts“-Prinzip)
- Konsistente Datenbestände und Regelung von konkurrierenden Zugriffen auf die Datenbank sind so möglich.
- Transaktionen entsprechen den ACID-Eigenschaften:
 - A – atomicity (alle Operationen innerhalb der Transaktion müssen ausgeführt worden sein oder es wird der vorherigen Zustand der Datenbank wieder hergestellt)
 - C – consistency (konsistente Regeln werden angewendet – z.B. foreign keys, constraints und trigger werden vor und nach der Transaktion sichergestellt)
 - I – isolation (Operationen in einer Transaktion werden von Operationen von anderen Transaktionen isoliert)
 - D – durability (...)

VORBEREITUNG

```
DROP TABLE IF EXISTS accounts;

CREATE TABLE accounts (
    account_id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    account NUMERIC(10,2));

INSERT INTO accounts(name,account)
VALUES
    ('Schmidt, Rene',100.00),
    ('Elbers, Anna', 100.00);

SELECT * FROM accounts;
```

GRUNDAUFAU

```
BEGIN;
```

```
-- SQL-Operationen
```

```
COMMIT; -- or ROLLBACK,
```

Beispiel:

```
BEGIN;
```

```
    UPDATE accounts SET account = account - 100 WHERE account_id = 1;
```

```
    UPDATE accounts SET account = account + 100 WHERE account_id = 2;
```

```
COMMIT;
```

```
END;
```

WHEN EXISTS

```
DO
$$
BEGIN
    CASE
        WHEN EXISTS(SELECT account FROM accounts WHERE account_id =1 and account > 80) THEN
            RAISE NOTICE 'Genug Geld';
        WHEN NOT EXISTS(SELECT account FROM accounts WHERE account_id =1 and account > 80) THEN
            RAISE NOTICE 'Nicht genug Geld';
    END CASE;
END;
$$ LANGUAGE plpgsql;
```

CASE

```
DO
$$
DECLARE
    v_row_count INT:=0;
BEGIN
    SELECT COUNT(*) INTO v_row_count FROM accounts WHERE account_id = 1 and account > 80;
    --SELECT COUNT(*) FROM accounts WHERE account_id = 1 and account > 80
    CASE v_row_count
        WHEN 1 THEN
            RAISE NOTICE 'Genug Geld';
            -- update
            SELECT COUNT(*) INTO v_row_count FROM accounts WHERE account_id = 2;
            WHEN 1 THEN
                RAISE NOTICE 'Empfänger vorhanden';
                --UPDATE ...

            COMMIT;
        ELSE
            RAISE NOTICE 'Empfänger nicht vorhanden';
            ROLLBACK;
        WHEN 0 THEN
            RAISE NOTICE 'Nicht genug Geld';
            ROLLBACK;
        ELSE
            RAISE NOTICE 'Überprüfung nicht möglich';
    END CASE;
END;
$$ LANGUAGE plpgsql;
```


FOR I IN 1..N LOOP

```
DROP TABLE IF EXISTS testcnt;
CREATE TABLE testcnt( transaction_ids bigint);

DO
$$
    DECLARE
        v_cnt integer := 3;
    BEGIN
        FOR i IN 1..v_cnt LOOP
            INSERT INTO testcnt(transaction_ids) VALUES(txid_current());
            --COMMIT;
        END LOOP;
        COMMIT;
    END;
$$ LANGUAGE plpgsql;
```

LOCKING – ROW-LEVEL-LOCK

-- Transaktion 1

BEGIN;

SELECT * FROM accounts WHERE id = 1 FOR UPDATE;

-- Transaktion 1 erstellt einen exklusiven Lock auf den DS mit id = 1

UPDATE accounts SET balance = balance + 80 WHERE id = 1;

SELECT * FROM accounts WHERE id = 2 FOR UPDATE;

UPDATE accounts SET balance = balance - 80 WHERE id = 2;

COMMIT;

/* Eine Transaktion 2 muss warten bis Lock Transaktion 1 aufgehoben wurde durch COMMIT oder ROLLBACK */

TABLE-LEVEL-LOCK

```
-- ***** Table-Level-Lock
```

```
BEGIN;
```

```
    LOCK TABLE accounts IN EXCLUSIVE MODE;
```

```
        -- Transaktion 1 blockiert die gesamte Tabelle exklusiv
```

```
    UPDATE accounts SET balance = balance + 80 WHERE id = 1;
```

```
COMMIT;
```

IF (SELECT ...) > 0

```
DO
$$
BEGIN -- or START TRANSACTION
    IF (SELECT COUNT(*) FROM accounts WHERE account_id = 1 and account > 80) > 0 THEN
        UPDATE accounts SET account = account - 80 WHERE account_id = 1;
        IF (SELECT COUNT(*) FROM accounts WHERE account_id = 2) > 0 THEN
            UPDATE accounts SET account = account + 80 WHERE account_id = 2;
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
    ELSE
        RAISE NOTICE 'account 1 - Nicht genug Geld';
        ROLLBACK;
    END IF;
END;
$$
```

LOOP ITERATIVE

```
DO
$$
DECLARE
    v_count INT := 0;
    v_iteration_count INT := 1;
BEGIN
    LOOP
        RAISE NOTICE 'Iteration Count %', v_iteration_count;
        v_count:=v_count+1;
        EXIT WHEN v_count=10;
        v_iteration_count:=v_iteration_count+1;
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```



INDEXE

DATENABFRAGEN BESCHNLEUNIGEN

WARUM INDIZIEREN?

- Bisher wurden bei einem Filter immer alle Daten durchsucht. [Full-Table-Scan]
- Das ist ausgesprochen langsam, weil alle Daten in den Arbeitsspeicher geladen werden müssen, um durchsucht werden zu können.
- Mit einem Index können wir das Durchsuchen einer Spalte massiv beschleunigen, weil diese unseren Datenbestand quasi vorsortieren und schnelle Wege zum gesuchten Wert kennen. M.a.W.: Ein Index ist eine Art Aufzeichnung, aus der hervorgeht, wo die gesuchten Daten am schnellsten zu finden sind. Hierdurch werden weniger Gesamtdaten durchsucht, reduzieren sich die „Leistungskosten“ zur Verarbeitung der Suchanfrage. [Heap-Datei; zumeist auf Bäumen basierende abstrakte Datenstruktur] [➔ 191 ff.]
- Mathematisch ausgedrückt: $O(n) \Rightarrow O(\log)$
- <https://www.wolframalpha.com/> – Rechner, um logarithmische Beschleunigung zu veranschaulichen.

WIE FUNKTIONIERT EIN INDEX?

<https://visualgo.net/en>

Binary search tree → AVL Tree (Balanced Tree)












Datenbank [bookstore]

EXPLAIN

SELECT * FROM orders WHERE

timestamp >= '2010-01-01 00:00:00' AND

timestamp <= '2010-12-31 00:00:00'

Data Output		Nachrichten	Notifications
         			
	QUERY PLAN text		
1	Seq Scan on orders (cost=0.00..23.78 rows=62 width=28)		
2	Filter: (("timestamp" >= '2010-01-01 00:00:00'::timestamp without time zone) AND ("timestamp" <= '2010-12-31 00:00:00'::timestamp without time z...		

ERSTELLEN EINES INDEXES

Grundaufbau:

```
CREATE INDEX <name> ON <tabelle> (  
    <spalte> ASC  
)
```

```
CREATE INDEX ON orders (timestamp)
```

reicht aus. Dann wird der Name des Indexes
<orders_timestamp_idx> automatisch erzeugt.

```
CREATE INDEX orders_timestamp_index ON orders (  
    timestamp ASC  
)
```

Selbst festgelegter Indexname

Data Output		Nachrichten	Notifications
<div><div><div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div><div><div></div><div></div></div></div></div></div>			

ERZWINGEN UND LÖSCHEN EINES INDEXES

- Erzwingen der Anwendung eines Indexes:

Postgres entscheidet selbst, ob die Anwendung des Indexes genutzt wird oder nicht.

Um die Wahrscheinlichkeit der Indexnutzung zu steigern, werden die erlaubten “Kosten” pro Suche von 4.0 auf 1 mit folgendem Befehl heruntergesetzt.

```
SET random_page_cost = 1;
```

```
CREATE INDEX customers_firstname_index ON customers (  
    firstname ASC);
```

- Löschen eines erstellten Indexes:

```
DROP INDEX IF EXISTS customers_firstname_index;
```

INDEX ÜBER MEHRERE SPALTEN

Für Suchen, die sich häufig auf mehrere Spalten beziehen, kann es sinnvoll sein, einen Index zu erstellen, der mehrere Spalten enthält.

ACHTUNG: Vermeiden Sie hier entgegengesetzte Sortierreihenfolgen der Indexspalten.

```
CREATE INDEX customers_fullname_index ON customers USING btree (  
    firstname ASC,  
    lastname ASC  
);
```

EXPLAIN

```
SELECT firstname, lastname FROM customers  
    WHERE firstname = 'Dana' AND lastname = 'Mai'
```

BENCHMARKING

Modifizieren wir unsere bisherige EXPLAIN SELECT – Anweisung folgendermaßen:











EXPLAIN ANALYSE

SELECT firstname, lastname

FROM customers

WHERE firstname ='Dana' AND lastname ='Mai'

ORDER BY firstname, lastname

Data Output		Nachrichten	Notifications
         SQL			
	QUERY PLAN		
	text		
1	Index Only Scan using customers_fullname_index on customers (cost=0.14..2.16 rows=1 width=14) (actual time=0.023..0.024 rows=1 loop...		
2	Index Cond: ((firstname = 'Dana'::text) AND (lastname = 'Mai'::text))		
3	Heap Fetches: 1		
4	Planning Time: 0.085 ms		
5	Execution Time: 0.036 ms		

Zeit, die zur Ausführung der Abfrage benötigt wurde.
Ausführungszeit von PG4 stellt Übertragungszeit zum Server dar.

SPEICHERBEDARF VON INDIZES

```
SELECT pg_size_pretty(pg_relation_size('bigdata'));
```

```
SELECT pg_size_pretty(pg_relation_size('bigdata_id_idx'));
```

```
SELECT pg_size_pretty(pg_relation_size('bigdata_details_id_idx'));
```

Was Sie bedenken sollte bzgl. der Anzahl der Indizes, die Sie erstellen:

1. Indizes brauchen Platz.
2. Indizes müssen aktualisiert werden, wenn es zu Änderungen in den Daten kommt (Einfügen, Verändern, Löschen)
 - das belastet die Performance der Datenbank.
3. PostgreSQL nutzt nicht in jedem Fall den erstellten Index.

AUTOMATISCH ERSTELLTE INDIZES

Es werden – ohne dass wir das merken – Indizes erstellt für:

- PRIMARY KEY
- UNIQUE Constraints

Diese Indizes werden in pgAdmin NICHT unter Indizes angezeigt.

Folgende Anweisung macht auch diese sichtbar:

```
SELECT relname, relkind
FROM pg_class
WHERE relkind = 'i';
```

BENCHMARKING

Um den Planer zu zwingen, der durch explizite JOINS festgelegten Verbindungsreihenfolge zu folgen, setzen Sie den Laufzeitparameter

```
SET join_collapse_limit = 1;
```

Beispiel mit folgenden 3 logisch identischen Anweisungen [Datenbank: dvdrental]

```
EXPLAIN ANALYZE -- 2.9 - 3.0
```

```
SELECT * FROM film AS f, film_actor AS fa, actor AS a  
WHERE f.film_id = fa.film_id AND fa.actor_id = a.actor_id;
```

```
EXPLAIN ANALYZE -- 3.0 - 3.0
```

```
SELECT * FROM film AS f CROSS JOIN film_actor AS fa CROSS JOIN actor AS a  
WHERE f.film_id = fa.film_id AND fa.actor_id = a.actor_id;
```

```
EXPLAIN ANALYZE -- 2.9 -- 2.6
```

```
SELECT * FROM film AS f JOIN (film_actor AS fa JOIN actor AS a ON (fa.actor_id = a.actor_id))  
ON (f.film_id = fa.film_id);
```

```
-- > 3. Var. hat die deutlichste Verbesserung – weitere Steigerung durch Index auf FOREIGN KEY
```

VERSCHIEDENE INDEX-TYPEN

<https://www.cs.usfca.edu/~galles/visualization/OpenHash.html>

Ziffern und Strings können systematisch einsortiert werden und werden so schneller gefunden.

HASH-Index ist für Bereichssuchen (LIKE, BETWEEN ...) ungeeignet.

BTREE ist i.d.R. die beste Wahl.



VIEWS & MATERIALIZED VIEWS



WARUM VIEWS?

1. Hinter VIEWS verbergen sich häufig komplexe Abfragen, die ich (anderen) Nutzern nicht zugänglich machen oder zumuten möchte.
2. VIEWS ermöglichen es, den Unternehmenskonventionen konforme Spaltenbezeichnungen z.B. standardmäßig in Abfrageergebnissen anzubieten. (z.B. nach einer Migration)
3. VIEWS reduzieren die benutzten Spalten auf eine überschaubare Größe.
4. VIEWS können Daten vorfiltern, so dass diese von unberechtigten Usern nicht eingesehen werden können.
5. VIEWS können den Zugriff auf Tabellen etc. auch den Usern ermöglichen, die standardmäßig hierzu keine Rechte besitzen. Dies kann aber auch verhindert werden.
6. VIEWS können auch von weniger geübten SQL-Anwendern genutzt werden.
7. Abfragen können auf VIEWS genauso angewendet werden wie auf die Originaltabellen.
8. VIEWS können auch mehrere Anweisungen nacheinander abarbeiten (z.B. SELECT, UPDATE, DELETE ...). Dies ermöglicht z.B. Anpassungen, die im Verborgenen laufen, um aktualisierte Daten anbieten zu können. Die erste Anweisung ist aber immer ein SELECT.

WARUM VIEWS?

1. VIEWS sind Pseudotabellen, Standardmäßig ist ein VIEW keine echte Tabelle, sondern greift auf eine oder mehrere Tabellen zurück, um ausgewählte oder alle Spalten auszugeben. VIEWS stellen Daten also zur Verfügung, wenn diese ausgeführt wird. Die Daten einer einfachen VIEW sind nur in den abgerufenen „richtigen“ Tabellen gespeichert. [anders bei: MATERIALIZED VIEWS]
2. VIEWS können befragt werden, wie „richtige“ Tabellen.
3. Die zur Verfügung gestellten Daten basieren allerdings auf der Abfrage in der View. Gibt es hier Filter, werden auch nicht alle Daten angezeigt werden können. Spalten, die nicht in der VIEW enthalten sind, können über diese auch nicht abgefragt werden.
4. In einer VIEW können auch UPDATE, DELETE, INSERT-Befehle benutzt werden. Diese werden allerdings nur beim Erstellen der VIEW ausgeführt.
5. Die Daten einer VIEW sind bei jedem Abfragen der VIEW aktuell, sofern diese keine in der VIEW enthaltenen UPDATE-, INSERT-Anweisungen voraussetzen. Dies kann z.B. über den Aufruf von Prozeduren, Transaktionen oder Triggern geregelt werden, in denen oder durch die eine Neuerstellung des VIEWS und damit eine Aktualisierung erfolgt oder eine Datenaktualität gewährleistet wird.

EINSCHRÄNKUNGEN BEI VIEWS?

Sie können Zeilen in einer Ansicht einfügen, aktualisieren und löschen, mit einigen Einschränkungen:

1. Wenn die Sicht Verknüpfungen zwischen mehreren Tabellen enthält, können Sie nur eine Tabelle in die Ansicht einfügen und aktualisieren, und Sie können keine Zeilen löschen.
2. Sie können Daten in Ansichten, die auf Union-Abfragen basieren, nicht direkt ändern. Sie können keine Daten in Ansichten ändern, die GROUP BY- oder DISTINCT-Anweisungen verwenden.
3. Alle Spalten, die geändert werden, unterliegen denselben Einschränkungen, als ob die Anweisungen direkt für die Basistabelle ausgeführt würden.
4. Text- und Bildspalten können nicht über Ansichten geändert werden.

ERSTELLEN & ABFRAGEN VON VIEWS

```
SELECT * FROM customers  
WHERE firstname LIKE 'E%'  
ORDER BY firstname;
```

Grundaufbau:

```
CREATE OR REPLACE VIEW <viewname> AS  
    SELECT ...;
```

Löschen von Views:

```
DROP VIEW IF EXISTS <viewname>;
```

Beispiel:

```
CREATE VIEW v_cust_order_top10 AS  
SELECT c.lastname AS "Nachname", c.firstname AS  
    "Vorname", SUM(o.amount) AS "Bestellwert",  
COUNT(o.amount) as  
    "Anzahl Bestellungen"  
FROM customers AS c  
JOIN orders AS o ON c.id = o.customer_id  
GROUP BY c.lastname, c.firstname  
ORDER BY SUM(o.amount) DESC LIMIT 10;
```

```
SELECT * FROM v_cust_order_top10;
```

```
SELECT * FROM v_cust_order_top10  
WHERE "Vorname" LIKE 'E%'  
ORDER BY "Vorname";
```

VIEWS MIT MEHREREN ANWEISUNGEN

```
DROP VIEW IF EXISTS v_akt_amount;
```

```
CREATE VIEW v_akt_amount AS
```

```
    SELECT A.id AS aid, A.lastname, A.firstname, A.orders AS aorders, A.amount AS aamount  
    FROM customers AS A;
```

```
    UPDATE v_akt_amount SET aorders =  
        (SELECT COUNT(*) FROM orders AS B  
         WHERE aid = B.customer_id);
```

```
    UPDATE v_akt_amount SET aamount =  
        (SELECT SUM(amount) FROM orders AS C  
         WHERE aid = C.customer_id);
```

```
SELECT * FROM v_akt_amount WHERE aamount IS NOT NULL  
ORDER BY aamount DESC;
```

Wurden Spalten benannt muss in den View-Anweisungen mit diesen gearbeitet werden!
Gleiches gilt für die Abfragen, die sich auf das View beziehen.

WAS IST EINE MATERIALIZED VIEW?

Im Unterschied zur VIEW werden durch die Erstellung eines MATERIALIZED VIEWS Daten in eine physische Tabelle gespeichert. Wann bieten sich das an?

- Bei komplexen, wenig performanten Abfragen.
- Bei großen Datenbeständen, die diesen zusätzlichen Schritt rechtfertigen.
- Bei Aufgabenstellungen, wo Daten berechnet werden und in den Originaltabellen keine zusätzlichen Spalten zum Speichern dieser Werte erstellt werden sollen oder dürfen.
- Bei Daten, die sich seltener ändern oder in einem längeren Zyklus erneuert werden.
- Bei Daten, die usern mit geringen Zugriffsrechten zur Verfügung gestellt werden sollen.

M.a.W.: Eine VIEW greift auf aktuelle Daten zu. Eine MATERIALIZED VIEW greift auf einen vom Echtbetrieb getrennten Datenbestand zurück und muss deshalb erneuert werden, wenn sich die Daten geändert haben und die gewünschten Daten aktuell sein müssen.

MATERIALIZED VIEW – SYNTAX

```
DROP MATERIALIZED VIEW IF EXISTS mv_customers_bestellungen;
```

```
CREATE MATERIALIZED VIEW mv_customers_bestellungen AS  
    SELECT id, lastname, firstname,  
           (SELECT COUNT(*) FROM orders AS o  
            WHERE c.id = o.customer_id) AS anz_bestellungen  
    FROM customers AS c  
    ORDER BY anz_bestellungen DESC;
```

```
SELECT * FROM mv_customers_bestellungen;
```

```
REFRESH MATERIALIZED VIEW mv_customers_bestellungen;
```




STORED FUNCTIONS

PROGRAMME MIT SQL ERSTELLEN

WARUM FUNCTIONS?

- Erlaubt das Schreiben eigener Programme.
- Es erfolgt eine direkte Ausführung in der Datenbank.
- Komplexe Anwendungslogiken können direkt in der Datenbank ausgeführt werden.
- Rechte können hiermit modifiziert werden ohne grundsätzlich die Rechtestruktur aufzugeben.
- FUNCTIONS werden in SQL geschrieben.
- PROCEDURES werden in PL/pgSQL geschrieben.
 - Ermöglicht Einsatz von erweiterten Kontrollstrukturen (z.B. For-Schleifen).
 - PROCEDURES sind also eine Erweiterung der FUNCTIONS.
 - Sehr ähnlich wie bei ORACLE.

FUNCTIONS – EINFACHE SYNTAX

- Laden Sie bitte das Script CREATE_emp.sql und führen Sie es aus.

```
DROP FUNCTION IF EXISTS clean_emp();
```

```
CREATE FUNCTION clean_emp() RETURNS void AS  
    DELETE FROM emp  
        WHERE salary IS NULL OR salary = 0;  
LANGUAGE SQL;
```

```
SELECT clean_emp();
```

```
CREATE OR REPLACE FUNCTION clean_emp() AS '  
    DELETE FROM emp  
        WHERE salary IS NULL OR salary = 0;  
' LANGUAGE SQL;
```

Wenn grundlegende Syntaxveränderungen (z.B. Veränderung des RETURNS) vorgenommen werden, reicht REPLACE **nicht** aus. Function muss mit DROP gelöscht und dann neu mit CREATE erzeugt werden.

FUNCTION muss gelöscht und neu erzeugt werden, wenn grundlegende Änderungen vorgenommen wurden.

SYNTAX MIT START- & END-TOKEN

- Laden Sie bitte das Script CREATE_emp.sql und führen Sie es aus.

```
DROP FUNCTION IF EXISTS clean_emp();
```

```
CREATE FUNCTION clean_emp() RETURNS void AS $$  
    DELETE FROM emp  
    WHERE salary IS NULL OR salary = 0;  
$$ LANGUAGE SQL;
```

```
CREATE OR REPLACE FUNCTION clean_emp() AS  
$CODE$  
    DELETE FROM emp  
    WHERE salary IS NULL OR salary = 0;  
$CODE$ LANGUAGE SQL;
```

```
SELECT clean_emp();
```

Start- & Endtoken müssen übereinstimmen und mindestens 2 \$-Zeichen enthalten. Weitergehende Bezeichnungen (z.B. \$CODE\$) können beliebig festgelegt werden.

RÜCKGABEWERTE

- Im ersten Beispiel:

```
CREATE FUNCTION clean_emp() RETURNS void AS $$
```

```
...
```

```
$$
```

- Einfaches Beispiel, in dem ein Zahlenwert zurückgegeben wird:

```
CREATE FUNCTION select1() RETURNS integer AS $$
```

```
    SELECT 1 AS result
```

```
$$ LANGUAGE SQL;
```

```
SELECT select1();
```

RÜCKGABEWERTE – EINE ZEILE

- Einen Rückgabewert erhalten

```
DROP FUNCTION IF EXISTS select2;
```

```
CREATE FUNCTION select2() RETURNS integer AS  
$$
```

```
    SELECT COUNT(*) FROM customers
```

```
$$ LANGUAGE SQL;
```

```
SELECT select2();
```

RÜCKGABEWERTE – MEHRERE ZEILEN

```
DROP FUNCTION IF EXISTS select3;
```

```
CREATE OR REPLACE FUNCTION select3() RETURNS integer AS $$
```

```
    SELECT id FROM customers
```

```
$$
```

➔ Nur eine Zeile wird ausgegeben.

```
DROP FUNCTION IF EXISTS select3;
```

```
CREATE OR REPLACE FUNCTION select3() RETURNS SETOF integer AS $$
```

```
    SELECT id FROM customers
```

```
$$
```

➔ Alle Zeilen werden ausgegeben.

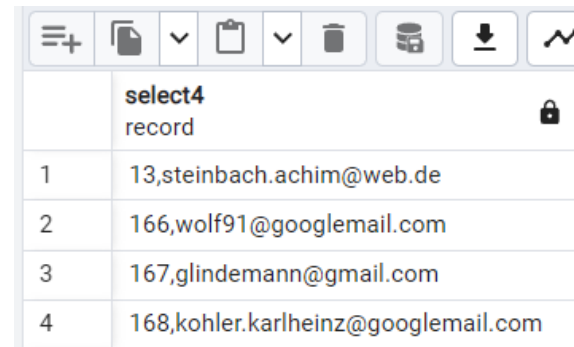
RÜCKGABEWERTE – MEHRERE SPALTEN

```
DROP FUNCTION IF EXISTS select4;
```

```
CREATE FUNCTION select4() RETURNS TABLE (id bigint, email varchar) AS $$  
    SELECT id, email FROM customers  
$$ LANGUAGE SQL;
```

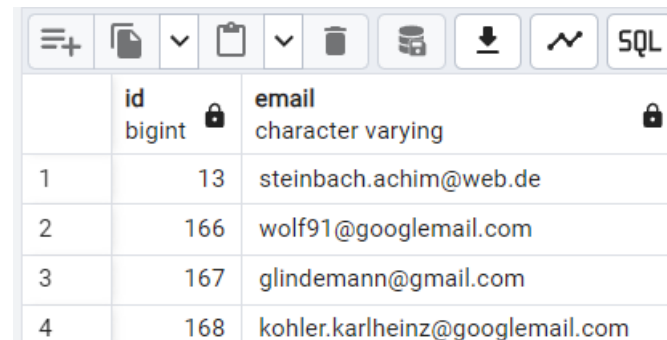
```
SELECT select4(); -- gibt "record" zurück
```

```
SELECT * FROM select4(); -- formatiert Rückgabe als Tabelle mit mehreren Spalten
```



The screenshot shows a database client interface with a toolbar at the top. Below the toolbar, a table titled 'select4 record' is displayed. The table has two columns: 'id' and 'email'. The data is as follows:

	id	email
1	13	steinbach.achim@web.de
2	166	wolf91@googlemail.com
3	167	glindemann@gmail.com
4	168	kohler.karlheinz@googlemail.com



The screenshot shows a database client interface with a toolbar at the top. Below the toolbar, a table titled 'select4' is displayed. The table has two columns: 'id' and 'email'. The data is as follows:

	id	email
1	13	steinbach.achim@web.de
2	166	wolf91@googlemail.com
3	167	glindemann@gmail.com
4	168	kohler.karlheinz@googlemail.com

FUNKTIONSPARAMETER ÜBERGEBEN

- Die Funktion CONCAT() enthält Parameter, die durch die Funktion verarbeitet werden.
Wie sieht eine Funktion aus, die mit Parametern aufgerufen wird, damit diese dann verarbeitet werden.

```
DROP FUNCTION IF EXISTS get_name;
```

```
CREATE FUNCTION get_name(varchar, varchar) RETURNS varchar AS $$  
    SELECT CONCAT($2, ', ', '$1')  
$$ LANGUAGE SQL;
```

```
SELECT get_name('Max', 'Müller');
```

MIT FUNKTIONSPARAMETERN FILTERN

- Mit Funktionen können übergebene Parameter zum Filtern der auszugebenden Datensätze genutzt werden.
- Beispiel: Durch Eingabe einer <id> soll ein spezifischer Kunde mit <id> und <email> ausgegeben werden.

```
DROP FUNCTION IF EXISTS get_customer(bigint);  
  
CREATE FUNCTION get_customer(bigint)  
    RETURNS TABLE(id bigint, email varchar) AS $MYCODE$  
    SELECT id, email FROM customers WHERE id = $1  
$MYCODE$ LANGUAGE SQL;  
  
SELECT * FROM get_customer(10);
```

FUNKTIONEN LÖSCHEN

Standardanweisung, wenn die FUNCTION nur **einmal** unter diesem Namen existiert:

```
DROP FUNCTION get_name; -- CONCAT-Funktion
```

Wenn es **mehrere FUNCTIONS mit dem gleichen Namen** aber unterschiedlichen Parameterdefinitionen gibt, müssen bei der Drop-Anweisung auch die Parameter angegeben werden:

```
DROP FUNCTION get_name(varchar, varchar);
```

wenn es z.B. auch folgende FUNCTION gibt

```
DROP FUNCTION get_name(bigint);
```

FUNCTIONS PERFORMANTER MACHEN

Es gibt die Möglichkeit, Funktionen performanter zu machen, indem – je nach Anwendungsfall – folgende Attribute hinzugefügt werden:

- IMMUTABLE** wenn es immer die gleichen Ergebnisse geben wird – maximale Beschleunigung.
- STABLE** jeweiliger Zugriff auf DB, aber bei mehrfacher Ausführung der Funktion in einem Statement, Optimierung.
- VOLATILE** keine Optimierungen möglich, da jeweils auf die DB zugegriffen wird bzw. werden muss.

Standardeinstellung (wenn kein Attribut vergeben wurde): VOLATILE

```
CREATE FUNCTION get_customer(integer) RETURNS  
    TABLE(id integer, email varchar) STABLE AS $$  
    SELECT id, email FROM customers WHERE id = $1  
$$ LANGUAGE SQL;  
  
SELECT get_customer(10), get_customer(10);
```

VOLATILE oder STABLE sinnvoll,
weil Parameter sich ändern
kann.

IMMUTABLE wäre für CONCAT-
Funktion geeignet

PRIVILEGIEN STEUERN

Über die Attribute SECURITY INVOKER und SECURITY DEFINER kann die Ausführbarkeit von Funktionen im Zusammenspiel mit dem Berechtigungskonzept gesteuert werden.

SECURITY INVOKER die geltenden Berechtigungen des ausführenden Users gelten

SECURITY DEFINER die Berechtigungen desjenigen, der die FUNCTION erstellt hat, gelten

Potenzial: Auch User, die normalerweise keine Zugriffsberechtigung auf Tabellen oder Spalten haben, können die Funktion ausführen.

```
CREATE FUNCTION get_customer(integer)
  RETURNS TABLE(id integer, email varchar)
  SECURITY INVOKER
  VOLATILE AS $$
  SELECT id, email FROM customers WHERE id = $1
$$ LANGUAGE SQL;
SELECT * FROM get_customer(10);
```

FAST SCHON EINE PROCEDURE

- FUNCTIONS können erweitert werden um mehrere Operationen – wie z.B. INSERT-Anweisungen

Beispiel:

Mit Aufruf der Funktion soll in eine Log-Datei ein Eintrag (Timestamp) erfolgen. Notwendige Vorbereitungen:

1. Anlegen der Tabelle mit `<id>` und `<timestamp>` → `<customer_logs>`
2. Anpassen der Funktion zur Listung der Customer → `customer_list_with_logs()`

```
DROP FUNCTION IF EXISTS customer_list_with_logs;
```

```
CREATE FUNCTION customer_list_with_logs() RETURNS TABLE (id bigint, email varchar) AS  
$$
```

```
    INSERT INTO customer_logs (timestamp) VALUES (CURRENT_TIMESTAMP);  
    SELECT id, email FROM customers  
$$ VOLATILE LANGUAGE SQL;
```

ÜBUNG

- Erstellen Sie eine Funktion, die einen einzelnen Kunden aufruft.
- Die Ausführung der Funktion soll in der <customer_logs>-Tabelle dokumentiert werden mit:
 - Wann die Funktion ausgeführt wurde
 - Welcher Kunde aufgerufen worden ist
 - Wer die Funktion aufgerufen hat
- Name der Funktion: customer_logs_single()

LÖSUNG ZUR ÜBUNG

```
ALTER TABLE customer_logs ADD COLUMN akt_user VARCHAR(100);  
ALTER TABLE customer_logs ADD COLUMN customer_id INTEGER DEFAULT 0;  
DROP FUNCTION IF EXISTS customer_logs_single(bigint);  
CREATE FUNCTION customer_logs_single(bigint)  
  RETURNS TABLE (id bigint, email varchar) AS $$  
  INSERT INTO customer_logs (timestamp, akt_user, customer_id)  
    VALUES (CURRENT_TIMESTAMP, CURRENT_USER, $1);  
  SELECT id, email FROM customers WHERE id = $1  
  $$ VOLATILE LANGUAGE SQL;
```

```
SELECT * FROM customer_logs_single(15);
```

```
SELECT * FROM customer_logs;
```

	id [PK] integer	timestamp timestamp with time zone	akt_user character varying (100)	customer_id integer	bemerkung character varying (1000)
1	12	2024-11-26 22:09:55.656205+01	postgres	200	[null]
2	13	2024-11-26 22:10:09.640164+01	postgres	200	[null]
3	14	2024-11-26 22:10:11.504594+01	postgres	200	[null]
4	15	2024-11-26 22:10:47.795209+01	postgres	100	[null]
5	16	2024-11-26 22:10:53.137567+01	postgres	101	[null]
6	17	2024-11-26 22:14:16.733391+01	postgres	101	[null]
7	18	2024-11-26 22:14:31.633971+01	postgres	10	[null]
8	19	2024-11-26 22:17:14.133834+01	postgres	0	ungültige Abfrage mit \$1

ÜBUNG

Aufgabe:

Es sollen am heutigen Tag alle Kunden mit ihrem aktuellen Gesamtumsatz, Timestamp und Stadt in eine Tabelle `<umsatz_monitor>` eingetragen werden, die in einer Stadt leben, die in der Tabelle `<city_auswahl>` eingetragen ist.

Erstellen Sie hierzu die geeignete Stored Function.

STORED FUNCTION – CITY-ABGLEICH

```
DROP FUNCTION IF EXISTS get_customer_city();
```

```
CREATE FUNCTION get_customer_city() RETURNS SETOF customers AS  
$BODY$
```

```
DECLARE
```

```
  r customers%rowtype;
```

```
BEGIN
```

```
  DELETE FROM umsatz_monitor WHERE
```

```
    to_char(timestamp,'YYYY-MM-DD') = to_char(NOW(),'YYYY-MM-DD');
```

```
  FOR r IN
```

```
    SELECT c.*,
```

```
      (SELECT SUM(o.amount) AS umsatz FROM orders AS o WHERE c.id = o.customer_id
```

```
        AND to_char(timestamp,'YYYY-MM-DD') = to_char(NOW(),'YYYY-MM-DD')) AS gesamtumsatz FROM customers AS c
```

```
        WHERE c.id > 0
```

```
        ORDER BY umsatz DESC, c.lastname, c.firstname
```

```
  LOOP
```

```
    IF r.city IN (SELECT city FROM city_auswahl) THEN
```

```
      INSERT INTO umsatz_monitor(customer_id, amount)
```

```
        VALUES (r.id, r.amount);
```

```
      RETURN NEXT r;
```

```
    --ELSE
```

```
  END IF;
```

```
END LOOP;
```

```
  RETURN;
```

```
END;
```

```
$BODY$ LANGUAGE plpgsql;
```

	monitor_id [PK] bigint	erfassdat timestamp with time zone	customer_id integer	amount numeric (10,2)
1	17	2024-11-29 22:02:03.457689+01	162	439.00
2	18	2024-11-29 22:02:03.457689+01	173	439.00
3	19	2024-11-29 22:02:03.457689+01	175	324.00
4	20	2024-11-29 22:02:03.457689+01	53	310.00

```
SELECT lastname,firstname,city,amount  
FROM get_customer_city()  
WHERE amount IS NOT NULL  
ORDER BY amount DESC;
```

	ca_id [PK] integer	city character varying (100)
1	1	Schwerin
2	2	Hamburg
3	3	München
4	4	Augsburg
5	5	Berlin
6	6	Kulmbach



STORED PROCEDURES

PROGRAMME MIT PGSQL

WARUM STORED PROCEDURES ?

```
CREATE FUNCTION customer_logs_single(bigint)
  RETURNS TABLE (id bigint, email varchar) AS $$
  INSERT INTO customer_logs (timestamp, akt_user, customer_id)
    VALUES (CURRENT_TIMESTAMP, CURRENT_USER, $1);
  SELECT id, email FROM customers WHERE id = $1
  $$ VOLATILE LANGUAGE SQL;

SELECT * FROM customer_logs_single(1500); -- Was passiert, wenn nach nicht vorhandener id gesucht wird?
```

- Wir erhalten keine Fehlermeldung.
- Es gibt keinen Sicherungsmechanismus, der derartige Fälle „abfängt“ und damit umgeht.
- Mit PL/pgSQL (Procedural Language for PostgreSQL) können Prozeduren, Funktionen und Trigger entwickelt werden.
- PL/pgSQL erweitert SQL um Kontrollstrukturen.
- Ermöglicht komplexere Computerverarbeitungen ...

GRUNDSTRUKTUR PL/PGSQL-FUNKTION

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
```

```
DECLARE
```

```
    variable := 0;
```

```
BEGIN
```

```
    -- Subblock
```

```
    DECLARE
```

```
        var2 := 100;
```

```
    BEGIN
```

```
        Anweisungen
```

```
    END
```

```
    RETURN variable;
```

```
END
```

```
$$ LANGUAGE plpgsql;
```

ANPASSEN UNSERER FUNCTION

```
DROP FUNCTION customer_logs_single(bigint);

CREATE FUNCTION customer_logs_single(bigint)
  RETURNS TABLE (id bigint, email varchar) AS $$
  DECLARE
    customer_exists bigint := 0;
  BEGIN
    SELECT COUNT(*) INTO customer_exists FROM customers WHERE customers.id = $1;
    IF customer_exists <> 0 THEN
      INSERT INTO customer_logs (timestamp, akt_user, customer_id)
        VALUES (CURRENT_TIMESTAMP, CURRENT_USER, $1);
      RETURN QUERY SELECT c.id, c.email FROM customers AS c WHERE c.id = $1;
    END IF;
  END
  $$ VOLATILE LANGUAGE plpgsql;
```

ELSE – EINE KLEINE ERWEITERUNG (SONST-FÄLLE DOKUMENTIEREN)

ALTER TABLE customer_logs ADD COLUMN bemerkung VARCHAR(1000); – Anpassung der Log-Datei

DROP FUNCTION customer_logs_single(bigint);

CREATE FUNCTION customer_logs_single(bigint)
RETURNS TABLE (id bigint, email varchar) AS \$\$

DECLARE

customer_exists bigint := 0;

BEGIN

SELECT COUNT(*) INTO customer_exists FROM customers AS c WHERE c.id = \$1;

IF customer_exists <> 0 THEN

INSERT INTO customer_logs (timestamp, akt_user, customer_id)

VALUES (CURRENT_TIMESTAMP, CURRENT_USER, \$1);

RETURN QUERY SELECT c.id, c.email FROM customers AS c WHERE c.id = \$1;

ELSE

INSERT INTO customer_logs (timestamp, akt_user, bemerkung)

VALUES (CURRENT_TIMESTAMP, CURRENT_USER, 'ungültige Abfrage mit customer_id: ' || \$1);

END IF;

END

\$\$ VOLATILE LANGUAGE plpgsql;

1. PROCEDURE

```
DROP PROCEDURE IF EXISTS get_number_procedure(bigint);
```

```
CREATE PROCEDURE get_number_procedure(bigint) AS  
$BODY$
```

```
    DECLARE
```

```
        i bigint := $1;
```

```
    BEGIN
```

```
        IF i < 10 THEN
```

```
            RAISE NOTICE 'Kleiner';
```

```
        ELSE
```

```
            RAISE NOTICE 'Größer';
```

```
        END IF;
```

```
        RETURN;
```

```
    END;
```

```
$BODY$
```

```
LANGUAGE plpgsql;
```

```
CALL get_number_procedure(100)
```

Ist das logisch richtig?

ÜBUNG: WEIHNACHTSGESCHENK

Aufgabe:

Es soll ein Kunde abgefragt werden können.

Es soll der Weihnachtsgeschenkvermerk angezeigt werden.

Die Bemerkung richtet sich nach dem summierten Bestellwert des Kunden:

Wenn der Umsatz > 500,00 € → „Großes Weihnachtsgeschenk“

Wenn der Umsatz > 250,00 € → „Mittleres Weihnachtsgeschenk“

SONST → „Kleines Weihnachtsgeschenk“

LÖSUNG: WEIHNACHTSGESCHENK

```
CREATE PROCEDURE pro_customer_weihnachtsgeschenk(bigint)
AS $$
DECLARE
    customer_exists bigint := 0;
    sum_bestellwert numeric := 0;
BEGIN
    SELECT COUNT(amount) INTO customer_exists FROM orders WHERE orders.id = $1;
    IF customer_exists > 0 THEN
        SELECT SUM(amount) INTO sum_bestellwert FROM orders WHERE customer_id = $1;
        IF sum_bestellwert > 500 THEN
            RAISE NOTICE 'Großes Weihnachtsgeschenk';
        ELSEIF sum_bestellwert > 250 THEN
            RAISE NOTICE 'Mittleres Weihnachtsgeschenk';
        ELSE
            RAISE NOTICE 'Kleines Weihnachtsgeschenk';
        --RETURN QUERY SELECT c.id, c.email FROM customers AS c WHERE c.id = $1;
        END IF;
    ELSE
        RAISE NOTICE 'Kunde existiert nicht';
    END IF;
    RETURN;
END; $$ LANGUAGE plpgsql;
```

```
CALL pro_customer_weihnachtsgeschenk(10) -- 194 - groß#
```

FOR r IN – SCHLEIFE

- Wird genutzt zur Ausgabe von (gefilterten) Daten aus einer Abfrage
- Es werden immer alle Spalten zurückgegeben, auch wenn das SELECT in Funktion nur wenige Spalten anspricht. Nicht in der SELECT-Abfrage enthaltene Spalten werden in der Ausgabe aber mit NULL-Wert repräsentiert.
- Der Spaltentyp kann automatisch erkannt werden.

FOR r IN – SCHLEIFE (RETURN NEXT)

```
DROP FUNCTION IF EXISTS get_all_customer();

CREATE OR REPLACE FUNCTION get_all_customer() RETURNS SETOF customers AS
$BODY$
DECLARE
    r customers%rowtype;
BEGIN
    FOR r IN
        SELECT * FROM customers WHERE id > 0
            ORDER BY lastname, firstname
    LOOP
        -- hier könnten weitere Anweisungen folgen
        RETURN NEXT r;           -- gibt aktuelle Zeile des SELECT aus
    END LOOP;
RETURN;
END;
$BODY$ LANGUAGE plpgsql;

SELECT * FROM get_all_customer();    -- tabellarische Darstellung auch ohne RETURNS TABLE
```

RETURN NEXT & RETURN QUERY

- Die aktuelle Implementierung von RETURN NEXT und RETURN QUERY speichert das gesamte Resultset, bevor es von der Funktion zurückgegeben wird, wie oben beschrieben. Das kann bei großen Resultsets zu Performanceverlusten führen.
- Daten werden auf den Datenträger geschrieben, um eine Speicherauslastung zu vermeiden, aber die Funktion selbst kehrt erst zurück, wenn das gesamte Resultset generiert wurde.
- Die Konfigurationsvariable *work_mem* steuert den Zeitpunkt, zu dem auf die Festplatte geschrieben wird. Bei ausreichend Arbeitsspeicher sollte eine Erhöhung dieses Parameters in Betracht gezogen werden (Aufgabe der Administratoren).



TRIGGER

PROGRAMME AUTOMATISCH AUSLÖSEN

GRUNDBAUSTEINE

1. Funktion, die durch den Trigger genutzt wird
2. Trigger selbst

Beispiel:

1. Funktion `update_timestamp()`
→ date `updated_at` up, indem der aufrufende Trigger zurückgegeben wird
2. Trigger `set_updated_at()`
→ löse die Funktion `update_timestamp()`, die auf <tabelle x> wirkt, aus, BEVOR das Update umgesetzt wird.

Diese Reihenfolge ist beim Erstellen und beim Löschen eines Triggers wichtig!!!

Soll ein Trigger gelöscht werden, muss zunächst dieser gelöscht werden.

GRUNDAUFBAU EINER TRIGGERFUNKTION

```
CREATE OR REPLACE FUNCTION Funktionsname()  
RETURNS TRIGGER AS $$  
BEGIN  
    -- Anweisungen  
END;  
$$ LANGUAGE plpgsql;
```

```
DROP FUNCTION IF EXISTS  
Funktionsname();
```

```
CREATE OR REPLACE TRIGGER Triggername  
BEFORE UPDATE ON <tabelle>  
FOR EACH ROW  
EXECUTE FUNCTION Funktionsname();
```

```
DROP TRIGGER IF EXISTS  
Triggername ON <tabelle>  
-- [CASCADE | RESTRICT];
```


UPDATES DOKUMENTIEREN

```
CREATE OR REPLACE FUNCTION update_timestamp()  
RETURNS TRIGGER AS $$  
BEGIN  
    NEW.updated_at = CURRENT_TIMESTAMP;  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE TRIGGER set_updated_at  
BEFORE UPDATE ON orders  
FOR EACH ROW  
EXECUTE FUNCTION update_timestamp();
```

ÜBUNG: ERFASSEN NEUEN DATENSATZES IN LOG-DATEI DOKUMENTIEREN

```
DROP FUNCTION IF EXISTS
insert_timestamp();

CREATE OR REPLACE FUNCTION
insert_timestamp()
RETURNS TRIGGER AS $CODE$
BEGIN
INSERT INTO orders_log
    (akt_user,inserted_at,customer_id)
VALUES
    (CURRENT_USER, CURRENT_TIMESTAMP,
    NEW.customer_id);
RETURN NEW;
END
$CODE$ LANGUAGE plpgsql;
```

```
DROP TRIGGER IF EXISTS insert_log
ON orders;

CREATE OR REPLACE TRIGGER insert_log
BEFORE INSERT ON orders
FOR EACH ROW
EXECUTE FUNCTION insert_timestamp();

INSERT INTO orders(customer_id,
    amount,timestamp)
VALUES
    (194,71.00,CURRENT_TIMESTAMP);
```