Aggregationen & Window Functions

Session 16 – Lecture (90 Minuten) **Block 4:** Theorie, Optimierung & Polyglot **Lernziel:** LZ 2 – SQL-Praxis vertiefen mit Aggregationen & Window Functions

Willkommen zur sechzehnten Vorlesung! Heute lernen Sie zwei mächtige SQL-Werkzeuge kennen: Aggregationen und Window Functions. Mit echten Wetterdaten werden wir Durchschnitte berechnen, Trends entdecken und gleitende Mittelwerte erstellen. Sie werden sehen, welche analytischen Möglichkeiten SQL bietet – von einfachen Summen bis zu komplexen Zeitreihen-Analysen.

Was erwartet Sie heute?

Heute lernen Sie die wichtigsten Werkzeuge für Datenanalyse in SQL: Klassische Aggregationen und fortgeschrittene Window Functions. Alles mit echten Wetterdaten – über 4000 Messungen aus mehreren Monaten.

Überblick

- **Klassische Aggregationen:** COUNT, SUM, AVG, MIN, MAX, GROUP BY, HAVING
- **Window Functions:** ROW_NUMBER, RANK, LAG, LEAD, gleitende Mittelwerte
- **Zeitbasierte Analytics:** EXTRACT, ROWS BETWEEN
- **Praktische Anwendungen:** Trend-Erkennung, Anomalie-Suche, Vergleiche

---

Setup: Wetterdaten laden

Lassen Sie uns mit unseren Daten starten. Wir haben echte Wettermessungen aus den letzten Monaten – Temperatur, Luftdruck, Windgeschwindigkeit, Luftfeuchte und mehr. Insgesamt über 4000 Zeilen.

Daten laden

```
1  const res = await fetch('../assets/dat/weather.csv', { cache: "no-sto
     );
2  if (!res.ok) throw new Error(res.statusText);
3  const csvText = await res.text();
4
5  // als "Datei" in DuckDB registrieren
6  await db.registerFileText('weather.csv', csvText);
7
8  // jetzt normal aus der "lokalen" Datei lesen
9  await conn.query(`CREATE TABLE weather AS SELECT * FROM read_csv('wea
     .csv');`);
10
11 console.log("ready")
```

```
-- Today
```

Schauen wir uns die Struktur an: Datum, Anzahl Messwerte pro Tag, dann verschiedene Sensoren.

Datenstruktur verstehen

```
DESCRIBE weather;
```

```
Error executing statement: DESCRIBE weather Catalog Error: Table with
name weather does not exist!
Did you mean "pg_attrdef"?
```

Sie sehen: 12 Spalten, hauptsächlich numerische Werte. Perfekt für Aggregationen!

Teil 1: Klassische Aggregationen

Starten wir mit den Basics: Aggregationsfunktionen. Diese kennen Sie bereits, aber heute schauen wir genauer hin, wie sie intern funktionieren.

COUNT, SUM, AVG, MIN, MAX

Die fundamentalen Aggregationsfunktionen – das Fundament jeder Datenanalyse. Beginnen wir mit einem ganz einfachen Beispiel.

Beispiel: Wie viele Tage haben wir?

```
SELECT COUNT(*) as anzahl_tage
FROM weather;
```

```
Error executing statement: SELECT COUNT(*) as anzahl_tage
FROM weather Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 2: FROM weather
             ^
```

Syntax-Erklärung: `COUNT(*)` zählt alle Zeilen in der Tabelle. Das Sternchen `*` bedeutet „alle Zeilen“. Mit `as anzahl_tage` geben wir der Spalte einen lesbaren Namen.

Jetzt kombinieren wir mehrere Aggregationen in einer Query:

Mehrere Aggregationen gleichzeitig

```
SELECT
  COUNT(*) as anzahl_tage,
  AVG(Temp_2m) as durchschnitts_temp,
  MIN(Temp_2m) as min_temp,
  MAX(Temp_2m) as max_temp
```

```
6  FROM weather;
```

```
Error executing statement: SELECT
  COUNT(*) as anzahl_tage,
  AVG(Temp_2m) as durchschnitts_temp,
  MIN(Temp_2m) as min_temp,
  MAX(Temp_2m) as max_temp
FROM weather Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 6: FROM weather
             ^
```

Syntax-Erklärung: - `AVG(Temp_2m)` berechnet den Durchschnitt aller Temperatur-Werte - `MIN(Temp_2m)` findet die niedrigste Temperatur - `MAX(Temp_2m)` findet die höchste Temperatur - Alle vier Werte werden in **einer** Zeile ausgegeben!
Die Zahlen haben viele Nachkommastellen. Mit ROUND machen wir sie lesbarer:

Zahlen runden mit ROUND

```
1  SELECT
2    COUNT(*) as anzahl_tage,
3    ROUND(AVG(Temp_2m), 2) as durchschnitts_temp,
4    ROUND(MIN(Temp_2m), 2) as min_temp,
5    ROUND(MAX(Temp_2m), 2) as max_temp
6  FROM weather;
```

```
Error executing statement: SELECT
  COUNT(*) as anzahl_tage,
  ROUND(AVG(Temp_2m), 2) as durchschnitts_temp,
  ROUND(MIN(Temp_2m), 2) as min_temp,
  ROUND(MAX(Temp_2m), 2) as max_temp
FROM weather Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 6: FROM weather
             ^
```

Syntax-Erklärung: `ROUND(wert, 2)` rundet auf 2 Nachkommastellen. Statt `7.123456` bekommen Sie `7.12`. Das ist viel lesbarer!

GROUP BY – Gruppierte Aggregationen

Jetzt wird es spannender: Gruppierungen! Statt einen Durchschnitt für alle Daten zu berechnen, wollen wir Durchschnitte pro Monat. Dafür brauchen wir GROUP BY.

Nach Monat gruppieren

```sql
SELECT
  EXTRACT(MONTH FROM Datum) as monat,
  ROUND(AVG(Temp_2m), 2) as avg_temp,
  COUNT(*) as anzahl_tage
FROM weather
GROUP BY EXTRACT(MONTH FROM Datum)
ORDER BY monat;
```

```
Error executing statement: SELECT
  EXTRACT(MONTH FROM Datum) as monat,
  ROUND(AVG(Temp_2m), 2) as avg_temp,
  COUNT(*) as anzahl_tage
FROM weather
GROUP BY EXTRACT(MONTH FROM Datum)
ORDER BY monat Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 5: FROM weather
             ^
```

Syntax-Erklärung: - `EXTRACT(MONTH FROM Datum)` holt die Monatszahl aus dem Datum (1 für Januar, 2 für Februar, usw.) - `GROUP BY` gruppiert alle Zeilen mit dem gleichen Monat zusammen - `AVG(Temp_2m)` berechnet dann den Durchschnitt **pro Gruppe** (also pro Monat) - `ORDER BY monat` sortiert von Januar (1) bis Dezember (12)

Wir sehen jetzt 12 Zeilen – eine für jeden Monat! Aber Moment: Was, wenn wir nur kalte Monate sehen wollen?

HAVING – Filterung nach Aggregation

```sql
SELECT
  EXTRACT(MONTH FROM Datum) as monat,
  ROUND(AVG(Temp_2m), 2) as avg_temp,
  COUNT(*) as anzahl_tage
FROM weather
GROUP BY EXTRACT(MONTH FROM Datum)
HAVING AVG(Temp_2m) < 5
ORDER BY avg_temp;
```

```
Error executing statement: SELECT
  EXTRACT(MONTH FROM Datum) as monat,
  ROUND(AVG(Temp_2m), 2) as avg_temp,
  COUNT(*) as anzahl_tage
FROM weather
GROUP BY EXTRACT(MONTH FROM Datum)
HAVING AVG(Temp_2m) < 5
ORDER BY avg_temp Catalog Error: Table with name weather does not
exist!
Did you mean "pg_attrdef"?
LINE 5: FROM weather
             ^
```

Syntax-Erklärung: - `HAVING AVG(Temp_2m) < 5` filtert **nach** der Aggregation - Nur Monate mit Durchschnittstemperatur unter 5°C werden angezeigt - **Wichtig:** WHERE filtert **vor** GROUP BY, HAVING filtert **nach** GROUP BY!
Der Unterschied zwischen WHERE und HAVING verwirrt oft. Hier ein Vergleich:

WHERE vs. HAVING:

- WHERE: Filtert einzelne Zeilen **vor** der Gruppierung (z.B. `WHERE Temp_2m > 0`)
- HAVING: Filtert Gruppen **nach** der Aggregation (z.B. `HAVING AVG(Temp_2m) < 5`)

Teil 2: Window Functions

Jetzt kommen wir zu den mächtigen Window Functions – ein Game-Changer für Analytics. Window Functions erlauben es Ihnen, Berechnungen über Zeilen-Bereiche durchzuführen, ohne zu gruppieren.

Grundkonzept: OVER

Window Functions sind anders als GROUP BY: Sie behalten **alle** Zeilen bei, fügen aber trotzdem aggregierte Werte hinzu. Das klingt kompliziert? Schauen wir uns ein Beispiel an!

Schritt 1: GROUP BY kollabiert Zeilen

```sql
SELECT
  EXTRACT(MONTH FROM Datum) as monat,
  ROUND(AVG(Temp_2m), 2) as avg_temp
FROM weather
GROUP BY EXTRACT(MONTH FROM Datum)
ORDER BY monat;
```

```
Error executing statement: SELECT
  EXTRACT(MONTH FROM Datum) as monat,
  ROUND(AVG(Temp_2m), 2) as avg_temp
FROM weather
GROUP BY EXTRACT(MONTH FROM Datum)
ORDER BY monat Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 4: FROM weather
             ^
```

Was passiert: Aus tausenden Zeilen werden nur 12 (eine pro Monat). Wir verlieren die einzelnen Tage! Aber was, wenn wir die einzelnen Tage **behalten** wollen, aber trotzdem den Monats-Durchschnitt sehen?

Schritt 2: Window Function behält alle Zeilen

```sql
SELECT
  Datum,
  Temp_2m,
  ROUND(AVG(Temp_2m) OVER (
    PARTITION BY EXTRACT(MONTH FROM Datum)
  ), 2) as monats_durchschnitt
FROM weather
ORDER BY Datum DESC
LIMIT 10;
```

```
Error executing statement: SELECT
  Datum,
  Temp_2m,
  ROUND(AVG(Temp_2m) OVER (
    PARTITION BY EXTRACT(MONTH FROM Datum)
  ), 2) as monats_durchschnitt
FROM weather
ORDER BY Datum DESC
LIMIT 10 Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 7: FROM weather
             ^
```

Syntax-Erklärung: - `AVG(Temp_2m) OVER (...)` ist eine Window Function - `OVER` sagt: „Berechne über ein Fenster“ - `PARTITION BY EXTRACT(MONTH FROM Datum)` teilt die Daten in Gruppen (hier: Monate) - **Wichtig:** Alle Zeilen bleiben erhalten! Jede Zeile bekommt den Durchschnitt ihres Monats dazu.

Jetzt können wir die Abweichung vom Monatsdurchschnitt berechnen:

Abweichung vom Monatsdurchschnitt

```sql
SELECT
  Datum,
  ROUND(Temp_2m, 2) as temp,
  ROUND(AVG(Temp_2m) OVER (
    PARTITION BY EXTRACT(MONTH FROM Datum)
  ), 2) as monats_avg,
  ROUND(
    Temp_2m - AVG(Temp_2m) OVER (
      PARTITION BY EXTRACT(MONTH FROM Datum)
    ), 2
  ) as abweichung
FROM weather
ORDER BY Datum DESC
LIMIT 10;
```

```
Error executing statement: SELECT
  Datum,
  ROUND(Temp_2m, 2) as temp,
  ROUND(AVG(Temp_2m) OVER (
    PARTITION BY EXTRACT(MONTH FROM Datum)
  ), 2) as monats_avg,
  ROUND(
    Temp_2m - AVG(Temp_2m) OVER (
      PARTITION BY EXTRACT(MONTH FROM Datum)
    ), 2
  ) as abweichung
FROM weather
ORDER BY Datum DESC
LIMIT 10 Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 12: FROM weather
             ^
```

Was wir sehen: Jeder Tag zeigt seine Temperatur, den Monatsdurchschnitt und die Abweichung. Positive Werte = wärmer als Durchschnitt, negative = kälter.

ROW_NUMBER, RANK, DENSE_RANK

Manchmal wollen Sie die Top 10 finden – die kältesten Tage, die heißesten Tage, etc. Dafür gibt es Ranking-Funktionen!

Die 5 kältesten Tage finden

```sql
SELECT
  Datum,
```

```
  ROUND(Temp_2m, 2) as temp,
  ROW_NUMBER() OVER (ORDER BY Temp_2m) as position
FROM weather
ORDER BY temp
LIMIT 5;
```

```
Error executing statement: SELECT
  Datum,
  ROUND(Temp_2m, 2) as temp,
  ROW_NUMBER() OVER (ORDER BY Temp_2m) as position
FROM weather
ORDER BY temp
LIMIT 5 Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 5: FROM weather
             ^
```

Syntax-Erklärung: - `ROW_NUMBER() OVER (ORDER BY Temp_2m)` gibt jeder Zeile eine Nummer - `ORDER BY Temp_2m` sortiert vom kältesten zum wärmsten - Position 1 = kältester Tag, Position 2 = zweitkältester, usw. - **Wichtig:** Jede Position kommt genau einmal vor!
Aber was, wenn zwei Tage die exakt gleiche Temperatur haben? Sollten beide Platz 1 bekommen?

Drei Ranking-Funktionen im Vergleich

```
SELECT
  Datum,
  ROUND(Temp_2m, 2) as temp,
  ROW_NUMBER() OVER (ORDER BY Temp_2m) as row_num,
  RANK() OVER (ORDER BY Temp_2m) as rank,
  DENSE_RANK() OVER (ORDER BY Temp_2m) as dense_rank
FROM weather
ORDER BY temp
LIMIT 8;
```

```
Error executing statement: SELECT
  Datum,
  ROUND(Temp_2m, 2) as temp,
  ROW_NUMBER() OVER (ORDER BY Temp_2m) as row_num,
  RANK() OVER (ORDER BY Temp_2m) as rank,
  DENSE_RANK() OVER (ORDER BY Temp_2m) as dense_rank
FROM weather
ORDER BY temp
LIMIT 8 Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 7: FROM weather
             ^
```

Unterschiede: - **ROW_NUMBER:** Durchnummeriert einfach durch (1, 2, 3, 4, …) – auch bei gleichen Werten! - **RANK:** Gleiche Werte bekommen gleiche Platzierung, danach wird übersprungen (1, 1, 3, 4, …) - **DENSE_RANK:** Gleiche Werte bekommen gleiche Platzierung, aber kein Sprung (1, 1, 2, 3, …)
Welche sollten Sie verwenden? Das hängt vom Kontext ab:

> **Wann welche Funktion?**
>
> - **ROW_NUMBER:** Wenn Sie eindeutige Positionen brauchen (z.B. Paginierung)
> - **RANK:** Klassisches Ranking mit Sprüngen (wie bei Sportplatzierungen)
> - **DENSE_RANK:** Ranking ohne Lücken (z.B. für „Top 10 unterschiedliche Temperaturen“)

LAG & LEAD – Zugriff auf Nachbarzeilen

Jetzt wird es wirklich praktisch! LAG und LEAD erlauben Ihnen, auf vorherige oder nächste Zeilen zuzugreifen. Perfekt für die Frage: „Wie viel wärmer war es heute als gestern?“

Schritt 1: Die Temperatur von gestern holen

```sql
SELECT
  Datum,
  ROUND(Temp_2m, 2) as temp_heute,
  ROUND(LAG(Temp_2m, 1) OVER (ORDER BY Datum), 2) as temp_gestern
FROM weather
ORDER BY Datum DESC
LIMIT 10;
```

```
Error executing statement: SELECT
  Datum,
  ROUND(Temp_2m, 2) as temp_heute,
  ROUND(LAG(Temp_2m, 1) OVER (ORDER BY Datum), 2) as temp_gestern
FROM weather
ORDER BY Datum DESC
LIMIT 10 Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 5: FROM weather
             ^
```

Syntax-Erklärung: - `LAG(Temp_2m, 1)` holt den Wert aus der **vorherigen** Zeile - Die `1` bedeutet: „1 Zeile zurück“ (also gestern) - `OVER (ORDER BY Datum)` sortiert nach Datum, damit „vorherige Zeile“ = „vorheriger Tag“ bedeutet - **Erste Zeile:** Hat keine vorherige Zeile → `NULL`
Jetzt können wir die Veränderung berechnen:

Schritt 2: Temperatur-Veränderung berechnen

```sql
SELECT
  Datum,
  ROUND(Temp_2m, 2) as temp_heute,
  ROUND(LAG(Temp_2m, 1) OVER (ORDER BY Datum), 2) as temp_gestern,
  ROUND(
    Temp_2m - LAG(Temp_2m, 1) OVER (ORDER BY Datum),
    2
  ) as veraenderung
FROM weather
ORDER BY Datum DESC
LIMIT 10;
```

```
Error executing statement: SELECT
  Datum,
  ROUND(Temp_2m, 2) as temp_heute,
  ROUND(LAG(Temp_2m, 1) OVER (ORDER BY Datum), 2) as temp_gestern,
  ROUND(
    Temp_2m - LAG(Temp_2m, 1) OVER (ORDER BY Datum),
    2
  ) as veraenderung
FROM weather
ORDER BY Datum DESC
LIMIT 10 Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 9: FROM weather
             ^
```

Was wir sehen: - Positive Werte = wärmer als gestern - Negative Werte = kälter als gestern - `NULL` = erster Tag (kein Vergleich möglich)
LEAD funktioniert genau umgekehrt – es schaut in die Zukunft!

LEAD – Vorausschau auf morgen

```sql
SELECT
  Datum,
  ROUND(Temp_2m, 2) as temp_heute,
  ROUND(LEAD(Temp_2m, 1) OVER (ORDER BY Datum), 2) as temp_morgen
FROM weather
ORDER BY Datum DESC
LIMIT 10;
```

```
Error executing statement: SELECT
  Datum,
  ROUND(Temp_2m, 2) as temp_heute,
  ROUND(LEAD(Temp_2m, 1) OVER (ORDER BY Datum), 2) as temp_morgen
FROM weather
ORDER BY Datum DESC
LIMIT 10 Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 5: FROM weather
             ^
```

Syntax-Erklärung: - `LEAD(Temp_2m, 1)` holt den Wert aus der **nächsten** Zeile - Ansonsten funktioniert es genau wie LAG - **Letzte Zeile:** Hat keine nächste Zeile → `NULL`

Sie können auch weiter zurück oder voraus schauen:

> **Tipp:** Mit `LAG(Temp_2m, 7)` bekommen Sie die Temperatur von vor 7 Tagen! Nützlich für Wochen-Vergleiche!

Gleitende Mittelwerte – Der Analytics-Klassiker

Jetzt kommt etwas sehr Praktisches: Gleitende Mittelwerte! Stellen Sie sich vor: Temperaturen schwanken täglich wild. Mit einem gleitenden Durchschnitt sehen Sie den echten Trend!

Schritt 1: Das Problem verstehen

```sql
SELECT
  Datum,
  ROUND(Temp_2m, 2) as temp
FROM weather
ORDER BY Datum DESC
```

```
ORDER BY Datum DESC
LIMIT 10;
```

```
Error executing statement: SELECT
  Datum,
  ROUND(Temp_2m, 2) as temp
FROM weather
ORDER BY Datum DESC
LIMIT 10 Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 4: FROM weather
             ^
```

Das Problem: Die Temperatur springt von Tag zu Tag. Heute 8°C, morgen 3°C, übermorgen 11°C. Wo ist der Trend? Schwer zu sehen!
Lösung: Ein 3-Tages-Durchschnitt! Wir nehmen immer die letzten 3 Tage.

Schritt 2: 3-Tages-Gleitender Durchschnitt

```sql
SELECT
  Datum,
  ROUND(Temp_2m, 2) as temp,
  ROUND(
    AVG(Temp_2m) OVER (
      ORDER BY Datum
      ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ),
    2
  ) as temp_3tage_avg
FROM weather
ORDER BY Datum DESC
LIMIT 10;
```

```
Error executing statement: SELECT
  Datum,
  ROUND(Temp_2m, 2) as temp,
  ROUND(
    AVG(Temp_2m) OVER (
      ORDER BY Datum
      ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ),
    2
  ) as temp_3tage_avg
FROM weather
ORDER BY Datum DESC
LIMIT 10 Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 11: FROM weather
              ^
```

Syntax-Erklärung Schritt für Schritt: 1. `AVG(Temp_2m) OVER (...)` = berechne Durchschnitt in einem Fenster 2. `ORDER BY Datum` = sortiere nach Datum 3. `ROWS BETWEEN 2 PRECEDING AND CURRENT ROW` = das Fenster umfasst: - `2 PRECEDING` = die 2 Zeilen davor - `AND CURRENT ROW` = plus die aktuelle Zeile - **Insgesamt: 3 Zeilen!**
Die erste und zweite Zeile haben weniger als 3 Werte – was passiert da?

> **Automatische Anpassung:** - Zeile 1: Nur 1 Wert verfügbar → Durchschnitt von 1 Wert - Zeile 2: Nur 2 Werte verfügbar → Durchschnitt von 2 Werten - Ab Zeile 3: Volle 3 Werte verfügbar → echter 3-Tages-Durchschnitt
>
> SQL passt das Fenster automatisch an!

Jetzt machen wir einen längeren Durchschnitt:

Schritt 3: 7-Tages-Gleitender Durchschnitt

```sql
SELECT
  Datum,
  ROUND(Temp_2m, 2) as temp,
  ROUND(
    AVG(Temp_2m) OVER (
      ORDER BY Datum
      ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ),
    2
  ) as temp_7tage_avg
FROM weather
ORDER BY Datum DESC
```

```
ORDER BY Datum DESC
LIMIT 15;
```

```
Error executing statement: SELECT
  Datum,
  ROUND(Temp_2m, 2) as temp,
  ROUND(
    AVG(Temp_2m) OVER (
      ORDER BY Datum
      ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ),
    2
  ) as temp_7tage_avg
FROM weather
ORDER BY Datum DESC
LIMIT 15 Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 11: FROM weather
              ^
```

Warum 6 PRECEDING? Weil wir 7 Tage wollen: - 6 Tage davor + 1 aktueller Tag = 7 Tage insgesamt - Bei 3 Tagen war es `2 PRECEDING` (2 + 1 = 3) - Bei 30 Tagen wäre es `29 PRECEDING` (29 + 1 = 30) Vergleichen Sie mal die beiden Spalten: temp springt wild, temp*7tage*avg ist viel glatter. Genau das wollen wir!

> **Anwendung:** Gleitende Durchschnitte werden überall verwendet: - Aktienkurse (50-Tage-Durchschnitt) - Infektionszahlen (7-Tage-Inzidenz) - Temperatur-Trends - Verkaufszahlen

---

FIRST_VALUE & LAST_VALUE

Zum Abschluss der Window Functions: `FIRST_VALUE` und `LAST_VALUE`. Diese Funktionen holen den ersten oder letzten Wert aus einem sortierten Fenster. Perfekt, um kälteste und wärmste Tage zu finden!

Kältester und wärmster Tag pro Monat

```
SELECT DISTINCT
  EXTRACT(MONTH FROM Datum) as monat,
  ROUND(
    FIRST_VALUE(Temp_2m) OVER (
      PARTITION BY EXTRACT(MONTH FROM Datum)
      ORDER BY Temp_2m ASC
      ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ), 2
  ) as kaeltester_tag
```

```
  ) as kaeltester_tag,
  ROUND(
    LAST_VALUE(Temp_2m) OVER (
      PARTITION BY EXTRACT(MONTH FROM Datum)
      ORDER BY Temp_2m ASC
      ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ), 2
  ) as waermster_tag
FROM weather
ORDER BY monat;
```

```
Error executing statement: SELECT DISTINCT
  EXTRACT(MONTH FROM Datum) as monat,
  ROUND(
    FIRST_VALUE(Temp_2m) OVER (
      PARTITION BY EXTRACT(MONTH FROM Datum)
      ORDER BY Temp_2m ASC
      ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ), 2
  ) as kaeltester_tag,
  ROUND(
    LAST_VALUE(Temp_2m) OVER (
      PARTITION BY EXTRACT(MONTH FROM Datum)
      ORDER BY Temp_2m ASC
      ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ), 2
  ) as waermster_tag
FROM weather
ORDER BY monat Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 17: FROM weather
              ^
```

Syntax-Erklärung: - `FIRST_VALUE(Temp_2m)` nimmt den **ersten** Wert aus dem sortierten Fenster - `LAST_VALUE(Temp_2m)` nimmt den **letzten** Wert aus dem sortierten Fenster - `ORDER BY Temp_2m ASC` sortiert von kalt (first) nach warm (last) - `PARTITION BY EXTRACT(MONTH FROM Datum)` teilt in Monate auf - `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` definiert das komplette Fenster (wichtig für LAST_VALUE!) - `DISTINCT` entfernt Duplikate (sonst hätten wir eine Zeile pro Tag mit den gleichen Werten)
Das ROWS BETWEEN ist bei LAST_VALUE wichtig – ohne diese Angabe würde SQL nur bis zur aktuellen Zeile schauen!

Alternative mit MIN/MAX (einfacher)

```
SELECT DISTINCT
  EXTRACT(MONTH FROM Datum) as monat,
```

```
  ROUND(MIN(Temp_2m) OVER (
    PARTITION BY EXTRACT(MONTH FROM Datum)
  ), 2) as kaeltester_tag,
  ROUND(MAX(Temp_2m) OVER (
    PARTITION BY EXTRACT(MONTH FROM Datum)
  ), 2) as waermster_tag
FROM weather
ORDER BY monat;
```

```
Error executing statement: SELECT DISTINCT
  EXTRACT(MONTH FROM Datum) as monat,
  ROUND(MIN(Temp_2m) OVER (
    PARTITION BY EXTRACT(MONTH FROM Datum)
  ), 2) as kaeltester_tag,
  ROUND(MAX(Temp_2m) OVER (
    PARTITION BY EXTRACT(MONTH FROM Datum)
  ), 2) as waermster_tag
FROM weather
ORDER BY monat Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 9: FROM weather
             ^
```

Vergleich: - `MIN`/`MAX` sind einfacher und reichen für Min/Max-Werte - `FIRST_VALUE`/`LAST_VALUE` sind flexibler – Sie können nach beliebigen Kriterien sortieren (z.B. Datum)
Wann ist FIRST_VALUE besser? Wenn Sie z.B. die Temperatur des ersten Tages im Monat brauchen – dann sortieren Sie nach Datum!

> **Praxis-Tipp:** Für simple Min/Max nutzen Sie MIN/MAX. Für „ersten/letzten nach Sortierung X“ nutzen Sie FIRST*VALUE/LAST*VALUE!

Teil 3: Praktische Anwendungen

Jetzt kombinieren wir alles: Gleitende Mittelwerte und Anomalie-Erkennung – praktische Analytics!

Anomalie-Erkennung mit Window Functions

Jetzt bauen wir etwas Praktisches: Wir finden Tage, an denen die Temperatur stark vom Durchschnitt abweicht – mögliche Wetterextreme!

Schritt 1: Abweichung berechnen

```
SELECT
```

```sql
  Datum,
  ROUND(Temp_2m, 2) as temp,
  ROUND(
    AVG(Temp_2m) OVER (
      ORDER BY Datum
      ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ),
    2
  ) as temp_7tage_avg,
  ROUND(
    Temp_2m - AVG(Temp_2m) OVER (
      ORDER BY Datum
      ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ),
    2
  ) as abweichung
FROM weather
ORDER BY Datum DESC
LIMIT 10;
```

```
Error executing statement: SELECT
  Datum,
  ROUND(Temp_2m, 2) as temp,
  ROUND(
    AVG(Temp_2m) OVER (
      ORDER BY Datum
      ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ),
    2
  ) as temp_7tage_avg,
  ROUND(
    Temp_2m - AVG(Temp_2m) OVER (
      ORDER BY Datum
      ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ),
    2
  ) as abweichung
FROM weather
ORDER BY Datum DESC
LIMIT 10 Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 18: FROM weather
              ^
```

Was wir sehen: Die Abweichung zeigt, wie sehr ein Tag vom 7-Tage-Durchschnitt abweicht. +5°C = viel wärmer, -5°C = viel kälter.

Jetzt filtern wir nur große Abweichungen – potenzielle Anomalien:

Schritt 2: Nur große Abweichungen anzeigen

```sql
WITH temp_analyse AS (
  SELECT
    Datum,
    ROUND(Temp_2m, 2) as temp,
    ROUND(
      AVG(Temp_2m) OVER (
        ORDER BY Datum
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
      ),
      2
    ) as temp_7tage_avg,
    ROUND(
      Temp_2m - AVG(Temp_2m) OVER (
        ORDER BY Datum
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
      ),
      2
    ) as abweichung
  FROM weather
)
SELECT *
FROM temp_analyse
WHERE abweichung > 4 OR abweichung < -4
ORDER BY abweichung DESC
LIMIT 15;
```

```
Error executing statement: WITH temp_analyse AS (
  SELECT
    Datum,
    ROUND(Temp_2m, 2) as temp,
    ROUND(
      AVG(Temp_2m) OVER (
        ORDER BY Datum
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
      ),
      2
    ) as temp_7tage_avg,
    ROUND(
      Temp_2m - AVG(Temp_2m) OVER (
        ORDER BY Datum
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
      ),
      2
    ) as abweichung
  FROM weather
)
SELECT *
FROM temp_analyse
WHERE abweichung > 4 OR abweichung < -4
ORDER BY abweichung DESC
LIMIT 15 Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 19:   FROM weather
                ^
```

Syntax-Erklärung: - `WITH temp_analyse AS (...)` erstellt eine temporäre Tabelle (CTE = Common Table Expression) - `WHERE abweichung > 4 OR abweichung < -4` filtert Tage mit Abweichung größer als 4°C (in beide Richtungen) - Das sind die Ausreißer – ungewöhnlich warme oder kalte Tage!
Wir können auch Kategorien vergeben:

Schritt 3: Kategorien mit CASE

```sql
WITH temp_analyse AS (
  SELECT
    Datum,
    ROUND(Temp_2m, 2) as temp,
    ROUND(
      AVG(Temp_2m) OVER (
        ORDER BY Datum
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
      ),
```

```
    ),
    2
  ) as temp_7tage_avg,
  ROUND(
    Temp_2m - AVG(Temp_2m) OVER (
      ORDER BY Datum
      ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ),
    2
  ) as abweichung
 FROM weather
)
SELECT
  Datum,
  temp,
  temp_7tage_avg,
  abweichung,
  CASE
    WHEN abweichung > 5 THEN 'Sehr warm'
    WHEN abweichung < -5 THEN 'Sehr kalt'
    WHEN abweichung > 3 THEN 'Warm'
    WHEN abweichung < -3 THEN 'Kalt'
    ELSE 'Normal'
  END as kategorie
FROM temp_analyse
WHERE abweichung > 3 OR abweichung < -3
ORDER BY abweichung DESC
LIMIT 15;
```

```
Error executing statement: WITH temp_analyse AS (
  SELECT
    Datum,
    ROUND(Temp_2m, 2) as temp,
    ROUND(
      AVG(Temp_2m) OVER (
        ORDER BY Datum
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
      ),
      2
    ) as temp_7tage_avg,
    ROUND(
      Temp_2m - AVG(Temp_2m) OVER (
        ORDER BY Datum
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
      ),
      2
    ) as abweichung
  FROM weather
)
SELECT
  Datum,
  temp,
  temp_7tage_avg,
  abweichung,
  CASE
    WHEN abweichung > 5 THEN 'Sehr warm'
    WHEN abweichung < -5 THEN 'Sehr kalt'
    WHEN abweichung > 3 THEN 'Warm'
    WHEN abweichung < -3 THEN 'Kalt'
    ELSE 'Normal'
  END as kategorie
FROM temp_analyse
WHERE abweichung > 3 OR abweichung < -3
ORDER BY abweichung DESC
LIMIT 15 Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 19:   FROM weather
               ^
```

CASE-Syntax: - `CASE WHEN bedingung THEN wert ELSE anderer_wert END` - Prüft Bedingungen von oben nach unten - Erste erfüllte Bedingung gewinnt - `ELSE` ist der Standard-Wert, wenn keine Bedingung zutrifft

---

Zusammenfassung: Monatliche Statistiken

Zum Abschluss kombinieren wir alles: Ein kompletter Monats-Überblick mit allen wichtigen Kennzahlen!

Alle Monats-Statistiken auf einen Blick

```sql
SELECT
  EXTRACT(MONTH FROM Datum) as monat,
  COUNT(*) as anzahl_tage,
  ROUND(AVG(Temp_2m), 2) as durchschnitt,
  ROUND(MIN(Temp_2m), 2) as minimum,
  ROUND(MAX(Temp_2m), 2) as maximum
FROM weather
GROUP BY EXTRACT(MONTH FROM Datum)
ORDER BY monat;
```

```
Error executing statement: SELECT
  EXTRACT(MONTH FROM Datum) as monat,
  COUNT(*) as anzahl_tage,
  ROUND(AVG(Temp_2m), 2) as durchschnitt,
  ROUND(MIN(Temp_2m), 2) as minimum,
  ROUND(MAX(Temp_2m), 2) as maximum
FROM weather
GROUP BY EXTRACT(MONTH FROM Datum)
ORDER BY monat Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 7: FROM weather
             ^
```

Was wir kombiniert haben: - `GROUP BY` für Gruppierung nach Monat - `COUNT(*)` für Anzahl Tage - `AVG()`, `MIN()`, `MAX()` für Statistiken - `ROUND()` für Lesbarkeit - Alles in einer einzigen Query!

So bekommen Sie einen perfekten Überblick über das ganze Jahr!

> **Das haben Sie gelernt:** Aus tausenden Zeilen haben Sie mit ein paar Zeilen SQL aussagekräftige Monats-Statistiken erstellt!

Zusammenfassung & Reflexion

Was für eine Session! Lassen Sie uns zusammenfassen, was Sie heute gelernt haben.

Was Sie heute gelernt haben

1. **Klassische Aggregationen:** `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`, `GROUP BY`, `HAVING`
2. **Window Functions:** `ROW_NUMBER`, `RANK`, `LAG`, `LEAD`, `FIRST_VALUE`, `LAST_VALUE`
3. **Gleitende Mittelwerte:** `ROWS BETWEEN` für flexible Fenster
4. **Zeitbasierte Analytics:** `DATE_TRUNC`, Monats-Aggregationen
5. **Praktische Anwendungen:** Anomalie-Erkennung, Trend-Analyse
6. **CTEs & CASE:** Kombinierte Analytics-Queries

Praktische Übung für Sie

Zum Abschluss eine Aufgabe: Nutzen Sie das Gelernte, um eine eigene Analyse zu bauen!

🎯 Ihre Aufgabe

Erstellen Sie eine Query, die:

1. **Gleitenden 14-Tages-Durchschnitt** für Luftfeuchte berechnet
2. **Tage findet, an denen Luftfeuchte > 95%** (Nebel/Regen-Kandidaten)
3. **Rangfolge** der feuchtesten Tage ausgibt (mit RANK)
4. **Abweichung vom Monatsdurchschnitt** zeigt

Starter-Code:

```sql
WITH humidity_analysis AS (
  SELECT
    Datum,
    Luftfeuchte,
    AVG(Luftfeuchte) OVER (
      ORDER BY Datum
      ROWS BETWEEN 13 PRECEDING AND CURRENT ROW
    ) as feuchte_14tage_avg,
    -- Ihre Erweiterungen hier!
  FROM weather
)
SELECT * FROM humidity_analysis
WHERE Luftfeuchte > 95
ORDER BY Datum DESC
LIMIT 10;
```

```
Error executing statement: WITH humidity_analysis AS (
  SELECT
    Datum,
    Luftfeuchte,
    AVG(Luftfeuchte) OVER (
      ORDER BY Datum
      ROWS BETWEEN 13 PRECEDING AND CURRENT ROW
    ) as feuchte_14tage_avg,
    -- Ihre Erweiterungen hier!
  FROM weather
)
SELECT * FROM humidity_analysis
WHERE Luftfeuchte > 95
ORDER BY Datum DESC
LIMIT 10 Catalog Error: Table with name weather does not exist!
Did you mean "pg_attrdef"?
LINE 10:   FROM weather
               ^
```

Tipp: Kombinieren Sie Window Functions, RANK und Abweichungs-Berechnungen!

Referenzen & Weiterführende Links

Zum Abschluss noch Ressourcen für Ihr Selbststudium.

Aggregationen & Window Functions

- PostgreSQL Window Functions Tutorial
- Modern SQL: Window Functions
- SQL Window Functions Cheat Sheet

DuckDB

- DuckDB Official Docs
- DuckDB SQL Functions
- DuckDB Window Functions

Praktische Tutorials

- Window Functions Explained
- SQL for Data Analysis

🎓 Ende der Lecture 16

Vielen Dank! Sie haben heute mächtige SQL-Werkzeuge kennengelernt. Nutzen Sie Aggregationen und Window Functions für Ihre eigenen Datenanalysen – sie sind in fast jedem Szenario nützlich!

Bis zur nächsten Vorlesung! 🚀 **Tipp:** Experimentieren Sie mit eigenen Daten und Window Functions – die Möglichkeiten sind endlos!