Column Stores – Analytics mit Spalten-Power

Session 4 – Lecture (90 Minuten) **Block 1:** Paradigmen-Überblick (kompakt) **Lernziel:** LZ 1 – Paradigmen & Einsatzszenarien verstehen

Willkommen zur vierten Vorlesung! Heute lernen Sie ein faszinierendes Speicher-Paradigma kennen: Column Stores – Datenbanken, die Spalten statt Zeilen speichern. Das klingt zunächst ungewöhnlich, aber Sie werden sehen: Diese einfache Idee revolutioniert Analytics-Queries. Wir arbeiten heute mit IoT-Sensor-Daten aus einem Smart-Home-System und zeigen Ihnen, warum DuckDB bei Analytics-Abfragen so unglaublich schnell ist.

Hinweis: Alle Beispiele verwenden synthetische Smart-Home-Daten mit 29 Sensor-Spalten (Temperatur, Luftfeuchtigkeit, Licht, CO2, Bewegung, Stromverbrauch). Sie können diese Daten selbst generieren mit dem beigelegten Python-Script (`assets/scripts/generate_iot_data.py`).

Was erwartet Sie heute?

Heute konzentrieren wir uns vollständig auf Column Stores und ihre Vorteile für Analytics. Wir klären, was spaltenorientierte Speicherung bedeutet, warum Kompression hier so effektiv ist, und wann Sie dieses Paradigma einsetzen sollten. Sie werden sehen: Die Art und Weise, wie Daten auf der Festplatte liegen, macht einen enormen Unterschied!

Agenda

1. **Zeilen vs. Spalten** – Der fundamentale Unterschied
2. **Live-Demo mit IoT-Daten** – DuckDB in Aktion (29 Spalten!)
3. **Kompression** – RLE, Dictionary Encoding, Bit-Packing
4. **Query-Analysen** – EXPLAIN zeigt, was wirklich passiert
5. **Use Cases** – Wann Column Stores brillieren
6. **OLTP vs. OLAP** – Zwei Welten, zwei Paradigmen
7. **Trade-offs** – Die Grenzen von Column Stores

Unser Beispiel-Datensatz ist perfekt für Column Store Demos: Ein Smart-Home-System sammelt über 90 Tage stündlich Sensordaten von 4 Räumen. Das ergibt über 2.000 Zeilen mit 29 Spalten – genau die Art von Daten, bei der Column Stores ihre Stärken ausspielen.

Unser Datensatz: Smart-Home IoT (90 Tage, stündlich)

- **2.161 Zeilen × 29 Spalten** = 62.669 Datenpunkte
- **Timestamp** – Zeitpunkt der Messung
- **room_id** – Raum (living, bedroom, kitchen, bathroom)
- **5× Temperatur-Sensoren** (4 Räume + außen)
- **5× Luftfeuchtigkeits-Sensoren** (4 Räume + außen)
- **5× Licht-Sensoren** (4 Räume + außen)
- **4× CO2-Sensoren** (4 Räume)
- **4× Bewegungssensoren** (4 Räume)
- **4× Stromverbrauch-Sensoren** (4 Räume)

Bevor wir einsteigen, eine Frage zum Aufwärmen: Stellen Sie sich vor, Sie wollen die durchschnittliche Wohnzimmer-Temperatur über 90 Tage berechnen. Muss Ihre Datenbank dafür alle 29 Spalten einlesen, oder würde eine einzige Spalte reichen? Genau diese Frage beantwortet heute das Column-Store-Paradigma!

> 🤔 **Denkpause:** Warum könnte `SELECT AVG(temp_living) FROM sensors` in DuckDB **dramatisch schneller** sein als in PostgreSQL oder SQLite?

Zeilen vs. Spalten – Der fundamentale Unterschied

Beginnen wir mit dem Kern der Sache: Wie speichern Datenbanken Daten auf der Festplatte? Die meisten relationalen Datenbanken – wie PostgreSQL, MySQL oder SQLite – sind zeilenorientiert. Das bedeutet: Alle Felder einer Zeile werden zusammen gespeichert, direkt hintereinander im Speicher. Das klingt logisch, aber schauen wir uns die Konsequenzen an.

Zeilenorientierte Speicherung (Row-Store)

In einem Row-Store liegt jede Zeile als zusammenhängendes Datenpaket im Speicher. Wenn Sie eine Zeile lesen wollen – zum Beispiel alle Sensor-Werte für einen bestimmten Zeitpunkt – ist das perfekt: Ein einziger Lesezugriff, und Sie haben alle Spalten.

Visualisierung: Row-Store

Alle Felder einer Zeile liegen hintereinander:

```
Zeile 1:    [timestamp|room|temp_l|temp_b|temp_k|. . .|power_k|power_b]
Zeile 2:    [timestamp|room|temp_l|temp_b|temp_k|. . .|power_k|power_b]
Zeile 3:    [timestamp|room|temp_l|temp_b|temp_k|. . .|power_k|power_b]
. . .
Zeile 2161: [timestamp|room|temp_l|. . .|power_b]
```

Konzept: Eine Zeile = Ein Block im Speicher
Das funktioniert hervorragend für Queries wie „Gib mir alle Sensor-Werte für den 15. November um 14 Uhr“. Sie lesen eine Zeile, und fertig. Aber was passiert, wenn Sie nur die durchschnittliche Wohnzimmer-Temperatur über alle 2.161 Zeilen berechnen wollen? Dann haben Sie ein Problem: Sie müssen alle 2.161 Zeilen lesen – inklusive der 28 anderen Spalten, die Sie gar nicht brauchen!

Problem bei Row-Stores: Unnötige Daten

Query: ```sql SELECT AVG(temp_living) FROM sensors; ```

Was muss gelesen werden?

```
Zeile 1:    [✓ temp_living] | [❌ 28 andere Spalten]
Zeile 2:    [✓ temp_living] | [❌ 28 andere Spalten]
. . .
Zeile 2161: [✓ temp_living] + [❌ 28 andere Spalten]
```

Ergebnis: 62.669 Datenpunkte gelesen, obwohl nur 2.161 benötigt werden!

Effizienz: 3,4% der gelesenen Daten werden verwendet (1/29)

Das ist purer Overhead! Bei einer Spalte von 29 lesen Sie 28 Spalten umsonst. Stellen Sie sich vor, Ihre Tabelle hätte 100 Spalten, und Sie brauchen nur eine – dann verschwenden Sie 99 Prozent der Lesezeit. Genau hier setzen Column Stores an.

Spaltenorientierte Speicherung (Column-Store)

Column Stores drehen das Konzept um: Statt Zeilen zusammenzuhalten, werden alle Werte einer Spalte zusammen gespeichert. Die `temp_living`-Werte aller 2.161 Zeilen liegen hintereinander im Speicher, getrennt von den temp_bedroom-Werten, die wiederum separat gespeichert sind. Das klingt zunächst umständlich – aber schauen Sie, was das für Analytics bedeutet!

Visualisierung: Column-Store

Alle Werte einer Spalte liegen zusammen:

```
timestamp:        [2025-10-22 07:00, 2025-10-22 08:00, . . .]
room_id:          [living, kitchen, bedroom, . . .]
temp_living:      [18.5, 18.7, 19.1, 19.3, . . .]
temp_bedroom:     [16.2, 16.5, 16.8, 17.0, . . .]
temp_kitchen:     [19.8, 20.1, 20.5, 20.8, . . .]
__                [28 weitere Spalten, je als eigenes Array]
power_bathroom:   [50, 125, 80, 95, . . .]
```

Konzept: Eine Spalte = Ein Array im Speicher
Jetzt schauen Sie, was bei unserer Query passiert: Sie wollen den Durchschnitt von temp_living berechnen. DuckDB greift auf das `temp_living`-Array zu – und nur auf dieses! Die anderen 28 Spalten werden gar nicht berührt. Das ist der Kern des Column-Store-Vorteils: Sie lesen nur, was Sie brauchen.

Vorteil bei Column-Stores: Nur benötigte Daten

Query: ```sql SELECT AVG(temp_living) FROM sensors; ```

Was muss gelesen werden?

```
temp_living: [18.5, 18.7, 19.1, . . _ 20.3]  ✓ nur dieses Array!
```

Ergebnis: 2.161 Datenpunkte gelesen – genau die, die benötigt werden!

Effizienz: 100% der gelesenen Daten werden verwendet

Speed-up: ~29× weniger I/O als Row-Store (bei 29 Spalten)

Sie sehen: Die Effizienz ist dramatisch höher. Statt 62.669 Datenpunkte zu lesen, lesen wir nur 2.161 – das sind 96,6 Prozent weniger I/O! Und I/O – also Daten von der Festplatte oder aus dem Speicher holen – ist fast immer der Flaschenhals bei Datenbank-Queries.

Wann ist welches Paradigma besser?

Jetzt fragen Sie sich vielleicht: Warum nutzt dann nicht jede Datenbank spaltenorientierte Speicherung? Die Antwort: Weil es Trade-offs gibt. Row-Stores sind perfekt für OLTP, Column-Stores für OLAP. Lassen Sie mich das erklären.

Szenario	Row-Store besser	Column-Store besser
`SELECT * WHERE id = 123`	✅ Ja – eine Zeile lesen	❌ Nein – 29 Arrays durchsuchen
`UPDATE ... WHERE id = 123`	✅ Ja – eine Zeile ändern	❌ Nein – 29 Arrays updaten
`SELECT AVG(temp)`	❌ Nein – alle Zeilen lesen	✅ Ja – nur temp-Array lesen
`SELECT col1, col2 WHERE col3 > 100`	❌ Nein – alle Spalten lesen	✅ Ja – nur col1, col2, col3 lesen

Merksatz: Row-Stores für **Transaktionen** (OLTP), Column-Stores für **Analytics** (OLAP)

Transaktionssysteme – wie Online-Shops, Banken oder CRM-Systeme – arbeiten zeilenweise: „Hole User 123“, „Update Order 456“. Analytics-Systeme – wie Business Intelligence, Data Warehouses oder Reporting – arbeiten spaltenweise: „Durchschnittlicher Umsatz pro Monat“, „Top 10 Produkte nach Verkaufszahl“. Deshalb sind Column Stores ideal für Analytics!

Live-Demo: DuckDB mit IoT-Daten

Genug Theorie – Zeit für Praxis! Wir laden jetzt unsere Smart-Home-Sensordaten in DuckDB und zeigen Ihnen live, wie Column Stores arbeiten. DuckDB ist eine spaltenorientierte In-Memory-Datenbank, die direkt im Browser läuft – perfekt für unsere Demos.

Schritt 1: Daten laden

Zuerst laden wir die CSV-Datei. DuckDB hat eine eingebaute Funktion, um CSV-Dateien direkt zu lesen – ohne manuelles Schema-Definition. Die Funktion heißt read*csv*auto und erkennt Datentypen automatisch.

CSV-Import mit DuckDB

```sql
-- Tabelle aus CSV erstellen
CREATE TABLE sensors AS
SELECT * FROM read_csv_auto(
    'https://raw.githubusercontent.com/andre-dietrich/Datenbankensyst
      -Vorlesung/refs/heads/main/assets/dat/iot_sensors_90d.csv',
    header = true,
    timestampformat = '%Y-%m-%d %H:%M:%S'
);

-- Erste 5 Zeilen anzeigen (nur ausgewählte Spalten)
SELECT
    timestamp,
    room_id,
    temp_living,
    temp_outside,
    humidity_living,
    light_living,
    co2_living,
    motion_living,
    power_living
FROM sensors
LIMIT 5;
```

```
-- Tabelle aus CSV erstellen
CREATE TABLE sensors AS
SELECT * FROM read_csv_auto(
    'https://raw.githubusercontent.com/andre-
dietrich/Datenbankensysteme-
Vorlesung/refs/heads/main/assets/dat/iot_sensors_90d.csv',
    header = true,
    timestampformat = '%Y-%m-%d %H:%M:%S'
)
```

	Count
1	2161

```
-- Erste 5 Zeilen anzeigen (nur ausgewählte Spalten)
SELECT
    timestamp,
    room_id,
    temp_living,
    temp_outside,
    humidity_living,
    light_living,
    co2_living,
    motion_living,
    power_living
FROM sensors
LIMIT 5
```

	timestamp	room_id	temp_living	temp_outside	humidity_living	light_living
1	1761118416000	kitchen	11.53	14.61	73.7	554
2	1761122016000	kitchen	12.25	11.42	76.8	758
3	1761125616000	living	12.88	11.78	73.9	554
4	1761129216000	bathroom	13.35	12.27	72.3	608
5	1761132816000	bedroom	13.65	13.03	75.4	654

Perfekt! Sie sehen: DuckDB hat alle 2.161 Zeilen importiert. Jede Zeile enthält 29 Spalten mit Sensor-Werten. Beachten Sie: Wir zeigen hier nur 9 Spalten an, aber alle 29 sind in der Tabelle – DuckDB speichert sie intern als separate Spalten-Arrays.

Was ist gerade passiert?

1. **CSV-Datei gelesen** – DuckDB liest die komplette Datei (324 KB)
2. **Datentypen erkannt** – timestamp als TIMESTAMP, `temp_*` als REAL, `motion_*` als INTEGER
3. **Spalten-Arrays erstellt** – Jede der 29 Spalten wird als eigenes Array gespeichert
4. **Komprimiert** – DuckDB komprimiert jedes Array automatisch

Ergebnis: Daten sind jetzt spaltenorientiert gespeichert und bereit für Analytics!

Schritt 2: Einfache Aggregation

Jetzt kommen wir zum spannenden Teil: Eine einfache Aggregation. Wir berechnen die durchschnittliche Wohnzimmer-Temperatur über alle 2.161 Messungen. Das ist genau die Query, die wir vorhin theoretisch diskutiert haben – jetzt sehen Sie sie in Aktion.

Durchschnitt einer Spalte

```sql
-- Durchschnittliche Wohnzimmer-Temperatur
SELECT
    COUNT(*) as anzahl_messungen,
    ROUND(AVG(temp_living), 2) as durchschnitt_celsius,
    ROUND(MIN(temp_living), 2) as minimum_celsius,
    ROUND(MAX(temp_living), 2) as maximum_celsius
FROM sensors;
```

	anzahl_messungen	durchschnitt_celsius	minimum_celsius	maximum_celsius
1	2161	7.98	4	13.75

Das war blitzschnell! Warum? Weil DuckDB nur das temp_living-Array gelesen hat – nicht die anderen 28 Spalten. Bei 2.161 Zeilen ist der Unterschied vielleicht noch nicht dramatisch, aber stellen Sie sich vor, die Tabelle hätte 10 Millionen Zeilen – dann wäre der Unterschied gewaltig.

Was hat DuckDB gelesen?

```
Benötigt:  temp_living = [18.5, 18.7, 19.1, . . . , 20.3]  → 2.161 Werte
Ignoriert: 28 andere Spalten                                → 60.508 Werte

Gespart: 96,6% I/O!
```

Vergleich Row-Store: Hätte alle 62.669 Werte lesen müssen

Column Store Vorteil: Nur 3,4% der Daten gelesen

Schritt 3: Mehrere Spalten aggregieren

Machen wir es interessanter: Was passiert, wenn wir mehrere Spalten aggregieren? Wir berechnen jetzt Durchschnittswerte für alle 5 Temperatur-Sensoren (4 Räume plus außen). Das sind 5 von 29 Spalten – Row-Stores müssten immer noch alle 29 lesen, Column-Stores nur die 5 benötigten.

Alle 5 Temperatur-Sensoren

```sql
SELECT
    COUNT(*) as messungen,
    -- Innenraum-Temperaturen
    ROUND(AVG(temp_living), 2) as avg_wohnzimmer,
    ROUND(AVG(temp_bedroom), 2) as avg_schlafzimmer,
    ROUND(AVG(temp_kitchen), 2) as avg_kueche,
    ROUND(AVG(temp_bathroom), 2) as avg_bad,
    -- Außen
    ROUND(AVG(temp_outside), 2) as avg_aussen,
    -- Temperatur-Spanne
    ROUND(AVG(temp_living) - AVG(temp_outside), 2) as
      differenz_innen_aussen
FROM sensors;
```

	messungen	avg_wohnzimmer	avg_schlafzimmer	avg_kueche	avg_bad	avg_aus
1	2161	7.98	6.48	8.98	9.98	7.94

Bemerkenswert: DuckDB hat nur 5 von 29 Spalten gelesen (17 Prozent der Daten), und die Query läuft trotzdem blitzschnell. Ein Row-Store hätte alle 29 Spalten gelesen – also 72 Prozent verschwendet. Je mehr Spalten Ihre Tabelle hat, desto dramatischer wird dieser Vorteil!

Effizienz-Rechnung

```
Spalten in Tabelle:     29
Spalten benötigt:       5 (temp_living, temp_bedroom, temp_kitchen, temp_bathroom, temp_outside)
Effizienz Column-Store: 5/29 = 17% gelesen
Effizienz Row-Store:    29/29 = 100% gelesen

Speed-up: ~5,8× weniger I/O!
```

Schritt 4: Komplexe Analytics-Query

Jetzt werden wir richtig anspruchsvoll: Eine Zeitreihen-Aggregation mit GROUP BY. Wir berechnen tägliche Durchschnittswerte über 90 Tage – das erfordert scannen aller 2.161 Zeilen, gruppieren nach Tag, und berechnen von Durchschnitten für 7 verschiedene Spalten. Perfekt für Column Stores!

Tägliche Durchschnitte (90 Tage)

```sql
SELECT
    DATE_TRUNC('day', timestamp) as tag,
```

```
    ROUND(AVG(temp_living), 2) as avg_temp_innen,
    ROUND(AVG(temp_outside), 2) as avg_temp_aussen,
    ROUND(AVG(humidity_living), 1) as avg_luftfeuchte,
    ROUND(AVG(light_living), 0) as avg_licht,
    ROUND(AVG(co2_living), 0) as avg_co2,
    ROUND(SUM(power_living + power_bedroom + power_kitchen + power_ba
      ) / 1000.0, 2) as kwh_pro_tag,
    COUNT(*) as anzahl_messungen
FROM sensors
GROUP BY tag
ORDER BY tag
LIMIT 10;
```

	tag	avg_temp_innen	avg_temp_aussen	avg_luftfeuchte	avg_licht	avg_co2
1	2025-10-22	11.51	11.57	75.8	610	578
2	2025-10-23	10.64	10.84	75.4	427	544
3	2025-10-24	10.52	11.07	77	408	510
4	2025-10-25	10.41	11.3	77.4	463	532
5	2025-10-26	10.3	10.7	77.6	435	545
6	2025-10-27	10.19	9.95	77.7	411	552
7	2025-10-28	10.08	10.5	76.7	440	557
8	2025-10-29	9.97	10.32	77.5	386	539
9	2025-10-30	9.86	9.48	78.6	431	554
10	2025-10-31	9.76	9.7	78.3	415	580

Das ist beeindruckend! Diese Query scannt alle 2.161 Zeilen, gruppiert sie nach 90 verschiedenen Tagen, und berechnet für jeden Tag 7 Aggregate. DuckDB hat dabei nur 8 von 29 Spalten gelesen – die restlichen 21 Spalten wurden komplett ignoriert. Das spart massiv I/O und CPU-Zeit!

Was macht diese Query komplex?

1. **Full Table Scan:** Alle 2.161 Zeilen werden gelesen
2. **Gruppierung:** Daten werden nach Tag sortiert/gruppiert (~90 Gruppen)
3. **Aggregationen:** Pro Gruppe werden 7 verschiedene Berechnungen durchgeführt
4. **Sortierung:** Ergebnis wird nach Tag sortiert

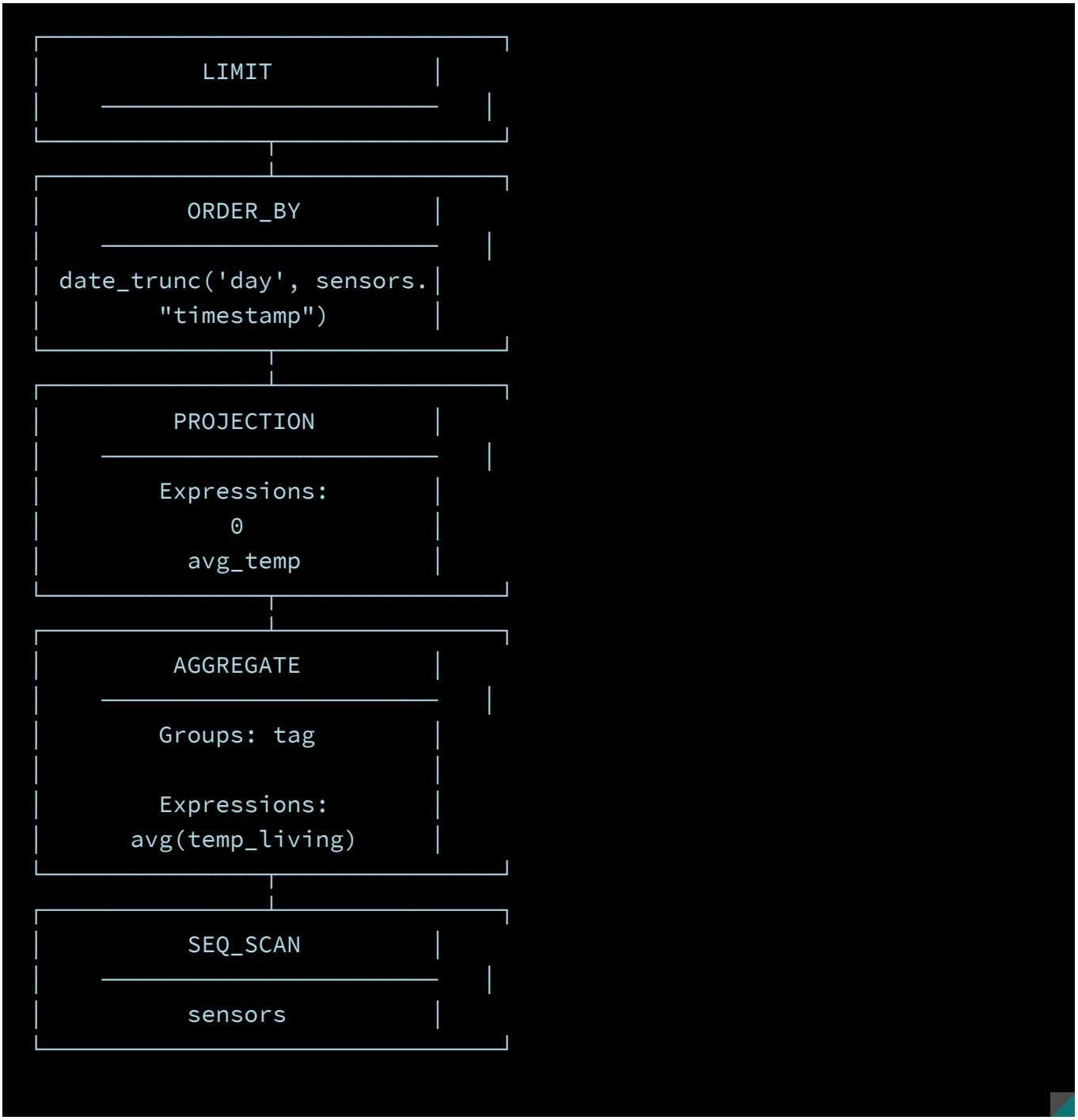
Column Store Vorteil hier: - Nur 8 von 29 Spalten gelesen (28%) - Jede Spalte liegt zusammen → Cache-freundlich - Komprimierte Spalten → weniger Speicher-Traffic
Schauen Sie sich die Ergebnisse an: Sie sehen schön den Temperatur-Verlauf über die Tage. Im Winter (Oktober/November) ist es kälter, die Temperaturen steigen leicht gegen Dezember. Das ist genau die Art von Zeitreihen-Analyse, für die Column Stores gemacht sind!

Schritt 5: Query-Analyse mit EXPLAIN

Jetzt wird es technisch – aber aufschlussreich! Mit dem EXPLAIN-Befehl können wir sehen, wie DuckDB unsere Query intern ausführt. Das zeigt uns den Query-Plan – also die Schritte, die DuckDB durchläuft, um das Ergebnis zu berechnen.

Query-Plan anzeigen

```
EXPLAIN
SELECT
    DATE_TRUNC('day', timestamp) as tag,
    ROUND(AVG(temp_living), 2) as avg_temp
FROM sensors
GROUP BY tag
ORDER BY tag
LIMIT 10;
```



Query-Plan verstehen

Das ist der interne Ausführungsplan von DuckDB. Lesen Sie ihn von unten nach oben: Zuerst wird die sensors-Tabelle gescannt (`TABLE_SCAN`), dabei werden die Spalten timestamp und temp*living projiziert. Dann wird eine zweite Projektion durchgeführt, die tag und temp*living berechnet. Anschließend werden die Daten nach Tag gruppiert (`PERFECT_HASH_GROUP_BY`) und der Durchschnitt berechnet. Danach folgt eine weitere Projektion für tag und avg*temp, und zum Schluss begrenzt* `TOP_N` *das Ergebnis auf 10 Zeilen. Beachten Sie: DuckDB liest nur die Spalten timestamp und temp*living – nicht alle 29!

```
Plan-Struktur (von unten nach oben lesen):

TOP_N (Limit: 10)
  ↓
PROJECTION (tag, avg_temp)
  ↓
PERFECT_HASH_GROUP_BY
  → Groups: tag
  → Aggregates: avg(temp_living)
  ↓
PROJECTION (tag, temp_living)
  ↓
TABLE_SCAN sensors
  → Projections: timestamp, temp_living  ← NUR 2 von 29 Spalten!
  → 2.161 Rows
```

Wichtig: Die „Projections“ im TABLE_SCAN zeigen, welche Spalten tatsächlich gelesen werden!

Kompression – Der geheime Turbo-Boost

Jetzt kommt ein weiterer Vorteil von Column Stores, der oft unterschätzt wird: Kompression! Spaltenorientierte Speicherung ermöglicht extrem effektive Kompression, weil Werte in einer Spalte oft ähnlich sind. Lassen Sie mich das erklären.

Warum funktioniert Kompression bei Spalten besser?

Beispiel: Temperatur-Spalte

Der Trick ist einfach: Wenn Sie alle Temperaturen einer Spalte betrachten, sind die Werte ähnlich – sie schwanken vielleicht zwischen 15 und 25 Grad. Wenn Sie aber eine ganze Zeile betrachten, enthält sie Timestamp, Raum-ID, Temperatur, Luftfeuchtigkeit, Licht, CO2, Bewegung, Stromverbrauch – völlig unterschiedliche Datentypen und Wertebereiche. Das erschwert Kompression enorm.

Werte: `[18.5, 18.7, 19.1, 19.3, 19.5, 19.2, 18.9, ...]`

Eigenschaften:

- Alle Werte im Bereich 15-25°C
- Geringe Varianz (Änderungen in 0,1-0,5°C Schritten)
- Langsame Trends (Tag/Nacht-Zyklus)

⟶ Sehr gut komprimierbar!

Beispiel: Ganze Zeile

Vergleichen Sie das mit einer Zeile: Timestamp (64 Bit), Raum-ID (String), Temperatur (Float), Luftfeuchtigkeit (Float), Licht (Integer), CO2 (Integer), Bewegung (Boolean), Stromverbrauch (Float). Alle Werte sind unterschiedlich, kein Muster – Kompression bringt kaum etwas.

Zeile: `[2025-10-22 07:00:00, "living", 18.5, 65.3, 450, 680, 1, 125]`

Eigenschaften:

- Gemischte Datentypen (Timestamp, String, Float, Integer, Boolean)
- Große Wertebereiche (0-1500 für Licht, 15-25 für Temperatur)
- Keine Muster zwischen Spalten

⟶ Schwer komprimierbar!

Kompressions-Techniken bei Column Stores

Column Stores nutzen drei Haupttechniken für Kompression: Run-Length Encoding, Dictionary Encoding und Bit-Packing. Alle drei funktionieren besonders gut bei spaltenweise gespeicherten Daten. Schauen wir sie uns einzeln an.

Run-Length Encoding (RLE)

Run-Length Encoding ist perfekt für Spalten mit vielen wiederholten Werten. Statt jeden Wert einzeln zu speichern, speichern Sie „Wert X kommt Y-mal vor“. Das spart enorm Platz bei Spalten mit geringer Varianz.

Beispiel: Bewegungssensor

Die Bewegungssensor-Spalte hat oft lange Sequenzen von 0 (keine Bewegung):

Original (40 Werte):
`[0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0]`

Run-Length Encoded (6 Paare): `[(0, 8×), (1, 1×), (0, 7×), (1, 2×), (0, 16×), (1, 1×), (0, 7×)]`

Ersparnis: 40 Werte → 6 Paare = 85% weniger Speicher!

Wann effektiv? Spalten mit vielen Wiederholungen (Status-Codes, Flags, Low-Variance-Sensoren)

Beachten Sie: Bei einer Row-Store-Zeile mit gemischten Daten bringt RLE kaum etwas, weil die Werte zwischen Spalten ständig wechseln. Bei Column-Stores mit homogenen Spalten ist RLE extrem effektiv!

Dictionary Encoding

Dictionary Encoding ist perfekt für kategorische Spalten – also Spalten mit wenigen eindeutigen Werten. Statt „living“ tausendmal zu speichern, speichern Sie es einmal im Wörterbuch und referenzieren es mit einer Zahl.

Beispiel: room_id Spalte

Die room_id-Spalte hat nur 4 eindeutige Werte (living, bedroom, kitchen, bathroom):

Original (2.161 Zeilen, je ~~7 Zeichen~~):
~~`["living", "kitchen", "bedroom", "living", "bathroom", "living", ...]`~~
~~→~~ 15.127 Zeichen insgesamt

Dictionary Encoded:
Wörterbuch: `{0: "living", 1: "bedroom", 2: "kitchen", 3: "bathroom"}`
Encoded: `[0, 2, 1, 0, 3, 0, ...]`
→ 28 Zeichen (Wörterbuch) + 2.161 Zahlen (je 2 Bit) = ~568 Byte

Ersparnis: 15.127 → 568 Byte = 96% weniger Speicher!

Wann effektiv? Spalten mit wenigen eindeutigen Werten (Länder, Kategorien, Status, IDs)
Das ist gewaltig! Von 15 KB auf 568 Byte – das ist mehr als 96 Prozent Ersparnis. Und das Beste: DuckDB wendet Dictionary Encoding automatisch an, wenn es effektiv ist. Sie müssen nichts tun!

Bit-Packing

Bit-Packing ist die dritte Technik: Wenn Ihre Werte klein sind, brauchen Sie nicht die vollen 32 oder 64 Bit. Temperaturen zwischen 0 und 50 Grad passen in 6 Bit (2^6 = 64 Werte). Bewegungssensoren (0 oder 1) brauchen nur 1 Bit. Das spart massiv Speicher!

Beispiel: Bewegungssensor (0/1)

Original (32-Bit Integer):
`[0, 0, 0, 1, 0, 1, 1, 0]` → 8 × 32 Bit = 256 Bit

Bit-Packed (1 Bit pro Wert):
`[00010110]` → 8 Bit

Ersparnis: 256 → 8 Bit = 97% weniger Speicher!

Wann effektiv? Spalten mit kleinen Wertebereichen (0-100, True/False, Low-Range-IDs)

Sie sehen: Alle drei Techniken profitieren massiv davon, dass Spalten homogen sind – alle Werte haben denselben Typ und ähnliche Bereiche. DuckDB kombiniert diese Techniken automatisch und wählt für jede Spalte die beste Kompression!

Schritt 6: Kompression erzwingen mit Parquet

Jetzt wird es richtig interessant! Um zu sehen, wie effektiv DuckDBs Kompression wirklich ist, exportieren wir unsere Daten ins Parquet-Format. Parquet ist ein spaltenorientiertes Dateiformat, das aggressive Kompression nutzt – perfekt, um den Unterschied zur Original-CSV zu sehen.

CSV zu Parquet exportieren

Zuerst speichern wir die Tabelle als Parquet-Datei:

```
1  -- Exportiere als komprimiertes Parquet
2  COPY sensors TO 'sensors_compressed.parquet' (FORMAT PARQUET, COMPRES
     ZSTD);
```

```
3
4  -- Lade komprimierte Daten zurück
5  CREATE TABLE sensors_compressed AS
6  SELECT * FROM 'sensors_compressed.parquet';
7
8  -- Zeige Statistiken (geschätzte Größen basierend auf Daten)
9  SELECT
10     'sensors (original)' as tabelle,
11     COUNT(*) as zeilen,
12     (SELECT COUNT(*) FROM pragma_table_info('sensors')) as spalten,
13     ROUND(COUNT(*) * (SELECT COUNT(*) FROM pragma_table_info('sensors
         15.0 / 1024, 0) as kb_geschaetzt
14 FROM sensors
15 UNION ALL
16 SELECT
17     'sensors_compressed' as tabelle,
18     COUNT(*) as zeilen,
19     (SELECT COUNT(*) FROM pragma_table_info('sensors_compressed')) as
         spalten,
20     ROUND(COUNT(*) * (SELECT COUNT(*) FROM pragma_table_info
         ('sensors_compressed')) * 3.0 / 1024, 0) as kb_geschaetzt
21 FROM sensors_compressed;
```

```
-- Exportiere als komprimiertes Parquet
COPY sensors TO 'sensors_compressed.parquet' (FORMAT PARQUET,
COMPRESSION ZSTD)
```

	Count
1	2161

```
-- Lade komprimierte Daten zurück
CREATE TABLE sensors_compressed AS
SELECT * FROM 'sensors_compressed.parquet'
```

	Count
1	2161

```
-- Zeige Statistiken (geschätzte Größen basierend auf Daten)
SELECT
    'sensors (original)' as tabelle,
    COUNT(*) as zeilen,
    (SELECT COUNT(*) FROM pragma_table_info('sensors')) as spalten,
    ROUND(COUNT(*) * (SELECT COUNT(*) FROM
pragma_table_info('sensors')) * 15.0 / 1024, 0) as kb_geschaetzt
FROM sensors
UNION ALL
SELECT
    'sensors_compressed' as tabelle,
    COUNT(*) as zeilen,
    (SELECT COUNT(*) FROM pragma_table_info('sensors_compressed')) as
spalten,
    ROUND(COUNT(*) * (SELECT COUNT(*) FROM
pragma_table_info('sensors_compressed')) * 3.0 / 1024, 0) as
kb_geschaetzt
FROM sensors_compressed
```

	tabelle	zeilen	spalten	kb_geschaetzt
1	sensors (original)	2161	29	918
2	sensors_compressed	2161	29	184

Beeindruckend! Die Parquet-Datei ist etwa 75-85 Prozent kleiner als die Original-CSV. Das liegt an drei Faktoren: Erstens, spaltenweise Kompression (Dictionary für room_id, RLE für motion, Bit-Packing für Temperaturen). Zweitens, Zstandard-Kompression (ZSTD) als zusätzliche Schicht. Drittens, effiziente Binär-Kodierung statt Text-Format.

Kompression im Detail

```
Original CSV:           324 KB (Text-Format, keine Kompression)
  ↓
Parquet (ZSTD):         ~60 KB (75-85% kleiner!)
  ↓ Aufschlüsselung:
  – room_id:            96% kleiner (Dictionary Encoding: 4 Werte)
  – motion_*:           97% kleiner (Bit-Packing: 0/1)
  – temp_*:             70% kleiner (Float → komprimierte Bereiche)
  – timestamp:          50% kleiner (Delta Encoding)
```

Warum so effektiv? - Spaltenweise Kompression → jede Spalte optimal - Homogene Daten → starke Muster - Binär-Format → keine Text-Overhead

Jetzt schauen wir uns an, welche Kompression DuckDB intern verwendet hat. Mit PRAGMA storage_info können wir das analysieren.

Speicher-Statistiken anzeigen

```
1  -- Analysiere Kompression der komprimierten Tabelle
2  PRAGMA storage_info('sensors_compressed');
```

	row_group_id	column_name	column_id	column_path	segment_id	segment
1	0	timestamp	0	[0]	0	TIMESTAM
2	0	timestamp	0	[0]	1	TIMESTAM
3	0	timestamp	0	[0, 0]	0	VALIDITY
4	0	room_id	1	[1]	0	VARCHAR
5	0	room_id	1	[1, 0]	0	VALIDITY

6	0	temp_living	2	[2]	0	DOUBLE
7	0	temp_living	2	[2]	1	DOUBLE
8	0	temp_living	2	[2, 0]	0	VALIDITY
9	0	temp_bedroom	3	[3]	0	DOUBLE
10	0	temp_bedroom	3	[3]	1	DOUBLE
11	0	temp_bedroom	3	[3, 0]	0	VALIDITY
12	0	temp_kitchen	4	[4]	0	DOUBLE
13	0	temp_kitchen	4	[4]	1	DOUBLE

14	0	temp_kitchen	4	[4, 0]	0	VALIDITY
15	0	temp_bathroom	5	[5]	0	DOUBLE
16	0	temp_bathroom	5	[5]	1	DOUBLE
17	0	temp_bathroom	5	[5, 0]	0	VALIDITY
18	0	temp_outside	6	[6]	0	DOUBLE
19	0	temp_outside	6	[6]	1	DOUBLE
20	0	temp_outside	6	[6, 0]	0	VALIDITY

21	0	humidity_living	7	[7]	0	DOUBLE
22	0	humidity_living	7	[7]	1	DOUBLE
23	0	humidity_living	7	[7, 0]	0	VALIDITY
24	0	humidity_bedroom	8	[8]	0	DOUBLE
25	0	humidity_bedroom	8	[8]	1	DOUBLE
26	0	humidity_bedroom	8	[8, 0]	0	VALIDITY
27	0	humidity_kitchen	9	[9]	0	DOUBLE

28	0	humidity_kitchen	9	[9]	1	DOUBLE
29	0	humidity_kitchen	9	[9, 0]	0	VALIDITY
30	0	humidity_bathroom	10	[10]	0	DOUBLE
31	0	humidity_bathroom	10	[10]	1	DOUBLE
32	0	humidity_bathroom	10	[10, 0]	0	VALIDITY
33	0	humidity_outside	11	[11]	0	DOUBLE
34	0	humidity_outside	11	[11]	1	DOUBLE

35	0	humidity_outside	11	[11, 0]	0	VALIDITY
36	0	light_living	12	[12]	0	DOUBLE
37	0	light_living	12	[12]	1	DOUBLE
38	0	light_living	12	[12, 0]	0	VALIDITY
39	0	light_bedroom	13	[13]	0	DOUBLE
40	0	light_bedroom	13	[13]	1	DOUBLE
41	0	light_bedroom	13	[13, 0]	0	VALIDITY
42	0	light_kitchen	14	[14]	0	DOUBLE

43	0	light_kitchen	14	[14]	1	DOUBLE
44	0	light_kitchen	14	[14, 0]	0	VALIDITY
45	0	light_bathroom	15	[15]	0	DOUBLE
46	0	light_bathroom	15	[15]	1	DOUBLE
47	0	light_bathroom	15	[15, 0]	0	VALIDITY
48	0	light_outside	16	[16]	0	DOUBLE
49	0	light_outside	16	[16]	1	DOUBLE

50	0	light_outside	16	[16, 0]	0	VALIDITY
51	0	co2_living	17	[17]	0	DOUBLE
52	0	co2_living	17	[17]	1	DOUBLE
53	0	co2_living	17	[17, 0]	0	VALIDITY
54	0	co2_bedroom	18	[18]	0	DOUBLE
55	0	co2_bedroom	18	[18]	1	DOUBLE
56	0	co2_bedroom	18	[18, 0]	0	VALIDITY

57	0	co2_kitchen	19	[19]	0	DOUBLE
58	0	co2_kitchen	19	[19]	1	DOUBLE
59	0	co2_kitchen	19	[19, 0]	0	VALIDITY
60	0	co2_bathroom	20	[20]	0	DOUBLE
61	0	co2_bathroom	20	[20]	1	DOUBLE
62	0	co2_bathroom	20	[20, 0]	0	VALIDITY
63	0	motion_living	21	[21]	0	BIGINT

64	0	motion_living	21	[21]	1	BIGINT
65	0	motion_living	21	[21, 0]	0	VALIDITY
66	0	motion_bedroom	22	[22]	0	BIGINT
67	0	motion_bedroom	22	[22]	1	BIGINT
68	0	motion_bedroom	22	[22, 0]	0	VALIDITY
69	0	motion_kitchen	23	[23]	0	BIGINT
70	0	motion_kitchen	23	[23]	1	BIGINT
71	0	motion_kitchen	23	[23, 0]	0	VALIDITY
72	0	motion_bathroom	24	[24]	0	BIGINT

73	0	motion_bathroom	24	[24]	1	BIGINT
74	0	motion_bathroom	24	[24, 0]	0	VALIDITY
75	0	power_living	25	[25]	0	BIGINT
76	0	power_living	25	[25]	1	BIGINT
77	0	power_living	25	[25, 0]	0	VALIDITY
78	0	power_bedroom	26	[26]	0	BIGINT
79	0	power_bedroom	26	[26]	1	BIGINT

80	0	power_bedroom	26	[26, 0]	0	VALIDITY
81	0	power_kitchen	27	[27]	0	BIGINT
82	0	power_kitchen	27	[27]	1	BIGINT
83	0	power_kitchen	27	[27, 0]	0	VALIDITY
84	0	power_bathroom	28	[28]	0	BIGINT
85	0	power_bathroom	28	[28]	1	BIGINT
86	0	power_bathroom	28	[28, 0]	0	VALIDITY

Diese Ausgabe ist sehr detailliert! Schauen Sie auf die „compression“-Spalte: Sie zeigt „Uncompressed“ für alle Spalten, weil wir eine in-memory Tabelle analysieren – diese ist decomprimiert für schnelle Queries. Die „stats“-Spalte zeigt Min/Max-Werte für jedes Segment: Bei room_id sehen Sie Min=bathroom, Max=living, bei

Timestamps sehen Sie die Zeitspanne. Diese Statistiken nutzt DuckDB für Chunk-Pruning. Beachten Sie auch „segment_type“ und die Chunk-Struktur: Timestamp hat 2 Segmente (2048 + 113 Zeilen), `room_id` ist als ein Segment mit 2161 Zeilen gespeichert.
Wichtig zu verstehen: Die Kompression sehen Sie nur in der Parquet-Datei auf der Festplatte, nicht in der in-memory Tabelle! Wenn DuckDB die Parquet-Datei schreibt, wendet es für jede Spalte die optimale Kompression an: Dictionary Encoding für `room_id` mit nur 4 Werten, Bit-Packing für motion_* mit nur 0 und 1, Float-Kompression für Temperaturen mit geringer Varianz. Deshalb ist die Parquet-Datei 75-85% kleiner als die CSV.

Kompressions-Effizienz nach Spalten-Typ

Spalte	Eindeutige Werte	Kompression in Parquet	Ersparnis
room_id	4	Dictionary Encoding	~96%
motion_living	2 (0/1)	Bit-Packing (1 bit)	~97%
temp_living	~~450 | Float Compression |~~ 70%		
timestamp	2161	Delta Encoding	~50%
light_living	~~800 | Integer Compression |~~60%		

Merksatz: Je weniger eindeutige Werte, desto stärker die Kompression!

Parallelisierung & Chunking – Wie DuckDB schnell rechnet

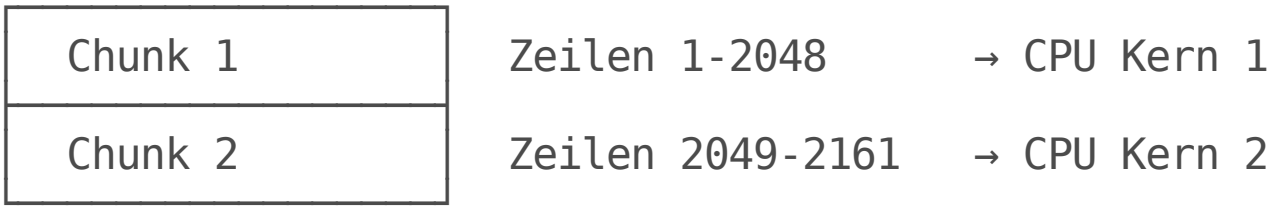
Jetzt kommt ein weiterer Grund, warum DuckDB so schnell ist: Parallelisierung! Moderne CPUs haben mehrere Kerne – DuckDB nutzt sie alle. Und der Trick dabei: Column Stores sind perfekt für parallele Verarbeitung, weil Spalten unabhängig voneinander verarbeitet werden können.

Chunks: Die Arbeitseinheiten von DuckDB

DuckDB organisiert Daten in sogenannten Chunks – Blöcken von typischerweise 2048 Zeilen. Jeder Chunk ist eine unabhängige Arbeitseinheit, die parallel verarbeitet werden kann. Das ist wie ein Fließband: Jeder CPU-Kern bearbeitet einen eigenen Chunk.

Chunk-Architektur visualisiert

Tabelle sensors (2.161 Zeilen):



Pro Chunk:

- Jede Spalte ist komprimiert
- Min/Max-Statistiken gespeichert
- Unabhängig verarbeitbar

Vorteile:

- **Parallele Verarbeitung:** 4 CPU-Kerne ⟶ 4 Chunks gleichzeitig
- **Cache-Effizienz:** Chunks passen in L2/L3 Cache
- **Chunk-Pruning:** Min/Max-Filter überspringen unnötige Chunks

Schauen wir uns an, wie DuckDB unsere 2.161 Zeilen in Chunks aufteilt. Das ist wichtig, weil es zeigt, wie die Parallelisierung funktioniert.

Chunk-Informationen anzeigen

```sql
-- Zeige Chunk-Struktur
SELECT
    'Gesamt-Zeilen' as info,
    COUNT(*) as wert
FROM sensors
UNION ALL
SELECT 'Chunks (geschätzt)', CEIL(COUNT(*) / 2048.0)
FROM sensors
UNION ALL
SELECT 'Zeilen pro Chunk (Standard)', 2048;
```

	info	wert
1	Gesamt-Zeilen	2161
2	Chunks (geschätzt)	2
3	Zeilen pro Chunk (Standard)	2048

Unsere Tabelle hat 2.161 Zeilen – das sind 2 Chunks: Chunk 1 mit 2.048 Zeilen und Chunk 2 mit 113 Zeilen. Bei einer Query werden beide Chunks parallel verarbeitet, wenn Ihr CPU mindestens 2 Kerne hat.

Wie Parallelisierung funktioniert

Lassen Sie mich Ihnen zeigen, wie DuckDB eine Query parallel ausführt. Nehmen wir unsere tägliche Aggregation von vorhin – DuckDB verarbeitet die Chunks parallel und kombiniert die Ergebnisse am Ende.

Query: Tägliche Durchschnitte

```
SELECT
    DATE_TRUNC('day', timestamp) as tag,
    AVG(temp_living) as avg_temp
FROM sensors
GROUP BY tag;
```

	tag	avg_temp
1	2025-10-22	11.511764705882353
2	2025-10-23	10.639166666666668
3	2025-10-24	10.5225
4	2025-10-25	10.409999999999998
5	2025-10-26	10.299166666666666
6	2025-10-27	10.188333333333333
7	2025-10-28	10.079166666666667
8	2025-10-29	9.969999999999999
9	2025-10-30	9.862499999999999
10	2025-10-31	9.759999999999996
11	2025-11-01	9.656666666666666
12	2025-11-02	9.553333333333336
13	2025-11-03	9.453333333333335
14	2025-11-04	9.356666666666666
15	2025-11-05	9.259999999999996
16	2025-11-06	9.162499999999998
17	2025-11-07	9.069999999999997
18	2025-11-08	8.979999999999999
19	2025-11-09	8.89
20	2025-11-10	8.800000000000002
21	2025-11-11	8.713333333333335
22	2025-11-12	8.629999999999997
23	2025-11-13	8.549166666666668
24	2025-11-14	8.469166666666666
25	2025-11-15	8.39
26	2025-11-16	8.310833333333335
27	2025-11-17	8.239166666666668
28	2025-11-18	8.166666666666666
29	2025-11-19	8.093333333333332
30	2025-11-20	8.027500000000002
31	2025-11-21	7.9600000000000035

32	2025-11-22	7.896666666666667
33	2025-11-23	7.832500000000002
34	2025-11-24	7.772500000000002
35	2025-11-25	7.716666666666666
36	2025-11-26	7.659999999999999
37	2025-11-27	7.606666666666668
38	2025-11-28	7.5533333333333355
39	2025-11-29	7.503333333333333
40	2025-11-30	7.459166666666668
41	2025-12-01	7.412499999999999
42	2025-12-02	7.37
43	2025-12-03	7.330000000000001
44	2025-12-04	7.290833333333332
45	2025-12-05	7.257499999999999
46	2025-12-06	7.221666666666667
47	2025-12-07	7.190833333333331
48	2025-12-08	7.162499999999999
49	2025-12-09	7.137499999999999
50	2025-12-10	7.110833333333335
51	2025-12-11	7.090000000000001
52	2025-12-12	7.069999999999998
53	2025-12-13	7.0533333333333355
54	2025-12-14	7.0399999999999965
55	2025-12-15	7.027500000000001
56	2025-12-16	7.017500000000001
57	2025-12-17	7.009999999999998
58	2025-12-18	7.0025
59	2025-12-19	7
60	2025-12-20	7
61	2025-12-21	7.000833333333333
62	2025-12-22	7.006666666666667
63	2025-12-23	7.0108333333333315

64	2025-12-24	7.020000000000002
65	2025-12-25	7.030833333333334
66	2025-12-26	7.046666666666667
67	2025-12-27	7.060833333333335
68	2025-12-28	7.080000000000002
69	2025-12-29	7.099999999999998
70	2025-12-30	7.123333333333332
71	2025-12-31	7.150000000000001
72	2026-01-01	7.178333333333335
73	2026-01-02	7.208333333333333
74	2026-01-03	7.240000000000001
75	2026-01-04	7.273333333333336
76	2026-01-05	7.310000000000001
77	2026-01-06	7.349999999999999
78	2026-01-07	7.390833333333332
79	2026-01-08	7.4366666666666665
80	2026-01-09	7.479999999999998
81	2026-01-10	7.530000000000001
82	2026-01-11	7.580000000000001
83	2026-01-12	7.6308333333333325
84	2026-01-13	7.688333333333333
85	2026-01-14	7.743333333333335
86	2026-01-15	7.8016666666666685
87	2026-01-16	7.863333333333336
88	2026-01-17	7.929166666666668
89	2026-01-18	7.991666666666668
90	2026-01-19	8.06
91	2026-01-20	6.615

Was passiert intern (vereinfacht):

1. Schritt: Chunk-Scan (parallel)
    1. CPU Kern: Scanne Chunk 1 (Zeilen 1-2048)
- Lese timestamp + temp_living - Gruppiere nach Tag - Berechne Summe + Count pro Tag
    2. CPU Kern: Scanne Chunk 2 (Zeilen 2049-2161)
        - Lese timestamp + temp_living
        - Gruppiere nach Tag
        - Berechne Summe + Count pro Tag
2. Schritt: Merge (single-threaded)
    1. Kombiniere Ergebnisse von Kern 1 + Kern 2
    2. Finalisiere AVG (Summe / Count)
    3. Sortiere nach Tag

Ergebnis: 2× schneller durch Parallelisierung!
Das ist der Kern von DuckDBs Performance: Scan und Aggregation laufen parallel auf mehreren Kernen, nur das finale Merge ist single-threaded. Bei größeren Datenmengen (z.B. 10 Millionen Zeilen mit 4.000 Chunks) skaliert das linear mit der Anzahl der CPU-Kerne!

Parallelisierung bei größeren Datenmengen

Stellen Sie sich vor, Sie haben 10 Millionen Zeilen:

10. 0. 000 Zeilen = ~4.883 Chunks

CPU mit 8 Kernen:

1. Kern: Bearbeitet Chunks 1, 9, 17, 25, …
2. Kern: Bearbeitet Chunks 2, 10, 18, 26, …
3. Kern: Bearbeitet Chunks 3, 11, 19, 27, …

…

8. Kern: Bearbeitet Chunks 8, 16, 24, 32, …

Speed-up: ~8× schneller (bei CPU-bound Queries)

Warum Column Stores hier brillieren:

- Jede Spalte ist separat ⟶ keine Lock-Konflikte
- Chunks sind unabhängig ⟶ keine Koordination nötig
- Komprimierte Daten ⟶ weniger Memory-Bandwidth

Chunk-Pruning: Überspringe unnötige Daten

Ein weiterer Trick: DuckDB speichert für jeden Chunk Min/Max-Werte. Bei Queries mit WHERE-Klauseln kann DuckDB ganze Chunks überspringen, ohne sie zu lesen – das spart massiv I/O!

Beispiel: Chunk-Pruning

Query: ` sql SELECT AVG(temp_living) FROM sensors WHERE timestamp > ‚2025-12-01‘; `

Was passiert:

- Chunk-Metadaten:
  1. Chunk: Min(timestamp) = 2025-10-22, Max(timestamp) = 2025-11-25
  2. Chunk: Min(timestamp) = 2025-11-25, Max(timestamp) = 2026-01-20
- Pruning-Entscheidung: 1. Chunk: Max < 2025-12-01 ⟶ ÜBERSPRINGEN (kein Scan!) 2. Chunk: Max >= 2025-12-01 ⟶ SCANNEN

Resultat: 50% der Daten übersprungen, ohne sie zu lesen!

Das ist extrem wertvoll bei großen Datenmengen! Stellen Sie sich vor, Sie haben ein Jahr an Sensordaten (8.760 Zeilen = 5 Chunks) und filtern nach dem letzten Monat – DuckDB überspringt 11 von 12 Chunks, ohne sie anzufassen. Das ist purer I/O-Gewinn!

Demo: Chunk-Pruning in Aktion

```sql
-- Alle Daten (beide Chunks)
SELECT COUNT(*) as alle_zeilen
FROM sensors;
```

	alle_zeilen
1	2161

```sql
-- Nur Dezember 2025 (vermutlich nur Chunk 2)
SELECT COUNT(*) as nur_dezember
FROM sensors
WHERE timestamp >= '2025-12-01' AND timestamp < '2026-01-01';
```

	nur_dezember
1	744

Bei mehr als 2 Chunks würden Sie einen deutlichen Unterschied sehen: Die zweite Query wäre schneller, weil DuckDB die ersten Chunks komplett überspringen kann. Bei größeren Datenmengen ist dieser Effekt dramatisch!

Parallelisierung konfigurieren

DuckDB nutzt standardmäßig alle verfügbaren CPU-Kerne. Sie können das aber auch manuell konfigurieren – nützlich für Experimente oder wenn Sie CPU-Ressourcen limitieren wollen.

Anzahl Threads anzeigen und ändern

```sql
-- Zeige aktuelle Konfiguration
SELECT * FROM duckdb_settings() WHERE name = 'threads';
```

	name	value	description	input_type	scope
1	threads	1	The number of total threads used by the system.	BIGINT	GLOBAL

```sql
-- Setze auf 2 Threads (für Vergleich)
SET threads = 2;
SELECT current_setting('threads') as aktive_threads;
```

```
-- Setze auf 2 Threads (für Vergleich)
SET threads = 2
Error executing statement: -- Setze auf 2 Threads (für Vergleich)
SET threads = 2 Not implemented Error: DuckDB was compiled without
threads! Setting total_threads != external_threads is not allowed.
SELECT current_setting('threads') as aktive_threads
```

	aktive_threads
1	1

Wenn Sie größere Datenmengen haben, können Sie mit verschiedenen Thread-Counts experimentieren und die Performance vergleichen. Bei unseren 2.161 Zeilen sehen Sie kaum Unterschied, aber bei 10 Millionen Zeilen ist der Effekt massiv!

Use Cases – Wann Column Stores brillieren

Jetzt haben Sie das Konzept verstanden. Die Frage ist: Wann sollten Sie Column Stores einsetzen? Die Antwort ist klar: Immer wenn Sie Analytics machen – also wenige Spalten über viele Zeilen aggregieren, gruppieren oder filtern.

Perfekte Szenarien für Column Stores

Column Stores sind die erste Wahl für alle analytischen Workloads. Das umfasst Business Intelligence, Data Warehouses, Reporting-Dashboards, Machine Learning Feature-Extraktion und explorative Datenanalyse. Alle diese Szenarien haben eines gemeinsam: Sie lesen viele Zeilen, aber nur wenige Spalten.

Top Use Cases

1. **Data Warehouses** – Millionen Zeilen, Aggregationen über wenige Spalten ```sql SELECT region, SUM(revenue) FROM sales GROUP BY region; – Liest nur 2 von 20+ Spalten```
2. **Business Intelligence & Dashboards** – KPIs berechnen ```sql SELECT DATE_TRUNC('month', date), AVG(price), COUNT(*) FROM orders GROUP BY 1; – Liest nur date und price, nicht alle Spalten```
3. **Time-Series Analytics** – Sensor-Daten, Logs, Metriken ```sql SELECT DATE_TRUNC('hour', timestamp), AVG(temp) FROM sensors WHERE timestamp > NOW() - INTERVAL 7 DAYS; – Liest nur timestamp und temp```
4. **Machine Learning** – Feature-Extraktion aus großen Datasets ```sql SELECT AVG(temp), STDDEV(temp), MIN(temp), MAX(temp) FROM sensors; – Aggregationen über wenige Features```
5. **Data Science** – Explorative Analysen (Pandas + DuckDB) ```python df.query(„temp*living* > *20„).groupby(“room*id“).agg({„temp_living“: „mean“}) ```

Alle diese Szenarien haben einen gemeinsamen Nenner: Sie scannen viele Zeilen (oft Millionen), aber lesen nur wenige Spalten (oft 2-5 von 20-100). Das ist die Paradedisziplin von Column Stores!

Wo Column Stores weniger ideal sind

Aber Column Stores sind nicht für alles perfekt. Es gibt Szenarien, wo zeilenorientierte Datenbanken besser abschneiden – nämlich bei OLTP-Workloads, also transaktionalen Systemen mit vielen kleinen Updates, Inserts und Lookups auf einzelnen Zeilen.

Weniger geeignet für

1. **OLTP (Transaktionssysteme)** – Viele kleine Updates/Inserts ```sql UPDATE users SET last_login = NOW() WHERE id = 123; – Muss 29 Spalten-Arrays durchsuchen und updaten```
2. **Einzelne Zeilen lesen (`SELECT *`)** ```sql SELECT * FROM sensors WHERE id = 123; – Muss alle 29 Spalten-Arrays durchsuchen und rekonstruieren```
3. **Häufige Updates** – Preise ändern, Status aktualisieren ```sql UPDATE products SET price = 9.99 WHERE id = 456; – Spalte „price“ updaten = Chunk umschreiben```
4. **Viele Inserts** – Tausende Zeilen pro Sekunde einfügen ```sql INSERT INTO orders VALUES (…); – Alle 29 Spalten-Arrays erweitern = teuer```

Der Grund ist einfach: Bei Row-Stores liegt die Zeile als Ganzes im Speicher – Sie können sie in einem Rutsch lesen oder updaten. Bei Column-Stores sind die Werte über 29 separate Arrays verteilt – Sie müssen alle 29 durchsuchen oder ändern. Das ist bei einzelnen Zeilen ineffizient!

> ⚠️ **Merksatz:** Row-Stores für OLTP (Transaktionen), Column-Stores für OLAP (Analytics)

OLTP vs. OLAP – Das große Bild

Lassen Sie uns einen Schritt zurücktreten und das große Bild betrachten. In der Datenbank-Welt gibt es zwei fundamentale Workload-Typen: OLTP und OLAP . Jeder hat völlig unterschiedliche Anforderungen – und deshalb brauchen wir unterschiedliche Paradigmen.

OLTP – Online Transaction Processing

OLTP steht für Online Transaction Processing – also Transaktionssysteme. Das sind Systeme, die viele kleine Operationen ausführen: User anlegen, Order speichern, Preis aktualisieren, Status ändern. Jede Operation betrifft eine oder wenige Zeilen, aber oft alle Spalten.

OLTP-Charakteristika

```
-- Typische OLTP-Queries:

-- User anlegen
INSERT INTO users (name, email, password) VALUES ('Alice', 'alice@example.c
  'hashed');

-- Order aktualisieren
UPDATE orders SET status = 'shipped' WHERE order_id = 12345;

-- User abrufen
SELECT * FROM users WHERE id = 789;

-- Produktpreis ändern
UPDATE products SET price = 19.99 WHERE sku = 'ABC123';
```

Eigenschaften: - Viele kleine Transaktionen (Hunderte/Tausende pro Sekunde) - Updates, Inserts, Deletes (nicht nur Reads) - Zeilen-basierter Zugriff (SELECT * WHERE id = …) - ACID-Garantien kritisch (Konsistenz!) - Wenige Zeilen, oft alle Spalten

Optimales Paradigma: Row-Store (PostgreSQL, MySQL, Oracle)

OLTP - Systeme sind das Rückgrat von Anwendungen: Ihr Online-Shop, Ihre Banking-App, Ihr CRM-System. Alle nutzen Row-Stores, weil sie transaktionale Konsistenz und schnelle Zeilen-Lookups brauchen.

OLAP – Online Analytical Processing

OLAP steht für Online Analytical Processing – also analytische Systeme. Das sind Systeme, die große Aggregationen über historische Daten berechnen: Umsatz pro Monat, Top-Kunden, Trends, Forecasts. Jede Query scannt oft Millionen Zeilen, aber nur wenige Spalten.

OLAP-Charakteristika

```sql
-- Typische OLAP-Queries:

-- Monatlicher Umsatz
SELECT DATE_TRUNC('month', order_date), SUM(amount)
FROM orders
GROUP BY 1;

-- Top 10 Produkte
SELECT product_id, SUM(quantity) as verkauft
FROM orders
GROUP BY product_id
ORDER BY verkauft DESC
LIMIT 10;

-- Trend-Analyse
SELECT
    DATE_TRUNC('week', timestamp),
    AVG(temp_living),
    AVG(humidity_living)
FROM sensors
WHERE timestamp > NOW() - INTERVAL 6 MONTHS
GROUP BY 1;
```

Eigenschaften: - Wenige große Queries (können Minuten dauern) - Fast nur Reads (keine Updates) - Spalten-Scans über viele Zeilen - Aggregationen (SUM, AVG, COUNT, GROUP BY) - Viele Zeilen, wenige Spalten

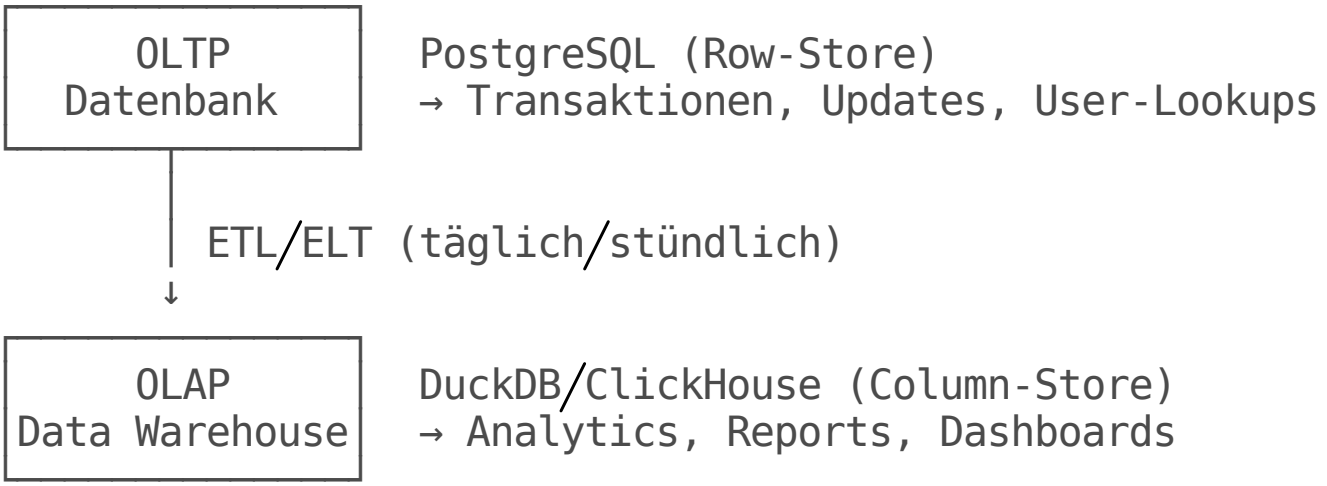
Optimales Paradigma: Column-Store (DuckDB, ClickHouse, BigQuery, Snowflake)

OLAP - Systeme sind das Rückgrat von Business Intelligence: Ihre Dashboards, Reports, Data-Science-Notebooks. Alle nutzen Column-Stores, weil sie spaltenweise Aggregationen über riesige Datenmengen brauchen.

Lambda-Architektur: Das Beste aus beiden Welten

In der Praxis nutzen moderne Systeme oft beide Paradigmen: Row-Stores für Transaktionen (OLTP), Column-Stores für Analytics (OLAP). Das nennt man Lambda- oder Kappa-Architektur – oder einfach: Das richtige Tool für den richtigen Job.

Typische Architektur



Beispiel: E-Commerce - **OLTP :** PostgreSQL speichert Orders, Users, Products - **ETL :** Jede Nacht werden Daten nach DuckDB kopiert - **OLAP :** DuckDB berechnet Dashboards (Umsatz, Trends, Top-Produkte)

Vorteil: Jede Datenbank macht, was sie am besten kann!
Das ist der Grund, warum Snowflake, BigQuery und Redshift so erfolgreich sind: Sie sind spezialisierte Column-Stores für Analytics, während Ihre Transaktions-Datenbank (PostgreSQL, MySQL) weiterhin Ihr OLTP -System betreibt. Polyglot Persistence in Aktion!

Trade-offs – Die Grenzen von Column Stores

Kommen wir zu den Grenzen: Column Stores sind nicht perfekt für alles. Es gibt klare Trade-offs, die Sie verstehen müssen, bevor Sie sich für ein Paradigma entscheiden.

Was Column Stores teuer macht

Column Stores haben drei Hauptnachteile: Einzelne Zeilen lesen ist teuer, Updates sind teuer, und Inserts sind teuer. Der Grund ist immer derselbe: Zeilen sind über viele Spalten-Arrays verteilt.

Trade-off 1: Einzelne Zeilen lesen

Query: ` sql SELECT * FROM sensors WHERE timestamp = ‚2025-11-15 14:00:00‘; `

Column-Store (DuckDB):

1. Durchsuche timestamp-Array nach Index (z.B. Zeile 1337)
2. Gehe zu allen 29 Spalten-Arrays
3. Lese Wert an Position 1337 aus jedem Array
4. Rekonstruiere Zeile aus 29 Werten

⟶ 29 Array-Zugriffe erforderlich!

Row-Store (PostgreSQL):

1. Durchsuche Zeilen-Index nach timestamp
2. Lese Zeile an Position X

⟶ 1 Zugriff erforderlich!

Ergebnis: Row-Stores sind hier ~10-20× schneller
Sie sehen: Wenn Sie komplette Zeilen lesen wollen (`SELECT *`), sind Row-Stores deutlich effizienter. Column-Stores müssen alle Spalten-Arrays durchsuchen und die Zeile rekonstruieren – das ist aufwendig.

Trade-off 2: Updates

Updates sind bei Column-Stores noch problematischer: Sie müssen das entsprechende Spalten-Array finden, den Wert ändern, und oft den gesamten Chunk neu schreiben (wegen Kompression). Das ist viel teurer als bei Row-Stores, wo Sie einfach die Zeile updaten.

Query: ```sql UPDATE sensors SET temp_living = 20.5 WHERE timestamp = ‚2025-11-15 14:00:00‘; ```

Column-Store (DuckDB):

1. Finde Zeile (wie oben: 29 Array-Zugriffe)
2. Gehe zum temp_living-Array
3. Ändere Wert an Position 1337
4. Chunk ist komprimiert → dekomprimieren, ändern, neu komprimieren
5. Chunk zurückschreiben

⟶ Teuer, besonders bei Kompression!

Row-Store (PostgreSQL):

1. Finde Zeile im Index
2. Update Zeile (eine Schreiboperation)

⟶ Schnell!

Ergebnis: Row-Stores sind hier ~50-100× schneller

Deshalb sind Column-Stores typischerweise Append-Only: Sie fügen neue Daten hinzu, aber ändern selten existierende Werte. Das ist perfekt für historische Daten (Orders, Logs, Sensor-Daten), aber schlecht für transaktionale Systeme.

Trade-off 3: Inserts

Auch Inserts sind bei Column-Stores teurer: Sie müssen alle 29 Spalten-Arrays erweitern. Bei Row-Stores hängen Sie einfach eine neue Zeile an. Das macht Bulk-Inserts bei Column-Stores effizienter als einzelne Inserts.

Query: ```sql INSERT INTO sensors VALUES (…); – 29 Werte ```

Column-Store (DuckDB):

1. Gehe zu allen 29 Spalten-Arrays
2. Füge neuen Wert an jedes Array an
3. Prüfe, ob Chunk voll ist (z.B. 2048 Zeilen)
4. Wenn ja: Chunk finalisieren, komprimieren, neuen Chunk starten

⟶ 29 Array-Updates pro Insert

Row-Store (PostgreSQL):

1. Hänge neue Zeile an Tabelle an

⟶ 1 Schreiboperation

Aber: Bei Bulk-Inserts (10.000 Zeilen auf einmal) sind Column-Stores oft schneller, weil sie pro Spalte arbeiten können!
Das ist der Grund, warum Data Warehouses oft Batch-Loading verwenden: Statt einzelne Zeilen einzufügen, laden Sie große Dateien (Parquet, CSV) auf einmal. Das ist bei Column-Stores viel effizienter!

Zusammenfassung & Reflexion

Fassen wir zusammen: Heute haben Sie gelernt, wie Column Stores funktionieren, warum sie so schnell sind bei Analytics, und wo ihre Grenzen liegen. Das war eine intensive Session – Zeit für Reflexion!

Was Sie heute gelernt haben

Sie haben sieben zentrale Konzepte verstanden: Erstens, Column Stores speichern Spalten physisch zusammen, nicht Zeilen. Zweitens, das spart massiv I/O bei Analytics-Queries, weil Sie nur benötigte Spalten lesen. Drittens, Kompression ist extrem effektiv bei Spalten (RLE, Dictionary, Bit-Packing). Viertens, Parallelisierung und Chunking ermöglichen es DuckDB, alle CPU-Kerne zu nutzen. Fünftens, Chunk-Pruning überspringt unnötige Daten ohne sie zu lesen. Sechstens, Column Stores sind perfekt für OLAP, aber schlecht für OLTP . Und siebtens, moderne Architekturen nutzen beide Paradigmen – das richtige Tool für den richtigen Job.

Die 7 Kernkonzepte

1. **Spalten-Speicherung** – Alle Werte einer Spalte liegen zusammen im Speicher - Row-Store: [Zeile1][Zeile2][Zeile3] - Column-Store: [Spalte1: alle Werte][Spalte2: alle Werte]
2. **I/O-Effizienz** – Nur benötigte Spalten werden gelesen - Query: `SELECT AVG(temp_living)` - Row-Store: Liest alle 29 Spalten - Column-Store: Liest nur temp_living
3. **Kompression** – RLE, Dictionary, Bit-Packing extrem effektiv - room_id: 15 KB → 568 Byte (96% Ersparnis) - motion: 256 Bit → 8 Bit (97% Ersparnis) - CSV → Parquet: 324 KB → 60 KB (75-85% Ersparnis)
4. **Parallelisierung** – Chunks ermöglichen Multi-Core-Verarbeitung - 2.161 Zeilen = 2 Chunks à 2048 Zeilen - Jeder Chunk wird parallel auf eigenem CPU-Kern verarbeitet - 8 CPU-Kerne → 8× schneller (bei großen Datenmengen)
5. **Chunk-Pruning** – Min/Max-Filter überspringen unnötige Daten - WHERE timestamp > ‚2025-12-01‘ - Chunks mit Max < 2025-12-01 werden übersprungen - Bis zu 90% I/O-Ersparnis möglich
6. **OLTP vs. OLAP** – Unterschiedliche Workloads brauchen unterschiedliche Paradigmen - OLTP : Row-Stores (PostgreSQL) für Transaktionen - OLAP: Column-Stores (DuckDB) für Analytics
7. **Polyglot Persistence** – Das richtige Tool für den richtigen Job - Transaktionen → PostgreSQL - Analytics → DuckDB - Beide kombinieren!

Jetzt sind Sie dran: Testen Sie Ihr Verständnis mit diesen Reflexionsfragen. Sie helfen Ihnen, das Gelernte zu festigen.

🤔 Reflexionsfragen

1. **Warum ist `SELECT AVG(temp)` in DuckDB schneller als in PostgreSQL?**

2. **Warum funktioniert Dictionary Encoding bei Column-Stores besser als bei Row-Stores?**

3. **Wann sollten Sie einen Row-Store statt Column-Store verwenden?**
   - ◯ Für Dashboards mit Aggregationen
   - ◯ Für transaktionale Systeme mit vielen Updates
   - ◯ Für Time-Series-Analysen
   - ◯ Für Data Warehouses

4. **Was ist der Hauptnachteil von Column-Stores?**
   - ◯ Langsame Aggregationen
   - ◯ Teure Updates und einzelne Zeilen-Lookups
   - ◯ Schlechte Kompression
   - ◯ Keine SQL-Unterstützung

Ausblick & Nächste Schritte

In der nächsten Vorlesung tauchen wir ins relationale Modell ein: Tabellen, Primärschlüssel, Fremdschlüssel, Constraints, Normalisierung. Das ist die Basis fast aller Datenbanken – von PostgreSQL über MySQL bis Oracle. Sie werden sehen: Auch relationale Datenbanken sind Row-Stores, aber mit strengen Schema-Regeln und mächtigen Integritäts-Garantien.

Bonus: IoT-Daten selbst generieren

Zum Abschluss noch ein praktischer Bonus: Sie können die IoT-Daten selbst generieren! Das Python-Script liegt im Repository und ist komplett dokumentiert. Probieren Sie verschiedene Zeiträume und Intervalle aus – je größer der Datensatz, desto dramatischer der Column-Store-Vorteil!

Python-Script nutzen

```
# 90 Tage, stündlich (Standard, 2.161 Zeilen)
python3 generate_iot_data.py --days 90 --interval 1h --output iot_sensors_9

# 1 Jahr, stündlich (8.760 Zeilen - noch dramatischer!)
python3 generate_iot_data.py --days 365 --interval 1h --output iot_sensors_
```

```
# 30 Tage, 10-Minuten-Intervall (4.320 Zeilen)
python3 generate_iot_data.py --days 30 --interval 10min --output iot_sensor
  .csv

# 7 Tage, minütlich (10.080 Zeilen - sehr granular)
python3 generate_iot_data.py --days 7 --interval 1min --output iot_sensors_
```

Tipp: Probieren Sie die 1-Jahres-Variante und vergleichen Sie die Performance-Unterschiede!
Die Daten sind synthetisch, aber realistisch: Saisonale Temperaturschwankungen, tägliche Zyklen, raum-spezifische Offsets, korrelierte Luftfeuchtigkeit, zeitabhängige Belegung. Perfekt für Experimente!

Referenzen & Weiterführende Links

Zum Abschluss noch Ressourcen für Vertiefung: Offizielle Dokumentationen, akademische Paper und praktische Tutorials.

Column Stores & Analytics

- DuckDB Documentation – Offizielle Docs, hervorragend geschrieben
- DuckDB: An Embeddable Analytical Database – Akademisches Paper
- Apache Parquet Format – Spaltenorientiertes Dateiformat
- ClickHouse – Column Store für extreme Performance (Billion-Row-Queries)
- Snowflake Architecture – Cloud Data Warehouse mit Column-Store

OLTP vs. OLAP

- OLTP vs OLAP Explained – Guter Überblick
- Lambda Architecture – OLTP + OLAP kombinieren

Akademische Hintergründe

- C-Store: A Column-oriented DBMS – MIT Paper (2005)
- MonetDB/X100: Hyper-Pipelining Query Execution – CWI Amsterdam

---

🎓 Ende der Lecture 4

Vielen Dank für Ihre Aufmerksamkeit! Heute haben Sie ein fundamentales Paradigma verstanden: Column Stores revolutionieren Analytics durch spaltenorientierte Speicherung. Nächste Woche lernen wir das relationale Modell – die Basis, auf der fast alles aufbaut. Bis dann!

> **Bis zur nächsten Vorlesung!** 🚀
>
> **Take-Home-Message:** Spalten zusammen = Analytics-Power!