Session 7 – SQL Introduction & SELECT Statements (Komplettlösung)

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.

```
1  INSTALL httpfs;
2  LOAD httpfs;
3
4  CREATE TABLE Products AS
5  SELECT * FROM read_json('https://raw.githubusercontent.com/andre-dietr
     /Datenbankensysteme-Vorlesung/refs/heads/main/assets/dat/products.js
```

```
INSTALL httpfs
```

No data

```
LOAD httpfs
```

No data

```
CREATE TABLE Products AS
SELECT * FROM read_json('https://raw.githubusercontent.com/andre-
dietrich/Datenbankensysteme-
Vorlesung/refs/heads/main/assets/dat/products.json')
```

	Count
1	932

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

```
1  const res  = await fetch('https://raw.githubusercontent.com/andre-die
     /Datenbankensysteme-Vorlesung/refs/heads/main/assets/dat/products.j
2  const text = await res.text();
3
```

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

// jetzt normal aus der "lokalen" Datei lesen
await conn.query(`CREATE TABLE Products AS SELECT * FROM read_json
  ('products.json');`);

console.log("ready")
```

```
ready
```

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

	product_id	name	category	brand	price	stock	rating	created_at
1	1	Laptop 14" 2.0	Electronics	PixelPeak	1399.03	138	3.7	2025-06-14
2	2	Portable SSD Max M	Electronics	ZenCore	805.93	211	3.9	2024-05-21
3	3	4K Monitor - Green	Electronics	Neutrino	522.48	261	5	2025-10-16
4	4	Tablet 10	Electronics	Voltix	985.75	54	3.9	2025-04-16
5	5	Portable SSD Lite	Electronics	OrbiTech	508.23	20	4.2	2025-03-12

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“

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

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

```
-- - 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"

Live-Beispiel:

```
1  SELECT * FROM Products LIMIT 10;
```

```
Error executing statement: SELECT * FROM Products LIMIT 10 Catalog
Error: Table with name Products does not exist!
Did you mean "duckdb_types"?
LINE 1: SELECT * FROM Products LIMIT 10
                      ^
```

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;
```

Beispiel: Nur ID, Name und Preis

```
1  SELECT product_id, name, price
2  FROM Products
```

```
3 LIMIT 10;
```

	product_id	name	price
1	1	Laptop 14" 2.0	1399.03
2	2	Portable SSD Max M	805.93
3	3	4K Monitor - Green	522.48
4	4	Tablet 10	985.75
5	5	Portable SSD Lite	508.23
6	6	Soundbar - Red	786.98
7	7	Mechanical Keyboard - Black	537.59
8	8	Laptop 14" Plus	1187.46
9	9	Action Camera - Green	502.12
10	10	Soundbar	1340.9

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;
```

Beispiel:

```
1 SELECT
2   product_id AS id,
3   name AS product_name,
4   price AS price_eur,
5   category
6 FROM Products
```

```
LIMIT 10;
```

	id	product_name	price_eur	category
1	1	Laptop 14" 2.0	1399.03	Electronics
2	2	Portable SSD Max M	805.93	Electronics
3	3	4K Monitor - Green	522.48	Electronics
4	4	Tablet 10	985.75	Electronics
5	5	Portable SSD Lite	508.23	Electronics
6	6	Soundbar - Red	786.98	Electronics
7	7	Mechanical Keyboard - Black	537.59	Electronics
8	8	Laptop 14" Plus	1187.46	Electronics
9	9	Action Camera - Green	502.12	Electronics
10	10	Soundbar	1340.9	Electronics

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 columns
FROM table
WHERE condition;
```

Beispiel: Alle Produkte unter 50 Euro

```
SELECT product_id, name, price
FROM Products
WHERE price < 50
```

```
4 LIMIT 10;
```

	product_id	name	price
1	11	4K Monitor	24.92
2	105	Basic T-Shirt Max	24.34
3	107	Jacket	44
4	113	Jacket L	29.28
5	114	Raincoat	43.81
6	115	Jeans - Silver	34.16
7	117	Leggings L	18.1
8	122	Jeans	35.78
9	140	Cap Lite	35.55
10	141	Dress Shirt - Blue	39.97

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`

Beispiel: Produkte über 1000 Euro

```
1 SELECT name, price, category
2 FROM Products
3 WHERE price >= 1000
```

```
4 LIMIT 10;
```

	name	price	category
1	Laptop 14" 2.0	1399.03	Electronics
2	Laptop 14" Plus	1187.46	Electronics
3	Soundbar	1340.9	Electronics
4	Noise-Canceling Earbuds Lite	1339.39	Electronics
5	Action Camera - Blue	1322.9	Electronics
6	Gaming Mouse	1402.13	Electronics
7	USB-C Hub - White	1093.57	Electronics
8	USB-C Hub Plus	1356.47	Electronics
9	Noise-Canceling Earbuds Max	1352.11	Electronics
10	Mechanical Keyboard - Blue	1106.59	Electronics

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

AND – Beide Bedingungen müssen wahr sein:

```
1 SELECT name, price, category, stock
2 FROM Products
3 WHERE price < 100 AND stock > 200
4 LIMIT 10;
```

	name	price	category	stock
1	Smartwatch - White	61.61	Electronics	203
2	4K Monitor	62.9	Electronics	219
3	Socks (5-pack) 2.0 M	84.42	Clothing	295
4	Basic T-Shirt Max	24.34	Clothing	702
5	Jacket L	29.28	Clothing	647
6	Raincoat	43.81	Clothing	761
7	Jeans - Silver	34.16	Clothing	318
8	Basic T-Shirt Lite	75.34	Clothing	520
9	Hoodie M	52.36	Clothing	701
10	Jeans	35.78	Clothing	525

OR – Mindestens eine Bedingung muss wahr sein:

```sql
SELECT name, price, category
FROM Products
WHERE category = 'Electronics' OR category = 'Office'
LIMIT 10;
```

	name	price	category
1	Laptop 14" 2.0	1399.03	Electronics
2	Portable SSD Max M	805.93	Electronics
3	4K Monitor - Green	522.48	Electronics
4	Tablet 10	985.75	Electronics
5	Portable SSD Lite	508.23	Electronics
6	Soundbar - Red	786.98	Electronics
7	Mechanical Keyboard - Black	537.59	Electronics
8	Laptop 14" Plus	1187.46	Electronics
9	Action Camera - Green	502.12	Electronics
10	Soundbar	1340.9	Electronics

NOT – Negiert eine Bedingung:

```sql
SELECT name, price, category
FROM Products
WHERE NOT category = 'Groceries'
LIMIT 10;
```

	name	price	category
1	Laptop 14" 2.0	1399.03	Electronics
2	Portable SSD Max M	805.93	Electronics
3	4K Monitor - Green	522.48	Electronics
4	Tablet 10	985.75	Electronics
5	Portable SSD Lite	508.23	Electronics
6	Soundbar - Red	786.98	Electronics
7	Mechanical Keyboard - Black	537.59	Electronics
8	Laptop 14" Plus	1187.46	Electronics
9	Action Camera - Green	502.12	Electronics
10	Soundbar	1340.9	Electronics

⚠️ **Achtung bei gemischten Operatoren:**

```sql
-- 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:

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

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

Beispiel: Produkte zwischen 100 und 500 Euro

```sql
SELECT name, price, category
FROM Products
WHERE price BETWEEN 100 AND 500
LIMIT 10;
```

	name	price	category
1	Laptop 14" Pro S	481.53	Electronics
2	USB-C Hub - Red	317.33	Electronics
3	Wireless Headphones	377.2	Electronics
4	Tablet 10" - Black	128.34	Electronics
5	Gaming Mouse Lite S	486.97	Electronics
6	4K Monitor	226.26	Electronics
7	Bluetooth Speaker XL	215.56	Electronics
8	Gaming Mouse Pro S	270.97	Electronics
9	Smartwatch Pro L - Graphite	394.22	Electronics
10	Portable SSD Plus	271.37	Electronics

Datum-Bereiche:

```sql
SELECT name, price, created_at
FROM Products
WHERE created_at BETWEEN '2025-01-01' AND '2025-06-30'
LIMIT 10;
```

	name	price	created_at
1	Laptop 14" 2.0	1399.03	2025-06-14
2	Tablet 10	985.75	2025-04-16
3	Portable SSD Lite	508.23	2025-03-12
4	Laptop 14" Plus	1187.46	2025-03-13
5	Action Camera - Green	502.12	2025-03-27
6	Soundbar	1340.9	2025-02-25
7	Action Camera XL	636.73	2025-04-14
8	Bluetooth Speaker Plus	865.42	2025-02-23
9	Gaming Mouse	1402.13	2025-06-28
10	USB-C Hub - Red	317.33	2025-06-25

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, ...)
```

Beispiel: Nur bestimmte Kategorien

```
1  SELECT name, price, category
2  FROM Products
3  WHERE category IN ('Electronics', 'Office', 'Groceries')
4  LIMIT 10;
```

	name	price	category
1	Laptop 14" 2.0	1399.03	Electronics
2	Portable SSD Max M	805.93	Electronics
3	4K Monitor - Green	522.48	Electronics
4	Tablet 10	985.75	Electronics
5	Portable SSD Lite	508.23	Electronics
6	Soundbar - Red	786.98	Electronics
7	Mechanical Keyboard - Black	537.59	Electronics
8	Laptop 14" Plus	1187.46	Electronics
9	Action Camera - Green	502.12	Electronics
10	Soundbar	1340.9	Electronics

Negation mit NOT IN:

```sql
SELECT name, price, category
FROM Products
WHERE category NOT IN ('Clothing', 'Home & Kitchen')
LIMIT 10;
```

	name	price	category
1	Laptop 14" 2.0	1399.03	Electronics
2	Portable SSD Max M	805.93	Electronics
3	4K Monitor - Green	522.48	Electronics
4	Tablet 10	985.75	Electronics
5	Portable SSD Lite	508.23	Electronics
6	Soundbar - Red	786.98	Electronics
7	Mechanical Keyboard - Black	537.59	Electronics
8	Laptop 14" Plus	1187.46	Electronics
9	Action Camera - Green	502.12	Electronics
10	Soundbar	1340.9	Electronics

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

Beispiel: Produkte, die „Laptop" im Namen haben

```
SELECT name, price, category
FROM Products
WHERE name LIKE '%Laptop%'
LIMIT 10;
```

	name	price	category
1	Laptop 14" 2.0	1399.03	Electronics
2	Laptop 14" Plus	1187.46	Electronics
3	Laptop 14" Pro S	481.53	Electronics
4	Laptop 14" 2.0 - Black	833.21	Electronics
5	Laptop 14" 2.0 - White	132.35	Electronics
6	Laptop 14	772.23	Electronics
7	Laptop 14" Lite	1290.81	Electronics
8	Laptop Stand XL	374.78	Office
9	Laptop Stand	113.7	Office
10	Laptop Stand	402.72	Office

Beispiel: Namen, die mit „Wireless" beginnen

```
SELECT name, price, category
FROM Products
WHERE name LIKE 'Wireless%'
LIMIT 10;
```

	name	price	category
1	Wireless Headphones	377.2	Electronics
2	Wireless Headphones M	697.05	Electronics
3	Wireless Headphones Pro S - Silver	523.27	Electronics
4	Wireless Headphones Max	639.32	Electronics
5	Wireless Headphones	852.73	Electronics
6	Wireless Headphones	995.22	Electronics
7	Wireless Mouse XL	375.27	Office
8	Wireless Mouse S	291.38	Office
9	Wireless Mouse	377.43	Office
10	Wireless Mouse M	87.62	Office

Beispiel: Produkt-IDs im Format „P00_01" (genau 6 Zeichen)

```sql
SELECT product_id, name
FROM Products
WHERE product_id LIKE 'P00_01'
LIMIT 10;
```

```
Error executing statement: SELECT product_id, name
FROM Products
WHERE product_id LIKE 'P00_01'
LIMIT 10 Binder Error: No function matches the given name and argument
types '~~(BIGINT, STRING_LITERAL)'. You might need to add explicit type
casts.
        Candidate functions:
        ~~(VARCHAR, VARCHAR) -> BOOLEAN

LINE 3: WHERE product_id LIKE 'P00_01'
                              ^
```

⚠️ **Case Sensitivity:**

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

```sql
SELECT name FROM Products WHERE name ILIKE '%laptop%' LIMIT 5;
```

	name
1	Laptop 14" 2.0
2	Laptop 14" Plus
3	Laptop 14" Pro S
4	Laptop 14" 2.0 - Black
5	Laptop 14" 2.0 - White

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
```

Beispiel: Produkte ohne Rating

```
SELECT name, price, rating
FROM Products
WHERE rating IS NULL
LIMIT 10;
```

	name	price	rating
1	Action Camera	1259.3	null
2	Hoodie M	52.36	null
3	Childrens Stories Pro	70.79	null
4	Dog Leash Nylon L	59.81	null
5	Cat Litter 10L	196	null

Beispiel: Produkte MIT Rating

```sql
SELECT name, price, rating
FROM Products
WHERE rating IS NOT NULL
LIMIT 10;
```

	name	price	rating
1	Laptop 14" 2.0	1399.03	3.7
2	Portable SSD Max M	805.93	3.9
3	4K Monitor - Green	522.48	5
4	Tablet 10	985.75	3.9
5	Portable SSD Lite	508.23	4.2
6	Soundbar - Red	786.98	5
7	Mechanical Keyboard - Black	537.59	4.4
8	Laptop 14" Plus	1187.46	4.6
9	Action Camera - Green	502.12	5
10	Soundbar	1340.9	3.7

⚠️ **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)

Beispiel: Produkte nach Preis sortiert (günstigste zuerst)

```
1  SELECT name, price, category
2  FROM Products
3  ORDER BY price ASC
4  LIMIT 10;
```

	name	price	category
1	Arabica Coffee Beans 1kg	0.9	Groceries
2	Honey 500g - Black	1.73	Groceries
3	Granola 750g XL - Graphite	1.99	Groceries
4	Organic Oat Milk 1L Lite S - Red	2.25	Groceries
5	Basmati Rice 1kg	2.33	Groceries
6	Almonds 200g Lite XL	2.48	Groceries
7	Sea Salt 500g Lite - Graphite	3.07	Groceries
8	Arabica Coffee Beans 1kg	3.09	Groceries
9	Almonds 200g	3.59	Groceries
10	Green Tea 50 bags	3.68	Groceries

Beispiel: Teuerste Produkte zuerst

```
1  SELECT name, price, category
2  FROM Products
```

```
ORDER BY price DESC
LIMIT 10;
```

	name	price	category
1	Bluetooth Speaker	1493.82	Electronics
2	Mechanical Keyboard	1462.4	Electronics
3	4K Monitor	1445.86	Electronics
4	Tablet 10	1433.7	Electronics
5	Soundbar Pro S	1425.74	Electronics
6	Bluetooth Speaker Lite M	1425.09	Electronics
7	Gaming Mouse	1402.13	Electronics
8	Laptop 14" 2.0	1399.03	Electronics
9	4K Monitor - White	1390.69	Electronics
10	4K Monitor XL	1389.73	Electronics

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], ...
```

Beispiel: Erst nach Kategorie, dann nach Preis

```
SELECT category, name, price
FROM Products
ORDER BY category ASC, price DESC
LIMIT 20;
```

	category	name	price
1	Beauty	Hand Cream 75ml Plus	139.58
2	Beauty	Face Mask Sheet Max	138.5
3	Beauty	Hand Cream 75ml Plus XL - Red	136.27
4	Beauty	Face Mask Sheet	135.28
5	Beauty	Sunscreen SPF50 100ml 2.0 - Green	135.21
6	Beauty	Facial Cleanser 200ml Plus	134.25
7	Beauty	Moisturizing Cream 50ml Lite	134.06
8	Beauty	Face Mask Sheet L	131.34
9	Beauty	Conditioner 300ml	130.17
10	Beauty	Micellar Water 400ml Pro	129.8
11	Beauty	Hair Oil 100ml 2.0 - White	127.31
12	Beauty	Body Lotion 250ml Pro - Black	126.14
13	Beauty	Moisturizing Cream 50ml	124.66
14	Beauty	Hair Oil 100ml Lite S	120.61
15	Beauty	Moisturizing Cream 50ml	120.47
16	Beauty	Sunscreen SPF50 100ml	114.51
17	Beauty	Body Lotion 250ml Pro	111.37
18	Beauty	Shampoo 300ml Plus L	107.01
19	Beauty	Conditioner 300ml Pro L	105.44
20	Beauty	Micellar Water 400ml Pro	100.06

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
```

Beispiel: Produkte nach Rating sortiert, NULLs am Ende

```
1  SELECT name, price, rating
2  FROM Products
3  ORDER BY rating DESC NULLS LAST
4  LIMIT 10;
```

	name	price	rating
1	4K Monitor - Green	522.48	5
2	Soundbar - Red	786.98	5
3	Action Camera - Green	502.12	5
4	Wireless Headphones M	697.05	5
5	Noise-Canceling Earbuds	784.44	5
6	Bluetooth Speaker M	853.91	5
7	USB-C Hub Pro	1371.22	5
8	Gaming Mouse	92.53	5
9	Action Camera - White	129.17	5
10	Socks (5-pack) 2.0 M	84.42	5

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;
```

Beispiel: Alle Kategorien (ohne Duplikate)

```sql
SELECT DISTINCT category
FROM Products
ORDER BY category;
```

	category
1	Beauty
2	Books
3	Clothing
4	Electronics
5	Groceries
6	Home & Kitchen
7	Office
8	Pets
9	Sports
10	Toys

Beispiel: Alle Brands

```sql
SELECT DISTINCT brand
FROM Products
ORDER BY brand;
```

	brand
1	AeroFlex
2	AeroLoom
3	AquaPets
4	AquaPure
5	Atlas Editions
6	Auralite
7	AurumCare
8	BlueWave
9	BrightGlow
10	CasaNova
11	CoreAthlete
12	Cotton&Co
13	CozyNest
14	DailyChoice
15	DeskPro
16	ErgoCraft
17	Everfarm
18	FitFoundry
19	FreshField
20	FurryFriends
21	GreenVale
22	HappyBlocks
23	HappyTail
24	Hearthstone
25	InkFlow
26	KiddyLab
27	KitchenCrafter
28	LunaLeaf
29	Meridian Wear
30	Neutrino
31	Nordline

32	Northwind Press
33	OceanMist
34	OpenPage
35	OrbiTech
36	Orion Leaf
37	PaperMint
38	Paw&Co
39	PeakMotion
40	PetNest
41	PixelPeak
42	PlayVerse
43	Pure&Simple
44	PureGlow
45	PureThread
46	RoboPal
47	Seaside
48	SilkAura
49	Silver Quill
50	StapleStar
51	SunHarvest
52	SunnyPlay
53	Sunset House
54	TrailRunner
55	UrbanTrail
56	VelvetSkin
57	Voltix
58	WaveRider
59	WildFeast
60	WonderWorks
61	WriteRight
62	ZenCore

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

Beispiel: Alle Kategorie-Brand-Kombinationen

```
1  SELECT DISTINCT category, brand
2  FROM Products
3  ORDER BY category, brand;
```

	category	brand
1	Beauty	AurumCare
2	Beauty	LunaLeaf
3	Beauty	OceanMist
4	Beauty	PureGlow
5	Beauty	SilkAura
6	Beauty	VelvetSkin
7	Books	Atlas Editions
8	Books	Northwind Press
9	Books	OpenPage
10	Books	Orion Leaf
11	Books	Silver Quill
12	Books	Sunset House
13	Clothing	AeroLoom
14	Clothing	Cotton&Co
15	Clothing	Meridian Wear
16	Clothing	Nordline
17	Clothing	PureThread
18	Clothing	Seaside
19	Clothing	UrbanTrail
20	Electronics	Auralite
21	Electronics	BlueWave
22	Electronics	Neutrino
23	Electronics	OrbiTech
24	Electronics	PixelPeak
25	Electronics	Voltix
26	Electronics	ZenCore
27	Groceries	DailyChoice
28	Groceries	Everfarm
29	Groceries	FreshField
30	Groceries	GreenVale
31	Groceries	Pure&Simple

32	Groceries	SunHarvest
33	Home & Kitchen	AquaPure
34	Home & Kitchen	BrightGlow
35	Home & Kitchen	CasaNova
36	Home & Kitchen	CozyNest
37	Home & Kitchen	Hearthstone
38	Home & Kitchen	KitchenCrafter
39	Office	DeskPro
40	Office	ErgoCraft
41	Office	InkFlow
42	Office	PaperMint
43	Office	StapleStar
44	Office	WriteRight
45	Pets	AquaPets
46	Pets	FurryFriends
47	Pets	HappyTail
48	Pets	Paw&Co
49	Pets	PetNest
50	Pets	WildFeast
51	Sports	AeroFlex
52	Sports	CoreAthlete
53	Sports	FitFoundry
54	Sports	PeakMotion
55	Sports	TrailRunner
56	Sports	WaveRider
57	Toys	HappyBlocks
58	Toys	KiddyLab
59	Toys	PlayVerse
60	Toys	RoboPal
61	Toys	SunnyPlay
62	Toys	WonderWorks

Wie viele eindeutige Kombinationen gibt es?

```
SELECT COUNT(DISTINCT category || '-' || brand) AS unique_combinations
FROM Products;
```

	unique_combinations
1	62

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

```
SELECT category, COUNT(*) AS count
FROM Products
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

Beispiel: Anzahl Produkte pro Kategorie

```sql
SELECT category, COUNT(*) AS product_count
FROM Products
GROUP BY category
ORDER BY product_count DESC;
```

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

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

Beispiel: Statistiken pro Kategorie

```sql
SELECT
  category,
  COUNT(*) AS total_products,
  AVG(price) AS avg_price,
  MIN(price) AS cheapest,
  MAX(price) AS most_expensive,
  SUM(stock) AS total_stock
```

```
FROM Products
GROUP BY category
ORDER BY total_products DESC;
```

	category	total_products	avg_price	cheapest	most_expensive	tota
1	Electronics	100	814.8121000000001	24.92	1493.82	1570
2	Groceries	100	21.123399999999997	0.9	39.84	1049
3	Pets	98	97.90653061224492	3.95	199.08	3153
4	Books	95	41.270526315789475	6.19	78.86	2392
5	Office	94	266.5322340425532	7.53	487.1	4150
6	Toys	93	106.06688172043015	6.85	199.29	3966
7	Clothing	92	116.82739130434777	6.13	245.35	3822
8	Sports	91	172.75527472527477	24.81	339.06	4566
9	Home & Kitchen	90	370.11433333333326	15.22	691.34	2788
10	Beauty	79	73.30569620253166	4.49	139.58	4420

Runden für Lesbarkeit:

```
SELECT
  category,
  COUNT(*) AS count,
  ROUND(AVG(price), 2) AS avg_price,
  ROUND(MIN(price), 2) AS min_price,
  ROUND(MAX(price), 2) AS max_price
FROM Products
GROUP BY category;
```

	category	count	avg_price	min_price	max_price
1	Electronics	100	814.81	24.92	1493.82
2	Clothing	92	116.83	6.13	245.35
3	Groceries	100	21.12	0.9	39.84
4	Office	94	266.53	7.53	487.1
5	Home & Kitchen	90	370.11	15.22	691.34
6	Toys	93	106.07	6.85	199.29
7	Sports	91	172.76	24.81	339.06
8	Beauty	79	73.31	4.49	139.58
9	Books	95	41.27	6.19	78.86
10	Pets	98	97.91	3.95	199.08

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, ...
```

Beispiel: Statistiken pro Kategorie und Brand

```
SELECT
  category,
  brand,
  COUNT(*) AS products,
  ROUND(AVG(price), 2) AS avg_price
FROM Products
GROUP BY category, brand
ORDER BY category, products DESC
LIMIT 20;
```

	category	brand	products	avg_price
1	Beauty	SilkAura	17	61.71
2	Beauty	VelvetSkin	16	87.72
3	Beauty	LunaLeaf	15	58.55
4	Beauty	AurumCare	12	99.37
5	Beauty	OceanMist	10	67.96
6	Beauty	PureGlow	9	65.35
7	Books	Atlas Editions	23	47.18
8	Books	Silver Quill	18	43.15
9	Books	Northwind Press	16	37.52
10	Books	OpenPage	15	44.37
11	Books	Orion Leaf	14	37.58
12	Books	Sunset House	9	29.67
13	Clothing	Nordline	21	127.98
14	Clothing	AeroLoom	17	99.44
15	Clothing	PureThread	14	122.46
16	Clothing	UrbanTrail	11	102.3
17	Clothing	Cotton&Co	10	97.1
18	Clothing	Seaside	10	106.19
19	Clothing	Meridian Wear	9	166.38
20	Electronics	Voltix	20	875.83

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
```

Beispiel: Nur Kategorien mit mehr als 50 Produkten

```sql
SELECT
  category,
  COUNT(*) AS product_count,
  ROUND(AVG(price), 2) AS avg_price
FROM Products
GROUP BY category
HAVING COUNT(*) > 50
ORDER BY product_count DESC;
```

	category	product_count	avg_price
1	Electronics	100	814.81
2	Groceries	100	21.12
3	Pets	98	97.91
4	Books	95	41.27
5	Office	94	266.53
6	Toys	93	106.07
7	Clothing	92	116.83
8	Sports	91	172.76
9	Home & Kitchen	90	370.11
10	Beauty	79	73.31

Beispiel: Brands mit Durchschnittspreis über 200€

```sql
SELECT
  brand,
  COUNT(*) AS products,
  ROUND(AVG(price), 2) AS avg_price
FROM Products
GROUP BY brand
HAVING AVG(price) > 200
ORDER BY avg_price DESC;
```

	brand	products	avg_price
1	PixelPeak	9	954.7
2	Auralite	17	923.31
3	Voltix	20	875.83
4	OrbiTech	15	773.56
5	Neutrino	14	736.94
6	BlueWave	9	730.2
7	ZenCore	16	698.99
8	AquaPure	11	438.38
9	CasaNova	10	424.64
10	CozyNest	12	378.65
11	KitchenCrafter	23	365.49
12	Hearthstone	19	339.7
13	ErgoCraft	11	332.83
14	BrightGlow	15	322.49
15	PaperMint	15	292.76
16	InkFlow	15	287.15
17	DeskPro	19	266.03
18	StapleStar	15	243.54
19	WriteRight	19	209.83

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

Beispiel: Electronics-Brands mit mehr als 5 Produkten

```sql
SELECT
  brand,
  COUNT(*) AS products,
  ROUND(AVG(price), 2) AS avg_price,
  ROUND(MIN(price), 2) AS min_price,
  ROUND(MAX(price), 2) AS max_price
FROM Products
WHERE category = 'Electronics'    -- Filter VORHER
GROUP BY brand
```

```
10  HAVING COUNT(*) > 5                    -- Filter NACHHER
11  ORDER BY products DESC;
```

	brand	products	avg_price	min_price	max_price
1	Voltix	20	875.83	132.35	1462.4
2	Auralite	17	923.31	415.47	1376.9
3	ZenCore	16	698.99	61.61	1493.82
4	OrbiTech	15	773.56	215.56	1445.86
5	Neutrino	14	736.94	24.92	1402.13
6	PixelPeak	9	954.7	298.11	1399.03
7	BlueWave	9	730.2	270.97	1390.69

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;
```

Beispiel: Die 10 teuersten Produkte

```
1  SELECT name, price, category
2  FROM Products
3  ORDER BY price DESC
4  LIMIT 10;
```

	name	price	category
1	Bluetooth Speaker	1493.82	Electronics
2	Mechanical Keyboard	1462.4	Electronics
3	4K Monitor	1445.86	Electronics
4	Tablet 10	1433.7	Electronics
5	Soundbar Pro S	1425.74	Electronics
6	Bluetooth Speaker Lite M	1425.09	Electronics
7	Gaming Mouse	1402.13	Electronics
8	Laptop 14" 2.0	1399.03	Electronics
9	4K Monitor - White	1390.69	Electronics
10	4K Monitor XL	1389.73	Electronics

Beispiel: Die 10 günstigsten Produkte

```sql
SELECT name, price, category
FROM Products
WHERE price > 0
ORDER BY price ASC
LIMIT 10;
```

	name	price	category
1	Arabica Coffee Beans 1kg	0.9	Groceries
2	Honey 500g - Black	1.73	Groceries
3	Granola 750g XL - Graphite	1.99	Groceries
4	Organic Oat Milk 1L Lite S - Red	2.25	Groceries
5	Basmati Rice 1kg	2.33	Groceries
6	Almonds 200g Lite XL	2.48	Groceries
7	Sea Salt 500g Lite - Graphite	3.07	Groceries
8	Arabica Coffee Beans 1kg	3.09	Groceries
9	Almonds 200g	3.59	Groceries
10	Green Tea 50 bags	3.68	Groceries

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

Beispiel: Paginierung (Seite 2, 10 pro Seite)

```
1  SELECT name, price, category
2  FROM Products
3  ORDER BY name
4  LIMIT 10 OFFSET 10;  -- Zeilen 11-20
```

```
SELECT name, price, category
FROM Products
ORDER BY name
LIMIT 10 OFFSET 10
```

	name	price	category
1	4K Monitor XL	1389.73	Electronics
2	A4 Copy Paper 500 sheets	461.91	Office
3	A4 Copy Paper 500 sheets	159.02	Office
4	A4 Copy Paper 500 sheets	33.8	Office
5	A4 Copy Paper 500 sheets - Blue	407.61	Office
6	A4 Copy Paper 500 sheets - Silver	417.1	Office
7	A4 Copy Paper 500 sheets M - Black	290.11	Office
8	Action Camera	1259.3	Electronics
9	Action Camera	1027.7	Electronics
10	Action Camera - Blue	1322.9	Electronics

```
-- Zeilen 11-20
```

No data

Seite 3:

```sql
SELECT name, price, category
FROM Products
ORDER BY name
LIMIT 10 OFFSET 20;  -- Zeilen 21-30
```

```
SELECT name, price, category
FROM Products
ORDER BY name
LIMIT 10 OFFSET 20
```

	name	price	category
1	Action Camera - Green	502.12	Electronics
2	Action Camera - White	129.17	Electronics
3	Action Camera 2.0	885.27	Electronics
4	Action Camera Plus	1376.9	Electronics
5	Action Camera XL	636.73	Electronics
6	Air Fryer 4L	572.27	Home & Kitchen
7	Air Fryer 4L	580.17	Home & Kitchen
8	Air Fryer 4L	27.29	Home & Kitchen
9	Air Fryer 4L - Black	327.12	Home & Kitchen
10	Air Fryer 4L - Black	552.19	Home & Kitchen

```
-- Zeilen 21-30
```

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:

```
1  -- Seite 1:
2  SELECT product_id, name, price
```

```sql
FROM Products
ORDER BY product_id
LIMIT 10;
-- Merke letzte ID: z.B. P00010

-- Seite 2:
SELECT product_id, name, price
FROM Products
WHERE product_id > 'P00010'
ORDER BY product_id
LIMIT 10;
```

```
-- Seite 1:
SELECT product_id, name, price
FROM Products
ORDER BY product_id
LIMIT 10
```

	product_id	name	price
1	1	Laptop 14" 2.0	1399.03
2	2	Portable SSD Max M	805.93
3	3	4K Monitor - Green	522.48
4	4	Tablet 10	985.75
5	5	Portable SSD Lite	508.23
6	6	Soundbar - Red	786.98
7	7	Mechanical Keyboard - Black	537.59
8	8	Laptop 14" Plus	1187.46
9	9	Action Camera - Green	502.12
10	10	Soundbar	1340.9

```
-- Merke letzte ID: z.B. P00010

-- Seite 2:
SELECT product_id, name, price
FROM Products
WHERE product_id > 'P00010'
ORDER BY product_id
LIMIT 10
Error executing statement: -- Merke letzte ID: z.B. P00010

-- Seite 2:
SELECT product_id, name, price
FROM Products
WHERE product_id > 'P00010'
ORDER BY product_id
LIMIT 10 Conversion Error: Could not convert string 'P00010' to INT64
LINE 6: WHERE product_id > 'P00010'
                          ^
```

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
```

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
```

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

```
 FROM Products
      ↓
418 Zeilen geladen
      ↓
 WHERE price > 100
      ↓
~200 Zeilen übrig
      ↓
 GROUP BY category
      ↓
4 Gruppen (Electronics, Clothing, Groceries, Office)
      ↓
 HAVING COUNT(*) > 50
      ↓
2 Gruppen übrig
      ↓
 SELECT category, COUNT(*), AVG(price)
      ↓
2 Zeilen mit Aggregaten
      ↓
 ORDER BY COUNT(*) DESC
      ↓
2 Zeilen sortiert
      ↓
 LIMIT 1
      ↓
1 Zeile final
```

Beispiel-Query:

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

```
```

```sql SELECT category, COUNT(*) AS product_count, ROUND(AVG(price), 2) AS avg_price, SUM(stock) AS total_stock FROM Products GROUP BY category ORDER BY product_count DESC; ``` **********

Aufgabe 3: Filterung mit HAVING

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

```
```

```sql SELECT name, price, rating, brand FROM Products WHERE category = ‚Electronics‘ AND (name LIKE ‚%Pro%‘ OR name LIKE ‚%Max%‘) AND rating > 4.0 ORDER BY rating DESC LIMIT 20; ``` **********

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)
- **Session 9:** Data Manipulation (INSERT, UPDATE, DELETE)
- **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!

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	total_inve
1	932	927	212.01	0.9	1493.82	413286

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	q3_pri
1	Electronics	100	814.81	820.37	424.99	486.97	1175.6
2	Groceries	100	21.12	21	12.2	10.35	32.26
3	Pets	98	97.91	98.74	57.43	55.45	142.56
4	Books	95	41.27	39.2	21.38	22.43	59.63
5	Office	94	266.53	282.44	140.13	159.54	376.69
6	Toys	93	106.07	113.46	55.36	56.62	146.49
7	Clothing	92	116.83	109.63	71.81	51.4	177.36
8	Sports	91	172.76	164.09	95.45	94.38	261.38
9	Home & Kitchen	90	370.11	376.09	189.66	199.48	541.02
10	Beauty	79	73.31	72.28	39.2	38.77	100.06

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_price
1	Beauty	Body Lotion 250ml Pro S - Graphite	4.49	Hand Cream 75ml Plus	139.58
2	Books	Travel Guide: Japan Pro - Graphite	6.19	Travel Guide: Japan S	78.86
3	Clothing	Dress Shirt 2.0 S	6.13	Leggings Max - Silver	245.35
4	Electronics	4K Monitor	24.92	Bluetooth Speaker	1493.82
5	Groceries	Arabica Coffee Beans 1kg	0.9	Green Tea 50 bags Plus	39.84
6	Home & Kitchen	Cutlery Set (24 pcs) Lite - White	15.22	Blender 1.5L Max	691.34
7	Office	Laptop Stand Lite	7.53	Laptop Stand	487.1
8	Pets	Pet Bed Medium Plus	3.95	Bird Seed 1kg	199.08
9	Sports	Bike Helmet	24.81	Dumbbells Pair 2x5kg Max M	339.06
10	Toys	Marble Run 2.0	6.85	Science Kit - Volcano - Green	199.29

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: ` sql – 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	205.47850931676984	15703

Beispiel: Pro Kategorie mit Filtern

```sql
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:**

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

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