Session 7 – SQL Introduction & SELECT Statements

Session-Typ: Lecture **Dauer:** 90 Minuten **Lernziele:** LZ 2 (SQL-Praxis)

Datenbank vorbereiten

Bevor wir mit SQL-Abfragen starten, laden wir unsere Produktdatenbank. Wir nutzen die Products-Tabelle aus der CSV-Datei mit 418 Elektronikartikeln, Kleidung, Lebensmitteln und Büromaterial. Diese Daten sind unser Spielplatz für alle SELECT-Statements in dieser Session.

```sql
INSTALL httpfs;
LOAD httpfs;

-- 1) Load raw data into a temp table
CREATE TEMP TABLE products_raw AS
SELECT *
FROM read_json('https://raw.githubusercontent.com/andre-dietrich
  /Datenbankensysteme-Vorlesung/refs/heads/main/assets/dat/products.j

-- 2) Create final table with a PK defined at creation
CREATE TABLE products (
  product_id INTEGER PRIMARY KEY,
  name TEXT,
  category TEXT,
  brand TEXT,
  price REAL,
  stock INTEGER,
  rating REAL,
  created_at DATE
);

-- 3) Insert with proper field mappings & casting
INSERT INTO products
SELECT
  CAST(product_id AS INTEGER) AS product_id,
  name::TEXT,
  category::TEXT,
  brand::TEXT,
  CAST(price AS REAL) AS price,
  CAST(stock AS INTEGER) AS stock,
  CAST(rating AS REAL) AS rating,
  CAST(created_at AS DATE) AS created_at
FROM products_raw;
```

```
INSTALL httpfs
```

No data

```
LOAD httpfs
```

No data

```
-- 1) Load raw data into a temp table
CREATE TEMP TABLE products_raw AS
SELECT *
FROM read_json('https://raw.githubusercontent.com/andre-
dietrich/Datenbankensysteme-
Vorlesung/refs/heads/main/assets/dat/products.json')
```

	Count
1	932

```
-- 2) Create final table with a PK defined at creation
CREATE TABLE products (
  product_id INTEGER PRIMARY KEY,
  name TEXT,
  category TEXT,
  brand TEXT,
  price REAL,
  stock INTEGER,
  rating REAL,
  created_at DATE
)
```

No data

```
-- 3) Insert with proper field mappings & casting
INSERT INTO products
SELECT
  CAST(product_id AS INTEGER) AS product_id,
  name::TEXT,
  category::TEXT,
  brand::TEXT,
  CAST(price AS REAL) AS price,
  CAST(stock AS INTEGER) AS stock,
```

```
  CAST(rating AS REAL) AS rating,
  CAST(created_at AS DATE) AS created_at
FROM products_raw
```

	Count
1	932

Falls das laden über URL nicht funktioniert, hier der Code zum manuellen Laden der Datei:

```
const res  = await fetch('https://raw.githubusercontent.com/andre-die
  /Datenbankensysteme-Vorlesung/refs/heads/main/assets/dat/products.j
const text = await res.text();

// als "Datei" in DuckDB registrieren
await db.registerFileText('products.json', text);

// jetzt normal aus der "lokalen" Datei lesen
await conn.query(`
CREATE TEMP TABLE Products_raw AS SELECT * FROM read_json('products.j

-- 2) Create final table with a PK defined at creation
CREATE TABLE products (
  product_id INTEGER PRIMARY KEY,
  name TEXT,
  category TEXT,
  brand TEXT,
  price REAL,
  stock INTEGER,
  rating REAL,
  created_at DATE
);

-- 3) Insert with proper field mappings & casting
INSERT INTO products
SELECT
  CAST(product_id AS INTEGER) AS product_id,
  name::TEXT,
  category::TEXT,
  brand::TEXT,
  CAST(price AS REAL) AS price,
  CAST(stock AS INTEGER) AS stock,
  CAST(rating AS REAL) AS rating,
  CAST(created_at AS DATE) AS created_at
FROM products_raw;

`);
```

```
38 console.log("ready")
```

```
ready
```

```
1 -- Kurzer Blick auf die Daten
2 SELECT * FROM Products LIMIT 5;
```

	product_id	name	category	brand	price	stock	rating
1	1	Laptop 14" 2.0	Electronics	PixelPeak	1399.030029296875	138	3.700000047
2	2	Portable SSD Max M	Electronics	ZenCore	805.9299926757812	211	3.900000095
3	3	4K Monitor - Green	Electronics	Neutrino	522.47998046875	261	5
4	4	Tablet 10	Electronics	Voltix	985.75	54	3.900000095
5	5	Portable SSD Lite	Electronics	OrbiTech	508.2300109863281	20	4.199999809

Was ist SQL?

Bevor wir in die praktischen Abfragen eintauchen, klären wir die Grundlagen: Was ist SQL überhaupt? SQL steht für Structured Query Language – eine deklarative Sprache, mit der Sie Datenbanken abfragen und manipulieren. Entwickelt wurde SQL in den 1970er Jahren bei IBM für das System R-Projekt. Heute ist es der Standard für relationale Datenbanken weltweit, mit ANSI- und ISO-Standardisierung.

SQL = Structured Query Language

- **Deklarativ:** Sie beschreiben **was** Sie wollen, nicht **wie** die Datenbank es holen soll
- **Standardisiert:** ANSI/ISO SQL – funktioniert auf MySQL, PostgreSQL, SQL Server, Oracle, SQLite, DuckDB
- **Mächtig:** Von einfachen Lookups (`SELECT * FROM ...`) bis zu komplexen Analysen mit Joins, Subqueries, Window Functions

Entwickelt: 1970er bei IBM (System R), erste kommerzielle Version: Oracle 1979

SQL ist keine monolithische Sprache, sondern besteht aus mehreren Komponenten. Die wichtigsten vier sind: DDL für Schema-Definitionen, DML für Datenmanipulation, DCL für Zugriffsrechte und TCL für Transaktionen. In dieser Session fokussieren wir uns auf DML – genauer gesagt auf SELECT, die Königin der SQL-Befehle.

SQL-Komponenten

Kategorie	Abkürzung	Zweck	Beispiele
Data Definition Language	DDL	Schema erstellen/ändern	`CREATE`, `ALTER`, `DROP`
Data Manipulation Language	DML	Daten lesen/schreiben	`SELECT`, `INSERT`, `UPDATE`, `DELETE`
Data Control Language	DCL	Zugriffsrechte	`GRANT`, `REVOKE`
Transaction Control Language	TCL	Transaktionen	`COMMIT`, `ROLLBACK`

Heute fokus: DML – speziell **SELECT**

Warum sollten Sie SQL lernen? Erstens: SQL ist portabel. Ein SQL-Query läuft mit minimalen Anpassungen auf fast jedem relationalen Datenbanksystem. Zweitens: SQL ist deklarativ – Sie sagen „Gib mir alle Produkte unter 50 Euro„, nicht „Öffne die Tabelle, iteriere über alle Zeilen, prüfe den Preis...“. Die Datenbank kümmert sich um die Optimierung. Drittens: SQL ist omnipräsent. Egal ob Sie mit Webdaten, Analytics, IoT oder Machine Learning arbeiten – irgendwo ist SQL im Spiel.

Warum SQL?

✅ **Deklarativ:** Sie beschreiben das „Was„, die Datenbank optimiert das „Wie“

```sql
-- Sie schreiben:
SELECT name, price FROM Products WHERE price < 50;

-- Die Datenbank entscheidet:
-- - Welchen Index nutzen?
-- - Table Scan oder Index Scan?
-- - Parallel execution?
```

✅ **Portabel:** SQL-Standard funktioniert auf vielen Systemen ✅ **Mächtig:** Von einfachen Queries bis komplexe Analysen ✅ **Verbreitet:** Fast jede Datenbank spricht SQL

SELECT & FROM – Die Grundlagen

Jede SQL-Abfrage beginnt mit SELECT und FROM. SELECT definiert, welche Spalten Sie sehen möchten. FROM sagt, aus welcher Tabelle die Daten kommen. Das ist das Fundament. Schauen wir uns die einfachste mögliche Abfrage an: SELECT * FROM Products – zeige mir alle Spalten aus der Products-Tabelle.

Die einfachste Query: SELECT *

Syntax:

```
SELECT * FROM Products;
```

- `SELECT *` = Alle Spalten
- `FROM Products` = Aus der Tabelle „Products"

Aufgabe: Schreiben Sie eine SQL-Abfrage, die **alle Spalten** aus der Tabelle `Products` anzeigt.

```
1  -- Ihre Lösung hier:
2
```

No data

In der Praxis wollen Sie selten ALLE Spalten. Besser ist es, explizit die Spalten zu benennen, die Sie brauchen. Das macht Ihre Query schneller, lesbarer und weniger anfällig für Fehler, wenn sich das Schema ändert. Schauen wir uns an, wie Sie nur die Produkt-ID, den Namen und den Preis abrufen.

Spezifische Spalten auswählen

Syntax:

```
SELECT column1, column2, column3
FROM table_name;
```

Aufgabe: Schreiben Sie eine Abfrage, die nur die Spalten `product_id`, `name` und `price` aus der Tabelle `Products` anzeigt.

```
1  -- Ihre Lösung hier:
2
```

No data

Warum nicht immer `SELECT *`?

- ❌ Langsamer (mehr Daten übertragen)
- ❌ Unlesbar (zu viele Spalten)
- ❌ Wartungsproblem (Schema ändert sich)

✅ **Best Practice:** Spalten explizit benennen
Sie können Spalten auch umbenennen mit AS. Das ist nützlich für Lesbarkeit oder wenn Sie berechnete Spalten erstellen. Zum Beispiel: Zeigen Sie den Preis in Euro statt in der Original-Währung, oder geben Sie der Spalte einen aussagekräftigeren Namen.

Spalten umbenennen mit AS (Aliase)

Syntax:

```
SELECT column_name AS new_name
FROM table_name;
```

Aufgabe: Schreiben Sie eine Abfrage, die folgende Spalten aus der Tabelle `Products` anzeigt und umbenennt: - `product_id` → `id` - `name` → `product_name` - `price` → `price_eur` - `category` (ohne Umbenennung)

```
1  -- Ihre Lösung hier:
2
```

No data

Wann Aliase nutzen?

- Berechnete Spalten: `price * 1.19 AS price_with_tax`
- Kurze Namen: `c.customer_name AS name`
- Lesbarkeit: `SUM(quantity) AS total_sold`

WHERE – Daten filtern

Die WHERE-Klausel ist Ihr Filter. Sie sagt der Datenbank: „Gib mir nur die Zeilen, die diese Bedingung erfüllen." Ohne WHERE bekommen Sie alle Zeilen. Mit WHERE filtern Sie auf genau das, was Sie brauchen. Beginnen wir mit einfachen Vergleichen: Alle Produkte, die weniger als 50 Euro kosten.

Grundlegende Filterung

Syntax:

```
SELECT col_name
```

```
SELECT columns
FROM table
WHERE condition;
```

Aufgabe: Schreiben Sie eine Abfrage, die die Spalten `product_id`, `name` und `price` aus der Tabelle `Products` anzeigt. Filtern Sie nur Produkte, deren Preis **kleiner als 50** ist.

```
-- Ihre Lösung hier:

```

No data

WHERE unterstützt alle Standard-Vergleichsoperatoren: gleich, ungleich, kleiner, größer, kleiner-gleich, größer-gleich. In SQL gibt es zwei Schreibweisen für „ungleich": `<>` (SQL-Standard) und `!=` (aus Programmiersprachen). Beide funktionieren in DuckDB, aber `<>` ist der offizielle Standard.

Vergleichsoperatoren

Operator	Bedeutung	Beispiel
`=`	Gleich	`price = 100`
`<>` oder `!=`	Ungleich	`category <> 'Electronics'`
`<`	Kleiner	`price < 50`
`>`	Größer	`stock > 100`
`<=`	Kleiner oder gleich	`rating <= 3.0`
`>=`	Größer oder gleich	`price >= 1000`

Aufgabe: Schreiben Sie eine Abfrage, die die Spalten `name`, `price` und `category` aus der Tabelle `Products` anzeigt. Filtern Sie nur Produkte, deren Preis **größer oder gleich 1000** ist.

```
-- Ihre Lösung hier:

```

No data

Sie können mehrere Bedingungen kombinieren mit AND und OR. AND bedeutet: Beide Bedingungen müssen erfüllt sein. OR bedeutet: Mindestens eine Bedingung muss erfüllt sein. Achten Sie auf Klammern, wenn Sie AND und OR mischen – die Priorität kann überraschend sein.

Logische Operatoren: AND, OR, NOT

Aufgabe (AND): Schreiben Sie eine Abfrage, die die Spalten `name`, `price`, `category` und `stock` aus der Tabelle `Products` anzeigt. Filtern Sie nur Produkte, deren Preis **kleiner als 100** ist **UND** deren Lagerbestand **größer als 200** ist.

```
1  -- Ihre Lösung hier:
2
```

No data

Aufgabe (OR): Schreiben Sie eine Abfrage, die die Spalten `name`, `price` und `category` aus der Tabelle `Products` anzeigt. Filtern Sie nur Produkte, deren Kategorie **'Electronics'** oder **'Office'** ist.

```
1  -- Ihre Lösung hier:
2
```

No data

Aufgabe (NOT): Schreiben Sie eine Abfrage, die die Spalten `name`, `price` und `category` aus der Tabelle `Products` anzeigt. Filtern Sie nur Produkte, deren Kategorie **NICHT** ‚Groceries' ist.

```
1  -- Ihre Lösung hier:
2
```

No data

⚠️ **Achtung bei gemischten Operatoren:**

```
-- Falsch (ohne Klammern):
WHERE category = 'Electronics' OR category = 'Office' AND price < 100
-- Bedeutet: (Electronics) ODER (Office UND price < 100)

-- Richtig (mit Klammern):
WHERE (category = 'Electronics' OR category = 'Office') AND price < 100
```

```
-- Bedeutet: (Electronics ODER Office) UND (price < 100)
```

Für Bereichsabfragen gibt es BETWEEN. Das ist syntaktischer Zucker für „größer-gleich UND kleiner-gleich". Statt `price >= 100 AND price <= 500` schreiben Sie `price BETWEEN 100 AND 500`. Beides funktioniert, aber BETWEEN ist lesbarer.

BETWEEN – Bereichsabfragen

Syntax:

```
WHERE column BETWEEN value1 AND value2
```

Entspricht: `column >= value1 AND column <= value2`

Aufgabe: Schreiben Sie eine Abfrage, die die Spalten `name`, `price` und `category` aus der Tabelle `Products` anzeigt. Filtern Sie nur Produkte, deren Preis **zwischen 100 und 500** (inklusive) liegt.

```
1  -- Ihre Lösung hier:
2
```

No data

Aufgabe (Datum-Bereiche): Schreiben Sie eine Abfrage, die die Spalten `name`, `price` und `created_at` aus der Tabelle `Products` anzeigt. Filtern Sie nur Produkte, deren Erstellungsdatum **zwischen dem 1. Januar 2025 und dem 30. Juni 2025** liegt. Datumswerte werden im Format ‚YYYY-MM-DD' geschrieben.

```
1  -- Ihre Lösung hier:
2
```

No data

IN ist perfekt für Listen. Statt `category = 'Electronics' OR category = 'Office' OR category = 'Clothing'` schreiben Sie `category IN ('Electronics', 'Office', 'Clothing')`. Das ist kürzer und lesbarer.

IN – Listen-Abfragen

Syntax:

```
WHERE column IN (value1, value2, value3, ...)
```

Aufgabe: Schreiben Sie eine Abfrage, die die Spalten `name`, `price` und `category` aus der Tabelle `Products` anzeigt. Filtern Sie nur Produkte, deren Kategorie **'Electronics'**, **'Office'** oder **'Groceries'** ist.

```
-- Ihre Lösung hier:

```

No data

Aufgabe (NOT IN): Schreiben Sie eine Abfrage, die die Spalten `name`, `price` und `category` aus der Tabelle `Products` anzeigt. Filtern Sie nur Produkte, deren Kategorie **NICHT** ‚Clothing' oder ‚Home & Kitchen' ist.

```
-- Ihre Lösung hier:

```

No data

LIKE ist für Muster-Matching. Sie können nach Produkten suchen, deren Name mit „Laptop„ beginnt, oder die „Pro" irgendwo im Namen haben. Der Prozent-Zeichen-Wildcard `%` steht für „beliebige Zeichen", der Unterstrich `_` für genau ein Zeichen.

LIKE – Pattern Matching

Wildcards:

- `%` = Beliebige Anzahl von Zeichen (0 oder mehr)
- `_` = Genau ein Zeichen

Aufgabe: Schreiben Sie eine Abfrage, die die Spalten `name`, `price` und `category` aus der Tabelle `Products` anzeigt. Filtern Sie nur Produkte, die das Wort **„Laptop"** irgendwo im Namen enthalten.

```
-- Ihre Lösung hier:

```

No data

Aufgabe: Schreiben Sie eine Abfrage, die die Spalten `name`, `price` und `category` aus der Tabelle `Products` anzeigt. Filtern Sie nur Produkte, deren Name **mit „Wireless" beginnt**.

```
-- Ihre Lösung hier:
```

```
2
```

No data

Aufgabe: Schreiben Sie eine Abfrage, die die Spalten `product_id` und `name` aus der Tabelle `Products` anzeigt. Filtern Sie nur Produkte, deren `product_id` dem Muster **'P00_01'** entspricht (exakt 6 Zeichen).

```
-- Ihre Lösung hier:

```

No data

⚠️ **Case Sensitivity:**

- DuckDB: LIKE ist standardmäßig **case-sensitive**
- Für case-insensitive: `ILIKE` (DuckDB, PostgreSQL)

Aufgabe (ILIKE): Schreiben Sie eine Abfrage, die nur die Spalte `name` aus der Tabelle `Products` anzeigt. Filtern Sie nur Produkte, die **„laptop“** (klein geschrieben) irgendwo im Namen enthalten (case-insensitive).

```
-- Ihre Lösung hier:

```

No data

NULL ist ein Spezialfall. NULL bedeutet „Wert unbekannt„ oder „Wert fehlt“. Sie können NULL nicht mit Gleichheit prüfen – `price = NULL` funktioniert nicht. Stattdessen müssen Sie `IS NULL` oder `IS NOT NULL` verwenden.

NULL-Werte behandeln

NULL ist NICHT gleich 0 oder leerer String!

NULL = „Wert unbekannt„ oder „Wert fehlt“

Falsch:

```
-- Funktioniert NICHT:
WHERE rating = NULL
```

Richtig:

```
WHERE rating IS NULL
```

Aufgabe: Schreiben Sie eine Abfrage, die die Spalten `name`, `price` und `rating` aus der Tabelle `Products` anzeigt. Filtern Sie nur Produkte, die **kein Rating** haben (NULL).

```
1  -- Ihre Lösung hier:
2
```

No data

Aufgabe: Schreiben Sie eine Abfrage, die die Spalten `name`, `price` und `rating` aus der Tabelle `Products` anzeigt. Filtern Sie nur Produkte, die **ein Rating** haben (NOT NULL).

```
1  -- Ihre Lösung hier:
2
```

No data

⚠️ **NULL in Vergleichen:**

- `NULL = NULL` → NULL (nicht TRUE!)
- `NULL > 5` → NULL
- `NULL AND TRUE` → NULL

ORDER BY – Ergebnisse sortieren

ORDER BY sortiert Ihre Ergebnisse. Standardmäßig aufsteigend (ASC = ascending), aber Sie können auch absteigend sortieren (DESC = descending). Sie können nach mehreren Spalten sortieren – zuerst nach Kategorie, dann nach Preis zum Beispiel.

Grundlegende Sortierung

Syntax:

```
SELECT columns
FROM table
ORDER BY column [ASC|DESC];
```

- **ASC** = Ascending (aufsteigend) – Standard
- **DESC** = Descending (absteigend)

Aufgabe: Schreiben Sie eine Abfrage, die die Spalten `name`, `price` und `category` aus der Tabelle `Products` anzeigt. Sortieren Sie die Ergebnisse nach `price` **aufsteigend** (günstigste Produkte zuerst).

```
-- Ihre Lösung hier:

```

No data

Aufgabe: Schreiben Sie eine Abfrage, die die Spalten `name`, `price` und `category` aus der Tabelle `Products` anzeigt. Sortieren Sie die Ergebnisse nach `price` **absteigend** (teuerste Produkte zuerst).

```
-- Ihre Lösung hier:

```

No data

Sie können nach mehreren Spalten sortieren. Die Reihenfolge ist wichtig: Erst wird nach der ersten Spalte sortiert, dann innerhalb gleicher Werte nach der zweiten Spalte, und so weiter. Hier sortieren wir erst nach Kategorie, dann innerhalb jeder Kategorie nach Preis.

Nach mehreren Spalten sortieren

Syntax:

```
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...
```

Aufgabe: Schreiben Sie eine Abfrage, die die Spalten `category`, `name` und `price` aus der Tabelle `Products` anzeigt. Sortieren Sie erst nach `category` **aufsteigend** (alphabetisch), dann innerhalb jeder Kategorie nach `price` **absteigend** (teuerste zuerst).

```
-- Ihre Lösung hier:

```

No data

Wie funktioniert das?

1. Alle Zeilen werden nach `category` sortiert (alphabetisch)
2. Innerhalb jeder Kategorie werden die Zeilen nach `price` sortiert (teuerste zuerst)

NULL-Werte sind ein Spezialfall bei der Sortierung. Standardmäßig kommen NULLs in DuckDB am Ende (bei ASC) oder am Anfang (bei DESC). Sie können das mit NULLS FIRST oder NULLS LAST explizit steuern.

NULL-Werte in ORDER BY

Standard-Verhalten:

- **ASC:** NULLs kommen am Ende
- **DESC:** NULLs kommen am Anfang

Explizite Steuerung:

```
ORDER BY column NULLS FIRST  -- NULLs zuerst
ORDER BY column NULLS LAST   -- NULLs zuletzt
```

Aufgabe: Schreiben Sie eine Abfrage, die die Spalten `name`, `price` und `rating` aus der Tabelle `Products` anzeigt. Sortieren Sie nach `rating` **absteigend** und platzieren Sie NULL-Werte **am Ende**.

```
1  -- Ihre Lösung hier:
2
```

No data

DISTINCT – Duplikate eliminieren

DISTINCT entfernt Duplikate aus Ihren Ergebnissen. Wenn Sie wissen wollen, welche Kategorien es gibt, ohne Wiederholungen, nutzen Sie `SELECT DISTINCT category`. Das gibt Ihnen eine Liste eindeutiger Werte.

Eindeutige Werte abrufen

Syntax:

```
SELECT DISTINCT column
FROM table;
```

Aufgabe: Schreiben Sie eine Abfrage, die alle **eindeutigen Kategorien** aus der Tabelle `Products` anzeigt. Sortieren Sie das Ergebnis alphabetisch nach `category`.

```
1  -- Ihre Lösung hier:
2
```

No data

Aufgabe: Schreiben Sie eine Abfrage, die alle **eindeutigen Brands** aus der Tabelle `Products` anzeigt. Sortieren Sie das Ergebnis alphabetisch nach `brand`.

```
1  -- Ihre Lösung hier:
2
```

No data

DISTINCT arbeitet über ALLE ausgewählten Spalten. Wenn Sie mehrere Spalten angeben, bekommt Sie jede eindeutige Kombination. Das ist wichtig zu verstehen: `SELECT DISTINCT category, brand` gibt Ihnen jede Kombination von Kategorie und Marke, die vorkommt.

DISTINCT über mehrere Spalten

DISTINCT gilt für die gesamte Zeile:

```
SELECT DISTINCT column1, column2
FROM table;
```

Gibt jede **eindeutige Kombination** von column1 + column2

Aufgabe: Schreiben Sie eine Abfrage, die alle **eindeutigen Kombinationen** von `category` und `brand` aus der Tabelle `Products` anzeigt. Sortieren Sie zuerst nach `category`, dann nach `brand`.

```
1  -- Ihre Lösung hier:
2
```

No data

Aufgabe (Zählen): Schreiben Sie eine Abfrage, die die **Anzahl der eindeutigen** Kombinationen von `category` und `brand` berechnet. Verwenden Sie `COUNT(DISTINCT ...)` und konkatenieren Sie die beiden Spalten mit `||`.

```
1  -- Ihre Lösung hier:
2
```

No data

DISTINCT vs. GROUP BY: Beides kann Duplikate entfernen, aber GROUP BY ist mächtiger. Mit GROUP BY können Sie aggregieren – zählen, summieren, Durchschnitt berechnen. DISTINCT gibt nur eindeutige Werte zurück, ohne Aggregation. Wir schauen uns GROUP BY im nächsten Abschnitt an.

DISTINCT vs. GROUP BY

DISTINCT:

- Entfernt Duplikate
- Keine Aggregation
- Einfacher, schneller für simple Fälle

```
SELECT DISTINCT category FROM Products;
```

GROUP BY:

- Gruppiert Zeilen
- Ermöglicht Aggregation (COUNT, SUM, AVG, …)
- Mächtiger, flexibler

```
1  SELECT category, COUNT(*) AS count
2  FROM Products
3  GROUP BY category;
```

	category	count
1	Electronics	100
2	Clothing	92
3	Groceries	100
4	Office	94
5	Home & Kitchen	90
6	Toys	93
7	Sports	91
8	Beauty	79
9	Books	95
10	Pets	98

Wann was nutzen?

- **DISTINCT:** Nur eindeutige Werte, keine Statistiken
- **GROUP BY:** Gruppierung + Aggregation

GROUP BY & HAVING – Aggregation

GROUP BY ist eine der mächtigsten Funktionen in SQL. Es gruppiert Zeilen mit gleichen Werten und ermöglicht Aggregation: Zählen Sie, wie viele Produkte es pro Kategorie gibt. Berechnen Sie den durchschnittlichen Preis pro Marke. Finden Sie die teuerste Ware pro Kategorie. All das geht mit GROUP BY.

Grundlagen von GROUP BY

Syntax:

```
SELECT column, AGGREGATE_FUNCTION(column)
FROM table
GROUP BY column;
```

Aggregat-Funktionen:

- `COUNT(*)` – Anzahl Zeilen
- `SUM(column)` – Summe
- `AVG(column)` – Durchschnitt
- `MIN(column)` – Minimum
- `MAX(column)` – Maximum

Aufgabe: Schreiben Sie eine Abfrage, die für jede Kategorie die **Anzahl der Produkte** berechnet. Zeigen Sie die Spalten `category` und `product_count` an. Sortieren Sie nach `product_count` **absteigend**.

```
1  -- Ihre Lösung hier:
2
```

No data

Sie können mehrere Aggregat-Funktionen gleichzeitig nutzen. Zum Beispiel: Für jede Kategorie zählen Sie die Produkte, berechnen den Durchschnittspreis, finden den günstigsten und teuersten Artikel. Das alles in einer Query.

Mehrere Aggregate gleichzeitig

Aufgabe: Schreiben Sie eine Abfrage, die für jede Kategorie folgende Statistiken berechnet: - `category` - `total_products` (Anzahl mit COUNT) - `avg_price` (Durchschnittspreis mit AVG) - `cheapest` (günstigstes Produkt mit MIN) - `most_expensive` (teuerstes Produkt mit MAX) - `total_stock` (Gesamtbestand mit SUM)

Sortieren Sie nach `total_products` absteigend.

```
1  -- Ihre Lösung hier:
2
```

No data

Aufgabe (mit Rundung): Schreiben Sie eine Abfrage, die für jede Kategorie die Spalten `category`, `count` (Anzahl), `avg_price`, `min_price` und `max_price` berechnet. **Runden** Sie alle Preis-Werte auf **2 Dezimalstellen** mit `ROUND()`.

```
1  -- Ihre Lösung hier:
2
```

No data

Nach mehreren Spalten gruppieren ist möglich. Zum Beispiel: Gruppieren Sie nach Kategorie UND Marke. Das gibt Ihnen Statistiken für jede Kombination – wie viele Laptops hat jede Marke, wie viele Handys, und so weiter.

Nach mehreren Spalten gruppieren

Syntax:

```
GROUP BY column1, column2, ...
```

Aufgabe: Schreiben Sie eine Abfrage, die Statistiken für jede **Kombination** von `category` und `brand` berechnet. Zeigen Sie die Spalten `category`, `brand`, `products` (Anzahl) und `avg_price` (gerundet auf 2 Dezimalstellen). Sortieren Sie erst nach `category`, dann nach `products` absteigend.

```
1  -- Ihre Lösung hier:
2
```

No data

HAVING ist wie WHERE, aber für gruppierte Daten. WHERE filtert VOR dem Gruppieren, HAVING filtert NACH dem Gruppieren. Sie können sagen: „Zeige mir nur Kategorien mit mehr als 50 Produkten„ oder „Brands mit einem Durchschnittspreis über 200 Euro“.

HAVING – Gruppen filtern

WHERE vs. HAVING:

- **WHERE:** Filtert **Zeilen** vor dem Gruppieren
- **HAVING:** Filtert **Gruppen** nach dem Gruppieren

Syntax:

```
SELECT column, AGGREGATE_FUNCTION(column)
FROM table
WHERE condition          -- Filter VORHER
GROUP BY column
HAVING condition;        -- Filter NACHHER
```

Aufgabe: Schreiben Sie eine Abfrage, die für jede Kategorie die Spalten `category`, `product_count` (Anzahl) und `avg_price` (gerundet auf 2 Dezimalstellen) berechnet. Filtern Sie mit HAVING nur Kategorien, die **mehr als 50 Produkte** haben. Sortieren Sie nach `product_count` absteigend.

```
1  -- Ihre Lösung hier:
2
```

No data

Aufgabe: Schreiben Sie eine Abfrage, die für jede Brand die Spalten `brand`, `products` (Anzahl) und `avg_price` (gerundet) berechnet. Filtern Sie mit HAVING nur Brands, deren **Durchschnittspreis über 200** liegt. Sortieren Sie nach `avg_price` absteigend.

```
1  -- Ihre Lösung hier:
2
```

No data

Komplexes Beispiel: Kombinieren Sie WHERE und HAVING. Filtern Sie erst die Zeilen (nur Electronics), dann gruppieren Sie (pro Brand), dann filtern Sie die Gruppen (nur Brands mit mehr als 5 Produkten). Das ist die Macht von SQL: Deklarativ beschreiben, was Sie wollen, und die Datenbank optimiert die Ausführung.

WHERE + GROUP BY + HAVING kombiniert

Aufgabe: Schreiben Sie eine Abfrage, die für jede Brand in der Kategorie **'Electronics'** folgende Spalten berechnet:

- `brand`
- `products` (Anzahl)
- `avg_price` (gerundet auf 2 Dezimalstellen)
- `min_price` (gerundet auf 2 Dezimalstellen)
- `max_price` (gerundet auf 2 Dezimalstellen)

Filtern Sie mit WHERE nur die Kategorie ‚Electronics'. Filtern Sie mit HAVING nur Brands mit **mehr als 5 Produkten**. Sortieren Sie nach `products` absteigend.

```
1  -- Ihre Lösung hier:
2
```

No data

Ausführungsreihenfolge:

1. `WHERE`: Nur Electronics
2. `GROUP BY`: Gruppiere nach Brand
3. `HAVING`: Nur Gruppen mit > 5 Produkten
4. `SELECT`: Berechne Aggregate
5. `ORDER BY`: Sortiere nach Anzahl

LIMIT – Top-N Queries

LIMIT begrenzt die Anzahl der zurückgegebenen Zeilen. Das ist perfekt für „Top 10„-Abfragen oder für Paginierung. Sie können auch einen Offset angeben: „Überspringe die ersten 20 Zeilen, dann gib mir die nächsten 10.“ Das ist die Grundlage für Seitennummerierung in Webanwendungen.

Anzahl Ergebnisse begrenzen

Syntax:

```
SELECT columns
FROM table
LIMIT n;
```

Aufgabe: Schreiben Sie eine Abfrage, die die Spalten `name`, `price` und `category` aus der Tabelle `Products` anzeigt. Sortieren Sie nach `price` **absteigend** und begrenzen Sie das Ergebnis auf die **10 teuersten Produkte**.

```
-- Ihre Lösung hier:

```

No data

Aufgabe: Schreiben Sie eine Abfrage, die die Spalten `name`, `price` und `category` aus der Tabelle `Products` anzeigt. Filtern Sie nur Produkte mit einem Preis **größer als 0**. Sortieren Sie nach `price` **aufsteigend** und begrenzen Sie das Ergebnis auf die **10 günstigsten Produkte**.

```
-- Ihre Lösung hier:

```

No data

Mit OFFSET können Sie Zeilen überspringen. Das ist die Basis für Paginierung: Seite 1 = LIMIT 10 OFFSET 0, Seite 2 = LIMIT 10 OFFSET 10, Seite 3 = LIMIT 10 OFFSET 20, und so weiter. Aber Achtung: OFFSET kann bei großen Werten langsam werden, weil die Datenbank alle übersprungenen Zeilen trotzdem verarbeiten muss.

OFFSET – Zeilen überspringen (Paginierung)

Syntax:

```
SELECT columns
FROM table
LIMIT n OFFSET m;
```

- `LIMIT n`: Maximal n Zeilen
- `OFFSET m`: Überspringe m Zeilen

Aufgabe: Schreiben Sie eine Abfrage, die die Spalten `name`, `price` und `category` aus der Tabelle `Products` anzeigt. Sortieren Sie nach `name`. Implementieren Sie Paginierung für **Seite 2** mit **10 Produkten pro Seite** (zeigen Sie Zeilen 11-20).

```
1  -- Ihre Lösung hier:
2
```

No data

Aufgabe: Schreiben Sie eine Abfrage für **Seite 3** (Zeilen 21-30).

```
1  -- Ihre Lösung hier:
2
```

No data

⚠️ **Performance-Problem bei großem OFFSET:**

- `OFFSET 10000` → Datenbank muss 10.000 Zeilen überspringen
- Besser: Cursor-basierte Paginierung (mit WHERE + ID)

Alternative zu OFFSET: Cursor-basierte Paginierung. Statt „überspringe 1000 Zeilen„ sagen Sie „gib mir alle Produkte mit ID größer als 1000“. Das ist viel schneller, weil die Datenbank direkt zum richtigen Startpunkt springen kann.

Cursor-basierte Paginierung (Alternative zu OFFSET)

Problem mit OFFSET:

```
-- Langsam bei großen Werten:
SELECT * FROM Products ORDER BY product_id LIMIT 10 OFFSET 10000;
```

Besser: WHERE + Cursor:

Aufgabe: Implementieren Sie cursor-basierte Paginierung. Schreiben Sie **Seite 2** einer Abfrage, die die Spalten `product_id`, `name` und `price` anzeigt. Nehmen Sie an, die letzte `product_id` von Seite 1 war **'P00010'**. Filtern Sie mit WHERE nur IDs **größer als** ‚**P00010**', sortieren Sie nach `product_id` und begrenzen Sie auf **10 Zeilen**.

```
1  -- Ihre Lösung hier:
2
```

No data

Vorteile:

- ✅ Schneller (kein Überspringen)
- ✅ Stabil bei Änderungen

Nachteil:

- ❌ Komplizierter Code
- ❌ Keine direkten Seitensprünge (Seite 5 → Seite 100)

Query Order & Execution

Jetzt wird es interessant: Die Reihenfolge, in der Sie SQL schreiben, ist NICHT die Reihenfolge, in der die Datenbank es ausführt. Sie schreiben SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT. Aber intern arbeitet die Datenbank in einer anderen Reihenfolge: FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY, LIMIT. Das zu verstehen ist wichtig für Performance und Fehlersuche.

Logische vs. Physische Ausführungsreihenfolge

Wie Sie es schreiben (logische Reihenfolge):

```
SELECT column, COUNT(*)       -- 5
FROM table                    -- 1
WHERE condition               -- 2
GROUP BY column               -- 3
HAVING condition              -- 4
ORDER BY column               -- 6
LIMIT n;                      -- 7
```

```
LIMIT ...;
```

Wie die Datenbank es ausführt (physische Reihenfolge):

1. **FROM:** Tabelle laden
2. **WHERE:** Zeilen filtern
3. **GROUP BY:** Gruppieren
4. **HAVING:** Gruppen filtern
5. **SELECT:** Spalten berechnen
6. **ORDER BY:** Sortieren
7. **LIMIT:** Anzahl begrenzen

Warum ist das wichtig? Erstens: Aliase aus SELECT sind in WHERE nicht verfügbar, weil WHERE VOR SELECT ausgeführt wird. Zweitens: HAVING kann Aliase nutzen, weil es NACH SELECT kommt. Drittens: ORDER BY kann Aliase nutzen. Das erklärt viele Fehler, die Anfänger machen.

Warum ist die Reihenfolge wichtig?

Beispiel-Query:

```sql
SELECT
  category,
  COUNT(*) AS product_count,
  AVG(price) AS avg_price
FROM Products
WHERE price > 100            -- Alias "avg_price" NICHT verfügbar
GROUP BY category
HAVING avg_price > 500       -- Alias "avg_price" verfügbar (nach SELECT)
ORDER BY product_count DESC; -- Alias "product_count" verfügbar
```

Warum?

- **WHERE** wird VOR **SELECT** ausgeführt → Aliase nicht verfügbar
- **HAVING** wird NACH **SELECT** ausgeführt → Aliase verfügbar
- **ORDER BY** wird NACH **SELECT** ausgeführt → Aliase verfügbar

Häufiger Fehler:

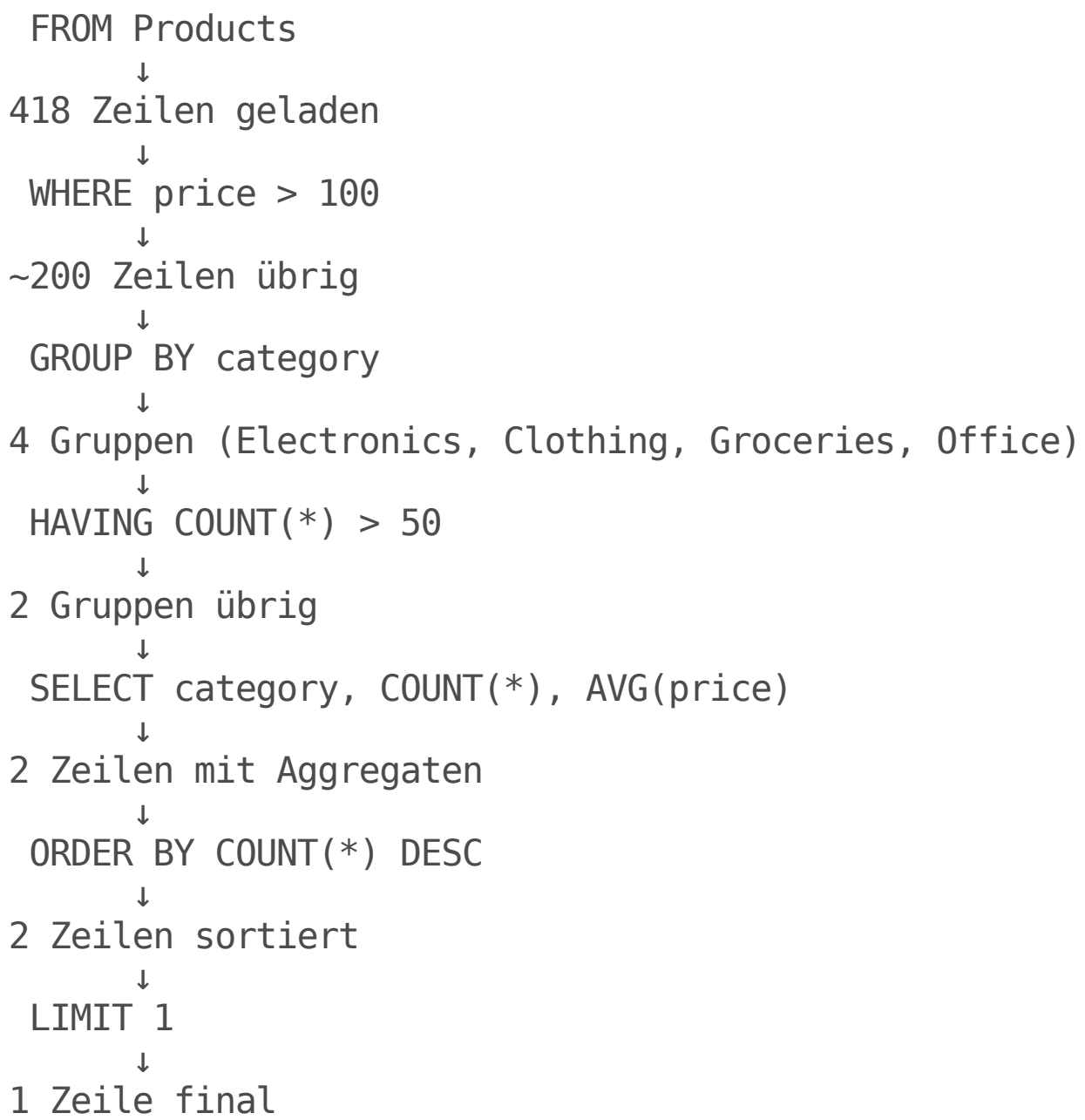
```sql
-- FEHLER:
SELECT name, price * 1.19 AS price_with_tax
FROM Products
WHERE price_with_tax > 100;   -- Alias nicht verfügbar!

-- RICHTIG:
SELECT name, price * 1.19 AS price_with_tax
FROM Products
WHERE price * 1.19 > 100;     -- Berechnung wiederholen
```

```
WHERE price * 1.19 > 100;          Berechnung wiederholen
```

Visualisieren wir das: FROM holt die Tabelle. WHERE reduziert die Zeilen. GROUP BY gruppiert. HAVING reduziert die Gruppen. SELECT berechnet die finalen Spalten. ORDER BY sortiert. LIMIT schneidet ab. Jeder Schritt arbeitet auf dem Ergebnis des vorherigen Schritts. Das ist die Pipeline.

Query-Ausführung visualisiert



Beispiel-Query:

```sql
SELECT
  category,
  COUNT(*) AS count,
  ROUND(AVG(price), 2) AS avg_price
FROM Products
WHERE price > 100
```

```sql
GROUP BY category
HAVING COUNT(*) > 50
ORDER BY count DESC
LIMIT 1;
```

	category	count	avg_price
1	Electronics	96	846.24

Zusammenfassung & Best Practices

Fassen wir zusammen: SELECT ist die Königin von SQL. Sie haben gelernt, Spalten auszuwählen, Daten zu filtern, zu sortieren, Duplikate zu entfernen, zu gruppieren und zu aggregieren. Sie verstehen die Ausführungsreihenfolge und wissen, warum Aliase manchmal funktionieren und manchmal nicht. Das ist Ihr Fundament für alle weiteren SQL-Sessions.

Was haben wir gelernt?

Konzept	Zweck	Syntax-Beispiel
SELECT	Spalten auswählen	`SELECT name, price FROM Products`
FROM	Tabelle angeben	`FROM Products`
WHERE	Zeilen filtern	`WHERE price > 100`
ORDER BY	Sortieren	`ORDER BY price DESC`
DISTINCT	Duplikate entfernen	`SELECT DISTINCT category`
GROUP BY	Gruppieren	`GROUP BY category`
HAVING	Gruppen filtern	`HAVING COUNT(*) > 10`
LIMIT	Anzahl begrenzen	`LIMIT 10`
OFFSET	Zeilen überspringen	`OFFSET 20`

Best Practices: Erstens – Spalten explizit benennen, nicht SELECT *. Zweitens – Aliase nutzen für Lesbarkeit. Drittens – WHERE für Zeilen-Filter, HAVING für Gruppen-Filter. Viertens – LIMIT für Top-N, aber Cursor für Paginierung. Fünftens – Ausführungsreihenfolge verstehen.

Best Practices

✅ Spalten explizit benennen

```
-- Gut:
SELECT product_id, name, price FROM Products;

-- Schlecht (außer für Exploration):
SELECT * FROM Products;
```

✅ Aliase für Lesbarkeit

```
SELECT
  product_id AS id,
  name AS product_name,
  price * 1.19 AS price_with_tax
FROM Products;
```

✅ WHERE für Zeilen, HAVING für Gruppen

```
-- Zeilen filtern:
WHERE price > 100

-- Gruppen filtern:
HAVING COUNT(*) > 10
```

✅ LIMIT für Top-N, Cursor für große Offsets

```
-- Top 10: OK
LIMIT 10

-- Seite 1000: Besser mit WHERE + ID
WHERE id > 'last_seen_id' LIMIT 10
```

✅ Ausführungsreihenfolge verstehen

```
FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT
```

Quiz: Testen Sie Ihr Wissen

Zeit für einen Selbsttest! Probieren Sie diese Fragen, um Ihr Verständnis zu überprüfen.

Frage 1: Welche Query zeigt die 5 teuersten Electronics-Produkte?

- ◯ `SELECT * FROM Products WHERE category = 'Electronics' LIMIT 5`
- ◯ `SELECT name, price FROM Products WHERE category = 'Electronics' ORDER BY price DESC LIMIT 5`
- ◯ `SELECT name, price FROM Products ORDER BY price LIMIT 5`
- ◯ `SELECT DISTINCT name FROM Products WHERE category = 'Electronics'`

Frage 2: Was ist der Unterschied zwischen WHERE und HAVING?

- ◯ Beide machen das Gleiche
- ◯ WHERE filtert Zeilen vor dem Gruppieren, HAVING filtert Gruppen nach dem Gruppieren
- ◯ WHERE ist schneller als HAVING
- ◯ HAVING kann nur mit COUNT() genutzt werden

Frage 3: In welcher Reihenfolge wird diese Query ausgeführt?

```sql
SELECT category, COUNT(*)
FROM Products
WHERE price > 100
GROUP BY category
ORDER BY COUNT(*) DESC;
```

- ◯ SELECT → FROM → WHERE → GROUP BY → ORDER BY
- ◯ FROM → WHERE → GROUP BY → SELECT → ORDER BY
- ◯ FROM → SELECT → WHERE → GROUP BY → ORDER BY
- ◯ FROM → GROUP BY → WHERE → SELECT → ORDER BY

Frage 4: Wie filtern Sie auf NULL-Werte?

- ◯ `WHERE column = NULL`
- ◯ `WHERE column == NULL`
- ◯ `WHERE column IS NULL`
- ◯ `WHERE ISNULL(column)`

Frage 5: Was macht DISTINCT?

- Entfernt Duplikate aus den Ergebnissen
- Sortiert die Ergebnisse
- Gruppiert die Ergebnisse
- Begrenzt die Anzahl der Ergebnisse

Übungsaufgaben

Probieren Sie diese Übungen selbst aus. Nutzen Sie die Products-Tabelle und experimentieren Sie mit verschiedenen Kombinationen.

Aufgabe 1: Grundlagen

Schreiben Sie eine Query, die alle Produkte der Kategorie „Clothing“ zeigt, sortiert nach Preis (günstigste zuerst), nur die ersten 10.

```
1
2
3
4
5
6
7
8
9
10
```

▶ **Lösung anzeigen**

Aufgabe 2: Aggregation

Berechnen Sie für jede Kategorie die Anzahl Produkte, den Durchschnittspreis und den Gesamtbestand. Sortieren Sie nach Anzahl Produkte (größte zuerst).

```
1
2
3
4
5
6
7
8
9
10
```

▶ **Lösung anzeigen**

Aufgabe 3: Filterung mit HAVING

Zeigen Sie alle Brands, die mehr als 20 Produkte haben UND deren Durchschnittspreis über 100 Euro liegt.

```
1
2
3
4
5
6
7
8
9
10
```

▶ **Lösung anzeigen**

Aufgabe 4: Komplexe Filterung

Finden Sie alle Electronics-Produkte, deren Name „Pro“ oder „Max“ enthält, mit einem Rating über 4.0, sortiert nach Rating (beste zuerst).

```
1
2
3
4
5
6
7
8
9
10
```

▶ **Lösung anzeigen**

Ausblick: Was kommt als Nächstes?

Sie haben jetzt die Grundlagen von SELECT gemeistert. In den nächsten Sessions gehen wir tiefer: Joins (mehrere Tabellen kombinieren), Subqueries (Queries in Queries), Window Functions (erweiterte Analysen), und vieles mehr. Aber alles baut auf dem auf, was Sie heute gelernt haben. SELECT ist Ihr Fundament.

Kommende Sessions:

- **Session 8:** Data Definition (CREATE, ALTER, DROP) & Data Manipulation (INSERT, UPDATE, DELETE)
- **Session 9:** Normalisierung & Datenmodellierung
- **Session 10:** Joins – Tabellen kombinieren
- **Session 11:** Subqueries & CTEs
- **Session 12:** Window Functions
- **Session 13:** Performance & Indexierung

🎉 **Herzlichen Glückwunsch!** Sie sind jetzt bereit, SQL-Abfragen zu schreiben!

Anhang: DuckDB Aggregat-Funktionen Übersicht

DuckDB bietet eine reiche Palette an Aggregatfunktionen – weit mehr als nur COUNT und SUM. Schauen wir uns die wichtigsten Kategorien an: Statistische Funktionen, String-Aggregate, logische Aggregate und approximative Funktionen. Jede hat ihren Einsatzzweck.

Standard-Aggregatfunktionen

Diese Funktionen kennen Sie bereits – sie sind das Fundament jeder Datenanalyse:

Funktion	Beschreibung	NULL-Verhalten	Beispiel
`COUNT(*)`	Anzahl aller Zeilen	Zählt auch NULL	`COUNT(*)`
`COUNT(column)`	Anzahl nicht-NULL-Werte	Ignoriert NULL	`COUNT(rating)`
`SUM(column)`	Summe aller Werte	Ignoriert NULL	`SUM(stock)`
`AVG(column)`	Durchschnitt	Ignoriert NULL	`AVG(price)`
`MIN(column)`	Kleinster Wert	Ignoriert NULL	`MIN(price)`
`MAX(column)`	Größter Wert	Ignoriert NULL	`MAX(price)`

Beispiel: Alle auf einmal

```sql
SELECT
  COUNT(*) AS total_products,
  COUNT(rating) AS rated_products,
  ROUND(AVG(price), 2) AS avg_price,
  MIN(price) AS cheapest,
  MAX(price) AS most_expensive,
  SUM(stock) AS total_inventory
FROM Products;
```

	total_products	rated_products	avg_price	cheapest	most_expensive
1	932	927	212.01	0.8999999761581421	1493.819946289061

Statistische Funktionen gehen über einfache Durchschnitte hinaus. Varianz und Standardabweichung zeigen, wie stark Ihre Daten streuen. Median ist robuster gegen Ausreißer als der Durchschnitt. Wenn ein einziges Produkt 10.000 Euro kostet, verschiebt das den Durchschnitt massiv – aber nicht den Median.

Statistische Funktionen

Diese Funktionen helfen bei tieferer Datenanalyse – Streuung, Median, Quantile:

Funktion	Beschreibung	Wann nutzen?
`STDDEV(column)`	Standardabweichung (Sample)	Streuung messen
`STDDEV_POP(column)`	Standardabweichung (Population)	Gesamtpopulation
`VARIANCE(column)`	Varianz (Sample)	Streuung^2
`VAR_POP(column)`	Varianz (Population)	Gesamtpopulation
`MEDIAN(column)`	Median (50. Perzentil)	Robuster Mittelwert
`QUANTILE(column, 0.25)`	Beliebiges Quantil	Quartile, Dezile

Beispiel: Preisverteilung analysieren

```sql
SELECT
  category,
  COUNT(*) AS products,
  ROUND(AVG(price), 2) AS mean_price,
  ROUND(MEDIAN(price), 2) AS median_price,
```

```
  ROUND(STDDEV(price), 2) AS price_stddev,
  ROUND(QUANTILE(price, 0.25), 2) AS q1_price,
  ROUND(QUANTILE(price, 0.75), 2) AS q3_price
FROM Products
GROUP BY category
ORDER BY products DESC;
```

	category	products	mean_price	median_price	price_stddev	q1_price
1	Electronics	100	814.81	820.3599853515625	424.99	486.970001
2	Groceries	100	21.12	21	12.2	10.3500003
3	Pets	98	97.91	98.73999786376953	57.43	55.4500007
4	Books	95	41.27	39.20000076293945	21.38	22.4300003
5	Office	94	266.53	282.44000244140625	140.13	159.539993
6	Toys	93	106.07	113.45999908447266	55.36	56.6199989
7	Clothing	92	116.83	109.62999725341797	71.81	51.4000015
8	Sports	91	172.76	164.08999633789062	95.45	94.3799972
9	Home & Kitchen	90	370.11	376.0799865722656	189.66	199.479995
10	Beauty	79	73.31	72.27999877929688	39.2	38.7700004

Was sagt uns das?

- **Mean vs. Median:** Große Differenz → Ausreißer vorhanden
- **Standardabweichung:** Hoch → große Preisspanne
- **Q1/Q3:** Interquartilsbereich = mittlere 50% der Daten

String-Aggregate sind mächtig, wenn Sie Werte zusammenfassen wollen. STRING*AGG sammelt alle Werte einer Gruppe in einen einzigen String – perfekt für „zeige mir alle Brands pro Kategorie„ oder „liste alle Produktnamen in einer Zeile“. LIST/ARRAY*AGG erstellt Arrays, die Sie später weiterverarbeiten können.

String- und Listen-Aggregate

Diese Funktionen sammeln Werte in Strings oder Arrays:

Funktion	Beschreibung	Ausgabe	Beispiel
`STRING_AGG(column, separator)`	Konkateniert Strings mit Trennzeichen	String	`STRING_AGG(name, ', ')`
`LIST(column)`	Sammelt Werte in Array	Array	`LIST(brand)`
`ARRAY_AGG(column)`	Alias für LIST	Array	`ARRAY_AGG(price)`

Beispiel: Alle Brands pro Kategorie

```sql
SELECT
  category,
  COUNT(DISTINCT brand) AS brand_count,
  STRING_AGG(DISTINCT brand, ', ' ORDER BY brand) AS brands
FROM Products
GROUP BY category
ORDER BY brand_count DESC;
```

	category	brand_count	brands
1	Electronics	7	Auralite, BlueWave, Neutrino, OrbiTech, PixelPeak, Voltix, ZenCore
2	Clothing	7	AeroLoom, Cotton&Co, Meridian Wear, Nordline, PureThread, Seaside, UrbanTrail
3	Groceries	6	DailyChoice, Everfarm, FreshField, GreenVale, Pure&Simple, SunHarvest
4	Office	6	DeskPro, ErgoCraft, InkFlow, PaperMint, StapleStar, WriteRight
5	Home & Kitchen	6	AquaPure, BrightGlow, CasaNova, CozyNest, Hearthstone, KitchenCrafter
6	Toys	6	HappyBlocks, KiddyLab, PlayVerse, RoboPal, SunnyPlay, WonderWorks
7	Sports	6	AeroFlex, CoreAthlete, FitFoundry, PeakMotion, TrailRunner, WaveRider
8	Beauty	6	AurumCare, LunaLeaf, OceanMist, PureGlow, SilkAura, VelvetSkin
9	Books	6	Atlas Editions, Northwind Press, OpenPage, Orion Leaf, Silver Quill, Sunset House
10	Pets	6	AquaPets, FurryFriends, HappyTail, Paw&Co, PetNest, WildFeast

Beispiel: Preis-Arrays für Analyse

```sql
SELECT
  category,
  LIST(price ORDER BY price) AS all_prices,
  LIST(price ORDER BY price DESC LIMIT 5) AS top_5_prices
FROM Products
GROUP BY category;
```

```
Error executing statement: SELECT
  category,
  LIST(price ORDER BY price) AS all_prices,
  LIST(price ORDER BY price DESC LIMIT 5) AS top_5_prices
FROM Products
GROUP BY category Parser Error: syntax error at or near "LIMIT"
LINE 4:   LIST(price ORDER BY price DESC LIMIT 5) AS top_5_prices
                                                  ^
```

💡 **Tipp:** `STRING_AGG` kann mit `ORDER BY` innerhalb der Funktion sortieren!
Logische Aggregate sind unterschätzt, aber extrem nützlich. BOOL_AND prüft: „Sind ALLE Werte in der Gruppe wahr?„ BOOL_OR prüft: „Ist MINDESTENS einer wahr?“ Das ist perfekt für Validierung: „Hat jede Kategorie mindestens ein Produkt auf Lager?„ oder „Sind alle Produkte bewertet?“

Logische Aggregate

Perfekt für Validierung und Bedingungsprüfungen:

Funktion	Beschreibung	Gibt TRUE wenn...
`BOOL_AND(condition)`	Logisches AND über alle Zeilen	ALLE Zeilen TRUE sind
`BOOL_OR(condition)`	Logisches OR über alle Zeilen	MINDESTENS eine Zeile TRUE ist
`EVERY(condition)`	Alias für BOOL_AND	ALLE Zeilen TRUE sind

Beispiel: Validierung pro Kategorie

```sql
SELECT
  category,
  COUNT(*) AS products,
  BOOL_AND(stock > 0) AS all_in_stock,
  BOOL_OR(stock > 100) AS some_high_stock,
  BOOL_AND(rating IS NOT NULL) AS all_rated,
  BOOL_OR(price > 1000) AS has_premium_items
FROM Products
GROUP BY category;
```

	category	products	all_in_stock	some_high_stock	all_rated	has_premium_iten
1	Electronics	100	true	true	false	true
2	Clothing	92	true	true	false	false
3	Groceries	100	true	true	true	false
4	Office	94	true	true	true	false
5	Home & Kitchen	90	true	true	true	false
6	Toys	93	true	true	true	false
7	Sports	91	true	true	true	false
8	Beauty	79	true	true	true	false
9	Books	95	true	true	false	false
10	Pets	98	true	true	false	false

Interpretation:

- `all_in_stock = TRUE` → Alle Produkte verfügbar
- `some_high_stock = TRUE` → Mindestens ein Produkt mit hohem Bestand
- `all_rated = FALSE` → Nicht alle Produkte haben Bewertungen

Approximative Funktionen sind die Geheimwaffe für Big Data. APPROX*COUNT*DISTINCT zählt eindeutige Werte, aber nicht exakt – dafür viel schneller und speicherschonender. Bei Millionen von Zeilen ist der Unterschied zwischen „exakt 10.234.567„ und „circa 10.2 Millionen“ oft irrelevant. Sie gewinnen massive Performance für minimalen Genauigkeitsverlust.

Approximative Aggregate (Performance)

Für große Datenmengen – schneller, aber mit kleinem Fehler:

Funktion	Beschreibung	Genauigkeit	Wann nutzen?
`APPROX_COUNT_DISTINCT(column)`	Ungefähre Anzahl eindeutiger Werte	~2% Fehler	Millionen von Zeilen
`APPROX_QUANTILE(column, 0.5)`	Approximatives Quantil	~1% Fehler	Große Datasets

Beispiel: Exakt vs. Approximativ

```
1 SELECT
```

```sql
SELECT
  'Exact' AS method,
  COUNT(DISTINCT product_id) AS unique_products,
  COUNT(DISTINCT brand) AS unique_brands
FROM Products

UNION ALL

SELECT
  'Approx' AS method,
  APPROX_COUNT_DISTINCT(product_id) AS unique_products,
  APPROX_COUNT_DISTINCT(brand) AS unique_brands
FROM Products;
```

	method	unique_products	unique_brands
1	Exact	932	62
2	Approx	1171	53

💡 **Performance-Tipp:**

- Bei < 1 Million Zeilen: Exakte Funktionen nutzen
- Bei > 10 Millionen Zeilen: Approximativ kann 10x schneller sein
- Bei 418 Zeilen (unsere Products): Kein Unterschied 😊

Fortgeschrittene Aggregate: FIRST und LAST holen den ersten oder letzten Wert in einer Gruppe – nützlich für Zeitreihen. BIT*AND/BIT*OR/BIT*XOR sind für Bit-Operationen. Und dann gibt es noch spezielle Aggregate wie MODE (häufigster Wert) und ARG*MIN/ARG_MAX (Wert einer anderen Spalte bei Min/Max).

Fortgeschrittene Aggregate

Spezialisierte Funktionen für besondere Anwendungsfälle:

Funktion	Beschreibung	Beispiel
`FIRST(column)`	Erster Wert in Gruppe	`FIRST(name ORDER BY price)`
`LAST(column)`	Letzter Wert in Gruppe	`LAST(name ORDER BY created_at)`
`ARG_MIN(arg, val)`	Argument beim Minimum-Wert	`ARG_MIN(name, price)`
`ARG_MAX(arg, val)`	Argument beim Maximum-Wert	`ARG_MAX(name, price)`
`MODE(column)`	Häufigster Wert	`MODE(category)`

Beispiel: Günstigstes und teuerstes Produkt pro Kategorie

```sql
SELECT
  category,
  ARG_MIN(name, price) AS cheapest_product,
  MIN(price) AS min_price,
  ARG_MAX(name, price) AS most_expensive_product,
  MAX(price) AS max_price
FROM Products
GROUP BY category
ORDER BY category;
```

	category	cheapest_product	min_price	most_expensive_product	max
1	Beauty	Body Lotion 250ml Pro S - Graphite	4.489999771118164	Hand Cream 75ml Plus	139
2	Books	Travel Guide: Japan Pro - Graphite	6.190000057220459	Travel Guide: Japan S	78.8
3	Clothing	Dress Shirt 2.0 S	6.130000114440918	Leggings Max - Silver	245
4	Electronics	4K Monitor	24.920000076293945	Bluetooth Speaker	149
5	Groceries	Arabica Coffee Beans 1kg	0.8999999761581421	Green Tea 50 bags Plus	39.8
6	Home & Kitchen	Cutlery Set (24 pcs) Lite - White	15.220000267028809	Blender 1.5L Max	691
7	Office	Laptop Stand Lite	7.53000020980835	Laptop Stand	487
8	Pets	Pet Bed Medium Plus	3.950000047683716	Bird Seed 1kg	199
9	Sports	Bike Helmet	24.809999465942383	Dumbbells Pair 2x5kg Max M	339
10	Toys	Marble Run 2.0	6.849999904632568	Science Kit - Volcano - Green	199

Beispiel: Häufigste Brand pro Kategorie

```sql
SELECT
  category,
  MODE(brand) AS most_common_brand,
  COUNT(*) AS total_products
FROM Products
GROUP BY category;
```

	category	most_common_brand	total_products
1	Electronics	Voltix	100
2	Clothing	Nordline	92
3	Groceries	DailyChoice	100
4	Office	DeskPro	94
5	Home & Kitchen	KitchenCrafter	90
6	Toys	SunnyPlay	93
7	Sports	TrailRunner	91
8	Beauty	SilkAura	79
9	Books	Atlas Editions	95
10	Pets	PetNest	98

💡 **Warum ARG*MIN/ARG*MAX?**

Statt zwei Queries:

```
-- Umständlich:
SELECT name FROM Products WHERE price = (SELECT MIN(price) FROM Products);

-- Elegant:
SELECT ARG_MIN(name, price) FROM Products;
```

FILTER-Klausel: Das ist ein Game-Changer für bedingte Aggregation. Statt mehrere CASE-Statements zu schreiben, nutzen Sie FILTER direkt in der Aggregatfunktion. „Zähle nur Electronics„, „Summiere nur Produkte über 100 Euro“, „Durchschnitt nur für bewertete Artikel" – alles in einer Zeile.

FILTER-Klausel (Bedingte Aggregation)

Syntax:

```
AGGREGATE_FUNCTION(column) FILTER (WHERE condition)
```

Aggregiert nur Zeilen, die die Bedingung erfüllen.

Beispiel: Kategorisierte Statistiken

```
SELECT
  COUNT(*) AS total,
  COUNT(*) FILTER (WHERE price < 100) AS budget_items,
  COUNT(*) FILTER (WHERE price BETWEEN 100 AND 500) AS mid_range,
  COUNT(*) FILTER (WHERE price > 500) AS premium,
  AVG(price) FILTER (WHERE rating > 4.0) AS avg_price_top_rated,
```

```
  SUM(stock) FILTER (WHERE category = 'Electronics') AS electronics_st
FROM Products;
```

	total	budget_items	mid_range	premium	avg_price_top_rated	electronics_stock
1	932	435	396	101	205.47850862373724	15703

Beispiel: Pro Kategorie mit Filtern

```
SELECT
  category,
  COUNT(*) AS total,
  COUNT(*) FILTER (WHERE stock > 0) AS in_stock,
  COUNT(*) FILTER (WHERE stock = 0) AS out_of_stock,
  ROUND(AVG(price) FILTER (WHERE rating >= 4.0), 2) AS avg_price_good_
FROM Products
GROUP BY category
ORDER BY total DESC;
```

	category	total	in_stock	out_of_stock	avg_price_good_rated
1	Electronics	100	100	0	818.7
2	Groceries	100	100	0	21.85
3	Pets	98	98	0	103.23
4	Books	95	95	0	40.19
5	Office	94	94	0	254.56
6	Toys	93	93	0	109.06
7	Clothing	92	92	0	111.81
8	Sports	91	91	0	165.61
9	Home & Kitchen	90	90	0	375.22
10	Beauty	79	79	0	73.67

💡 **Statt CASE WHEN:**

```
-- Umständlich:
SUM(CASE WHEN price > 100 THEN 1 ELSE 0 END)

-- Elegant:
COUNT(*) FILTER (WHERE price > 100)
```