

Structured Query Language (SQL)

Warum SQL?

- Datenbanken und SQL sind aus der Informatik nicht mehr wegzudenken und bilden die Basis für alle Datenverarbeitungsprozesse inklusive:
 - Speichern von Daten
 - Analysieren von Daten
 - Visualisieren von Daten
- Alle Daten, auch die Daten die im Internet getauscht, dargestellt und gepostet werden haben als Backend eine Datenbank, die z. B. genutzt wird für:
 - Formulardaten zum Einkaufen, Registrieren und Anmelden
 - Content-Management-Systemen
 - Blogs & Social Media
 - Die Daten werden vor dem Speichern in SQL übersetzt und als Transaktion auf ACID überprüft
- SQL ist auch zentraler Bestandteil für neue Felder wie:
 - KI/AI
 - Machine Learning
 - Natural Language Processing
 - Robotik
 - Industrie 4.0

- ...

Was kann SQL?

- SQL erlaubt die Abfrage, das Editieren, Ändern und Löschen von Informationen in einer Datenbank
- SQL ist standardisiert und funktioniert (fast) auf jedem Datenbanksystem gleich
- Der Standard wird festgelegt von der American National Standards Institute (ANSI) in 1986 und von der International Organization for Standardization (ISO) in 1987
 - Standardisierung ermöglicht eine weite Verbreitung und Nutzung -> nicht jedes System hat seinen eigenen Standard, da Kompatibilität gewahrt bleibt
- Allerdings hat jedes Datenbanksystem auch immer noch Abweichungen (flavours, dialects) und eigene Implementationen
- SQL gehört zur 4. Generation (4GL) an Programmiersprachen und ist rein deklarativ:
 - Daten werden angefordert, man hat aber keinen Einfluss, wie diese Daten abgeholt werden
 - Diese Logik ist komplett dem RDMS überlassen
 - Unter der Haube arbeitet ein Query Optimiser, der Anfragen optimiert und sie im RAM bereitstellt
 - Auch wo und wie die Daten physisch gespeichert werden, ist komplett dem RDBMS überlassen
 - dies erschwert einen einfachen Austausch der Daten wiederum

Wie "spreche" ich mit der Datenbank?

- Datenbankmanagementsysteme erlauben in der Regel den Zugriff über:
 - eine graphische Benutzeroberfläche (GUI)
 - für PostgreSQL ist das pgAdmin
 - über die Kommandozeile: `psql -h localhost postgres postgres`
 - über eine Schnittstelle (API, z. B. pyodbc für Python)
- Für dieses Modul beschränken wir uns auf die GUI -> pgAdmin

SQL auf einen Blick

Statement	Zugehörig
SELECT	Data retrieval (Abfrage)
CREATE ALTER DROP	Data definition language (DDL)
INSERT UPDATE DELETE MERGE TRUNCATE	Data manipulation language (DML)
COMMIT ROLLBACK SAVEPOINT	Transaction control

Statement	Zugehörig
GRANT REVOKE	Data control language (DCL)

- ✓ Dies sind bereits alle möglichen SQL-Statements
- Data definition language (DDL) ermöglicht das Erstellen oder Ändern des Datenmodells
- Data Manipulation Language (DML) bezieht sich auf die Daten selbst, also das Hinzufügen, Ändern oder Löschen von Daten
- Transaction control erzwingt die Konsistenz beim Schreiben der Daten
- Data Control Language (DCL) richtet die Zugangskontrolle für Datenbankobjekte ein. Über Nutzer und Rollen kann ein sehr filigraner Zugriffsschutz erstellt werden

Data Definition Language (DDL)

Wie erstelle ich ein Datenmodell?

An Anfang muss die Grundstruktur (Datenmodell) geschaffen werden, um Daten aufzunehmen. Später kann das Datenmodell auch geändert werden:

- Das Datenmodell kann:
 - Tabellen erzeugen (CREATE TABLE)
 - Tabellen ändern (ALTER TABLE)
 - Tabellen löschen (DROP TABLE)

Tabellen erzeugen

- Tabellen zu erzeugen ist der Anfang aller Datenmodelle
- Grundsätzlich wird es durch das **CREATE TABLE** Statement erstellt:
 - der Name der Tabelle
 - Spaltennamen und Datentypen
 - Konsistenzprüfungen (Constraints) definiert
 - Constraints können sein:
 - CHECK (Domäne = Spalte)
 - PRIMARY KEY (Entität = Tabelle)
 - FOREIGN KEY (referentiell = zu anderen Tabellen)

```
--erzeugt Tabelle mit Spalten, Datentypen & Integritätsbedingungen
CREATE TABLE tabelle (
    id int PRIMARY KEY,          --PRIMARY KEY (Entität)
    vorname varchar(50) NOT NULL, -- CHECK CONSTRAINT (Domäne)
    nachname varchar(50) NOT NULL,
    abteilungs_id smallint REFERENCES abteilung (id) --FOREIGN KEY
(referentiell)
);
```

mehr hierzu: CREATE TABLE

- Tabellen können immer geändert werden, z. B.:
 - Spalten ergänzt oder vom Datentyp geändert
 - Integritätsbedingungen ergänzt oder verändert werden
 - ... etc.

```
ALTER TABLE table_name [ADD | DROP | ALTER] [COLUMN | CONSTRAINT]
```

- oder gelöscht werden:

```
DROP TABLE table_name [CASCADE] --CASCADE = Lösche auch Objekte die vom gelöschten Objekt abhängen
```

Data Manipulation Language

Wie ändere ich Daten in den Tabellen?

Nachdem das Datenmodell erstellt ist, muss es mit Leben (Daten) gefüllt werden:

- Daten können:
 - eingefügt (INSERT)
 - aktualisiert (UPDATE) oder
 - gelöscht (DELETE) werden

Wie kriege ich Daten in die Datenbank?

```
--alle Spalten gegeben
INSERT INTO tabelle VALUES (1, 'Wert 1', 'Wert 2');

--mit Spaltenauswahl
INSERT INTO (col2, col4) VALUES (1, 'Wert 1')

-- Mehrere Zeilen
INSERT INTO tabelle VALUES
(1, 'Wert 1'),
(2, 'Wert 2')
```

```
--Tabelle muss bereits bestehen
COPY
```

```
-- Neue Tabellen aus bestehenden Tabellen erzeugen
CREATE TABLE AS SELECT * FROM tabelle

SELECT * INTO [table_name] FROM ...
```

Automatisiert

psql

IDE, z. B. python (import pyodbc, import psycopg)

```
UPDATE tabelle SET spalte = 'Wert' WHERE spalte = 'Wert'
```

```
DELETE FROM tabelle WHERE spalte = 'Wert'
```

Aufgabe 1

! Die folgende Aufgabe enthält zunächst die Erstellung eines konzeptionellen Datenbank-Modells (ERM), das wir in SQL transformieren und aus dem wir ein physisches Modell generieren und mit Beispieldaten füllen und Integritätsverletzungen provozieren. Bitte geben Sie für die Aufgabe ein Lösungsdokument im *.docx oder *.pdf Format ab, aus dem ALLE wichtigen Schritte hervorgehen.



Erstellen Sie über pgAdmin und seinem ERD-Tool ein Datenmodell, das eine Anwendung in ihrem Studium oder ähnliches implementiert. Was Sie modellieren können Sie beliebig wählen. Achten Sie aber bitte darauf, folgende Punkte zu integrieren:

1. Das Datenmodell enthält ca. 5 Tabellen
 - Achten Sie auf die richtige Vergabe des Datentyps
 - Jede Tabelle sollte auch einen Primärschlüssel haben
2. mindestens **eine** 1:n, n:m Beziehung zwischen den Tabellen
 - Setzen Sie hier die referentielle Integrität für die Beziehungen zwischen den Tabellen
3. Setzen Sie in ihrem Modell auch mindestens **eine** der 3 Integritätsbedingungen in den Tabellen um, d. h.:
 - Domänenintegrität
 - Primärschlüssel (für alle Tabellen)
 - Fremdschlüssel (für die 1:n & n:m Beziehungen)
4. Lassen Sie sich aus dem graphischen Modell SQL generieren (Generate SQL) und spielen Sie diese Statements zurück in eine Datenbank
5. Füllen Sie die Tabellen mit ein paar Beispieldatensätzen (hier reichen wenige)
 - Provozieren Sie nun für jeden INSERT INTO Befehl eine Verletzung der Integritätsbedingungen. Wie sind die Fehlermeldungen?

Datenabfrage (SELECT)


- SELECT ist ein sehr mächtiger Befehl, um Daten aus Tabellen abzufragen
- Wenn die Daten bereits definiert sind (DDL), bewegt man sich fast ausschließlich mit diesem Befehl, um Daten zu analysieren und auszuwerten
- Daten können sehr effizient zusammengefügt und verbunden werden.
 - Die Analyse von Daten wird im Vergleich zu z. B. Excel wesentlich flexibler und einfacher

```
-- Ich bin ein Kommentar  
SELECT * FROM products;
```

-  Zur Einführung ein Code-Beispiel, wie wir es auf den nächsten Seiten öfter sehen werden
-  Ein wichtiger Einstieg in SQL ist die Selektion von Spalten und Zeilen, fangen wir mit den Spalten an:

```
-- Wie selektiere ich Spalten?  
SELECT product_name, quantity_per_unit FROM products; --product_name =  
Spalte  
  
SELECT * FROM products --alle Spalten
```

Operatoren

- Wie selektiere (filtere) ich Zeilen? :
- mit dem WHERE statement:

```
SELECT * FROM products  
WHERE spalten_name = 'Wert'
```

Arithmetisch

Operator	Bedeutung
=	ist gleich
<	kleiner
<=	kleiner gleich
>	größer
>=	größer gleich
<> oder !=	ungleich

Null-Werte

Operator	Bedeutung
is Null	Null-Werte
is not Null	nicht Null-Werte

-  Null-Werte stehen für unbekannte Werte

```
-- Alle Produkte größer gleich $50
SELECT * FROM products
WHERE unit_price >= 50
```

AND, NOT, OR

Operator	Beschreibung
AND	Alle Bedingungen müssen erfüllt sein
OR	Nur eine Bedingung muss erfüllt sein
NOT	Negation

Beispiele


```
-- Alle Produkte, die mit "A" anfangen UND über 50 $ kosten
SELECT * FROM products WHERE product_name LIKE 'A%' and unit_price > 50
-- Alle Produkte, die **nicht** mit "B" anfangen
SELECT * FROM products WHERE product_name NOT LIKE ('B%')
```

```
-- Welche Produkte kosten über $50?
SELECT product_id, product_name
FROM products
WHERE unit_price >= 50;
```

- Eine Besonderheit im **WHERE** Keyword ist das Filtern mit **LIKE**:


```
SELECT * FROM products
-- 1) Mit "A" startet
WHERE productname LIKE 'A%'
-- 2) "A" enthält
WHERE productname LIKE '%a%'
-- 3) Mit "A" endet
WHERE productname LIKE 'a%'
```

Aggregate Functions

- Aggregatfunktionen sind Funktionen, die über alle oder bestimmte Spalten aggregieren
-  Beispiele siehe **GROUP BY** oben



```
-- Was ist das teuerste Produkt? (Aggregation auf gesamte Tabelle)
SELECT MAX(unit_price)
FROM products;
```

```
-- Was ist der Durchschnittspreis pro Händler? (Aggregation auf eine Spalte (supplier_id))
SELECT supplier_id, AVG(unit_price)
FROM products
GROUP BY supplier_id
ORDER BY avg
```

-  **!** Jede Spalte, die im **SELECT** Keyword auftaucht (außer der Aggregationsfunktion selbst), muss auch im **GROUP BY** Keyword vorkommen

```
-- Geht das bitte mit aufgelöstem Händlername?
SELECT company_name, avg FROM suppliers
LEFT JOIN
(SELECT supplier_id, AVG(unit_price)
FROM products
GROUP BY supplier_id
ORDER BY avg) ave_price ON suppliers.supplier_id=ave_price.supplier_id
```

	company_name character varying (40)	 avg double precision	
1	Exotic Liquids		14.5
2	New Orleans Cajun Delights	20.34999990463257	
3	Grandma Kelly's Homestead	31.666666666666668	

-   Ein SELECT-Statement kann insgesamt folgende Keywords enthalten:

Keyword	Beschreibung
SELECT	Filtert Spalten ("*" für alle)
FROM	Tabelle
WHERE	Filtert Zeilen


Keyword	Beschreibung
GROUP BY	Ermöglicht das Aggregieren auf Spalten, z. B. mit SUM(), MAX(), MIN(), COUNT(), AVG()
HAVING	Filtert wieder Zeilen nach dem Aggregieren
ORDER BY	Sortiert Ergebnis nach Spalte
LIMIT	Limitiert die Ergebnisse auf eine bestimmte Anzahl, z. B. 100

SQL JOINS


```

flowchart LR
    markdown["Tabelle 1"]
    newLines["Tabelle 2"]
    markdown --> newLines
    newLines --> markdown

```

-  Tabellen werden hier horizontal verbunden, d.h. die Spaltenanzahl erhöht sich bis auf alle Spalten von beiden Tabellen (solange keine Spaltenselektion vorgenommen wird)
- JOINS verbinden Tabellen (in der Regel) auf einen bestimmten Schlüssel -> referentielle Integrität
- So können Daten wieder denormalisiert werden und lesbar gemacht werden.
- Wir erinnern uns: Datentabellen enthalten nur Schlüssel, ähnlich wie hier (order_details):

customer_id	product_id	Kaufdatum
1	14	01.04.2024
2	16	03.01.2025

-  Um Daten wieder lesbar zu machen, müssen Sie über einen JOIN wieder verknüpft werden, d. h. die Schlüssel (hier Fremdschlüssel) der Primärtabelle angehängt werden

```

--Welche Firma (customer) hat welche Produkte gekauft?
SELECT company_name, product_name FROM order_details
LEFT JOIN orders ON order_details.order_id=orders.order_id
LEFT JOIN customers ON orders.customer_id=customers.customer_id
LEFT JOIN products on order_details.product_id=products.product_id
WHERE order_details.order_id=10248

```



- JOINS können INNER, OUTER oder CROSS sein

 JOINS Source: <https://www.linkedin.com/pulse/sql-inner-join-tutorial-matt-l>

SET Operatoren

SET operators

```
graph TD;
  tab1["Tabelle 1"]
  tab2["Tabelle 2"]
  tab1 --> tab2;
  tab2 --> tab1;
```

-  SET Operatoren hängen Tabellen zusammen bzw. finden die Differenz in den Zeilen
-  Hierfür müssen die Tabellen, die exakt gleiche Anzahl an Spalten und gleiche Datentypen haben
- SET Operatoren verbinden Tabellen vertikal, sie erhöhen oder vermindern die Zeilenanzahl

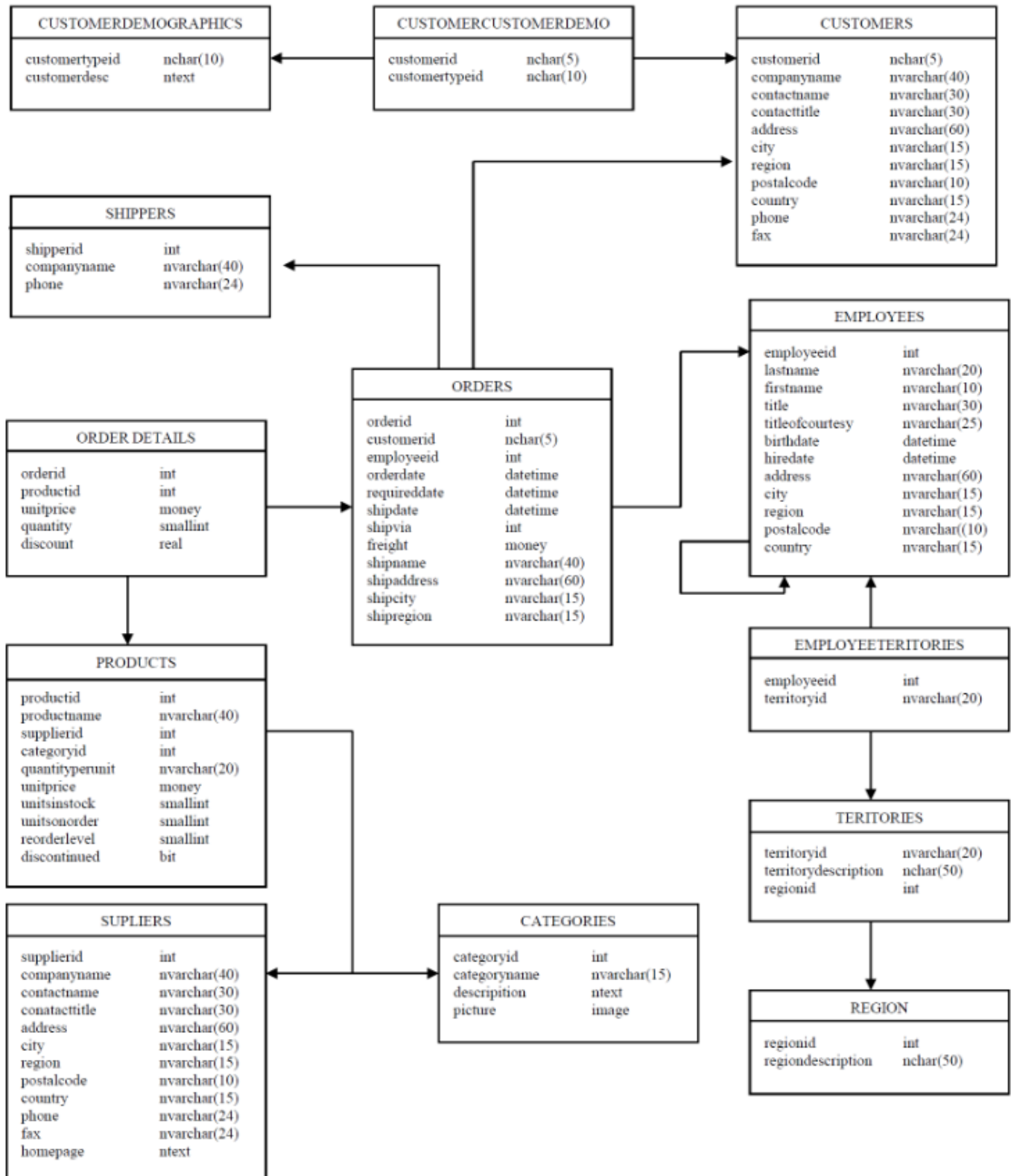
Operator	Bedeutung
UNION oder UNION ALL	Hängt 2 Tabellen aneinander, UNION ALL erlaubt Duplikate, UNION entfernt diese
INTERSECT	gibt die Menge der überschneidenden Elemente zurück (d. h. sowohl in Tabelle 1 als auch 2 enthalten)
MINUS	findet die Menge, die nur in der einen, nicht aber in der anderen Tabelle ist

```
SELECT product_name, quantity_per_unit, unit_price FROM products
INTERSECT
SELECT product_name, quantity_per_unit, unit_price FROM products
WHERE product_id != 1
```

Unterabfragen, Common Table Expressions (CTE) und temporäre Tabellen

- Es gibt Fälle bei komplexen Abfragen, in der man Zwischenergebnisse generieren kann, die man leichter verstehen und analysieren kann

Ein kleines Beispiel. Gegeben ist das ERM-Schema der Northwind-Datenbank:



Die Firma will einen Bonus für jeden Mitarbeiter (Tabelle employees) einen Bonus ausschütten und zwar abhängig von der Summe der verkauften Produkte (sum(unit_price) in Tabelle order_details)

Was müssen wir dafür tun? :thinking:

Dies kann tatsächlich auf vielen Wegen gelöst werden. Aufgrund der Komplexität der JOINS zwischen 3 Tabellen, sollte auch auf Hilfswerkzeuge, d. h. Zwischenergebnisse, zurückgegriffen werden. Folgend, das konkrete Beispiel mit einer

- a) Unterabfrage, einer

- b) Common Table Expression (CTE) und einer
- c) temporären Tabelle:

-- a) mit einer Unterabfrage

```
SELECT first_name, last_name, SUM(unit_price)
FROM
(SELECT * FROM employees AS e
LEFT JOIN orders AS o ON e.employee_id = o.employee_id
LEFT JOIN order_details AS od ON o.order_id = od.order_id
WHERE date_part('year', order_date) = 1997) as foo
GROUP BY first_name, last_name
ORDER BY sum DESC
```

-- b) mit einer CTE (Common Table Expression)

```
WITH base_query AS (
SELECT * FROM employees AS e
LEFT JOIN orders AS o ON e.employee_id = o.employee_id
LEFT JOIN order_details AS od ON o.order_id = od.order_id
WHERE date_part('year', order_date) = 1997
)
SELECT first_name, last_name, SUM(unit_price)
FROM base_query
GROUP BY first_name, last_name
ORDER BY sum DESC
```

-- c) mit einer temporären Tabelle

```
CREATE TEMP TABLE base_query AS
SELECT first_name, last_name, unit_price FROM employees AS e
LEFT JOIN orders AS o ON e.employee_id = o.employee_id
LEFT JOIN order_details AS od ON o.order_id = od.order_id
WHERE date_part('year', order_date) = 1997;
```

```
SELECT first_name, last_name, AVG(unit_price)
FROM base_query
GROUP BY first_name, last_name;
```

-- temporäre Tabellen existieren zwar nur zur Laufzeit, aber doch empfohlen sie zu löschen

```
DROP TABLE base_query;
```

DATA CONTROL LANGUAGE

- Natürlich darf nicht jeder auf **alle** Daten zurückgreifen, diese müssen und können sehr feinteilig gegliedert werden:

GRANT
REVOKE