DataBlick — Kursvorschau

Ausschnitt aus: Datenbankmodellierung und SQL mit DataBlick Theorie · Teil 1 · Teil 2

Vorwort

Dieses Dokument ist eine Leseprobe aus dem Kurs **Datenbankmodellierung und SQL mit DataBlick**. Es enthält repräsentative Ausschnitte aus drei der vier Kursteile:

- **Theoriekapitel:** Warum Datenbankmodellierung? Wie kommt man vom Sachverhalt zur Tabellenstruktur?
- **Teil 2 / Lektion 1:** Der SQL-Editor in DataBlick — der Einstieg in die Abfragesprache.
- **Teil 2 / Lektion 2:** SELECT — Spalten auswählen, berechnen und benennen.
- **Teil 2 / Lektion 6:** GROUP BY und Aggregatfunktionen — Daten zusammenfassen und auswerten.

Über den Kurs

DataBlick ist eine Lehrumgebung für relationale Datenbanken, die speziell für den Unterricht und für Fortbildungen entwickelt wurde. Es verbindet eine grafische Abfrageoberfläche (QBE-Designer) mit einem vollständigen SQL-Editor und verwendet SQLite als zugrunde liegendes Datenbanksystem.

Der Kurs richtet sich an Lehrkräfte und Lernende ohne Vorkenntnisse in SQL oder Datenbankmodellierung. Die Lektionen bauen schrittweise aufeinander auf. Wer Teil 1 abgeschlossen hat — Import, Tabellenstruktur, einfache QBE-Abfragen — kann mit Teil 2 direkt in SQL einsteigen. Das Theoriekapitel kann parallel oder als Vertiefungsblock verwendet werden.

Was dieses Dokument zeigt

Die ausgewählten Abschnitte sind so gewählt, dass sie einen typischen Querschnitt durch Tiefe, Tempo und Didaktik des Kurses geben:

Das **Theoriekapitel** arbeitet mit konkreten Tabellen und Anomalie-Beispielen, bevor es Begriffe einführt. Das **SQL-Kapitel** stützt sich auf Datenbanken, die im Unterricht entstehen — keine abstrakten Dummy-Daten, sondern eine Kunden-Bestellungen-Tabelle, die im Laufe von Teil 1 per CSV-Import aufgebaut wurde.

Jede Lektion endet mit einer Mini-Aufgabe und einer Selbstkontrolle. Lösungshinweise befinden sich gesammelt im Anhang des jeweiligen Teils — so lässt sich der Lösungsteil bei Bedarf abtrennen.

Theoriekapitel: Datenbankmodellierung

Lektion 1: Warum Datenbankmodellierung?

Lernziel: Du verstehst, warum man eine Datenbank zuerst plant, bevor man Tabellen in DataBlick anlegt.

Eine Datenbank ist nicht einfach eine Sammlung von Tabellen. Eine gute Datenbank bildet einen Ausschnitt der Wirklichkeit so ab, dass die wichtigen Informationen gespeichert, verändert und ausgewertet werden können, ohne dass ständig Widersprüche entstehen.

Stell dir einen kleinen Online-Shop vor, der alle Bestelldaten in einer einzigen Tabelle speichert:

BestellID	Kundename	Stadt	Produktname	Kategorie	Einzelpreis	A
1	Anna Becker	München	USB-Stick 32 GB	Technik	9.99	
1	Anna Becker	München	Webcam HD	Technik	45.99	
2	Ben Fischer	München	Notizbuch A5	Schreibwaren	5.49	
3	Clara Wagner	Berlin	USB-Stick 32 GB	Technik	12.99	

Auf den ersten Blick ist diese Tabelle verständlich. Auf den zweiten Blick erkennt man Probleme: Kundendaten wiederholen sich bei mehreren Bestellungen. Produktdaten wiederholen sich bei jedem Verkauf. Der Lagerbestand eines Produkts steht mehrfach in Bestellzeilen. Das ist nicht nur eine unnötige Wiederholung, sondern auch fachlich falsch: Der Lagerbestand gehört zum aktuellen Zustand eines Produkts, nicht zu einer einzelnen Bestellung. Wenn sich der aktuelle Preis eines Produkts ändert, muss man außerdem unterscheiden: Soll der alte Verkaufspreis in früheren Bestellungen erhalten bleiben oder überschrieben werden?

Solche Probleme nennt man **Anomalien**.

Anomalie	Bedeutung	Beispiel
Änderungsanomalie	Eine Änderung muss an mehreren Stellen vorgenommen werden.	Der Lagerbestand oder Produktname steht in vielen Bestellzeilen.
Einfügeanomalie	Eine Information kann noch nicht sinnvoll gespeichert werden.	Ein neues Produkt kann erst gespeichert werden, wenn es verkauft wurde.
Löschanomalie	Beim Löschen gehen unbeabsichtigt andere Informationen verloren.	Wird die einzige Bestellung eines Produkts gelöscht, verschwindet auch die Produktinformation.

Datenbankmodellierung versucht, diese Probleme durch eine sinnvolle Tabellenstruktur zu vermeiden. Die Grundidee lautet: Jede fachlich eigenständige Sache bekommt eine eigene Tabelle. Informationen werden nicht unnötig wiederholt, sondern über Schlüssel

miteinander verknüpft.

Eine bessere Struktur im Online-Shop ist:

Tabelle	Inhalt
`Kategorie`	Produktgruppen wie Technik, Schule oder Freizeit
`Produkt`	Produktname, aktueller Preis, Lagerbestand, Kategorie
`Kunde`	Kundendaten wie Name, Stadt, Land und Kundengruppe
`Bestellung`	Bestelldatum, Status, Versandkosten und zugehöriger Kunde
`Bestellposition`	Produkt, Anzahl und Einzelpreis innerhalb einer Bestellung

Auffällig ist das Feld `Einzelpreis` in `Bestellposition`: Obwohl der aktuelle Einzelpreis auch beim Produkt gespeichert wird, kann es sinnvoll sein, den tatsächlich verwendeten Preis zusätzlich in der Bestellposition zu speichern. Das ist keine schlechte Redundanz, sondern historische Dokumentation. Wenn ein Produkt später teurer wird, soll eine alte Bestellung ihren damaligen Preis behalten.

Merksatz: Normalisierung bedeutet nicht, jede Wiederholung blind zu entfernen. Es geht darum, ungewollte und fehleranfällige Wiederholungen zu vermeiden. Manchmal werden Werte bewusst gespeichert, weil sie einen Zustand zu einem bestimmten Zeitpunkt dokumentieren.

Beispiel in DataBlick ansehen

Erstelle die Vorlage 🛒 **Online-Shop** über die Startseite von DataBlick. Öffne anschließend das Schema-Diagramm.

Beobachte:

- Die Produktdaten stehen in `Produkt`, nicht in jeder Bestellung.
- Die Kundendaten stehen in `Kunde`, nicht in jeder Bestellung.
- Die Bestellung selbst steht in `Bestellung`.
- Die einzelnen gekauften Produkte stehen in `Bestellposition`.
- Die Tabelle `Bestellposition` verbindet `Bestellung` und `Produkt`.

Damit vermeidet die Datenbank viele Wiederholungen, bleibt aber auswertbar.

Mini-Aufgabe

Markiere in der großen Ein-Tabellen-Darstellung oben drei Informationen, die mehrfach vorkommen und deshalb vermutlich in eigene Tabellen gehören.

Überlege außerdem: Warum wäre es falsch, den Produktnamen und den aktuellen Lagerbestand direkt in jeder Bestellposition dauerhaft zu speichern?

Lektion 2: Vom Sachverhalt zum Datenmodell

Lernziel: Du kannst aus einem beschriebenen Sachverhalt schrittweise ein erstes Datenbankmodell entwickeln.

Datenbankmodellierung beginnt nicht mit DataBlick und nicht mit einem Diagrammwerkzeug. Sie beginnt mit einer fachlichen Beschreibung: einem Text, einem Formular, einem Geschäftsprozess oder einer Beobachtung. Daraus muss man herausarbeiten, welche Informationen dauerhaft gespeichert werden sollen.

Ein bewährtes Vorgehen besteht aus mehreren Schritten:

Schritt	Leitfrage
Sachverhalt lesen	Worum geht es fachlich?
Konkrete Dinge markieren	Welche Personen, Gegenstände oder Vorgänge werden genannt?
Gleichartige Dinge bündeln	Welche Entitätstypen bzw. späteren Tabellen entstehen daraus?
Beziehungen formulieren	Wie hängen diese Entitätstypen zusammen?
Kardinalitäten bestimmen	Ist die Beziehung 1:1, 1:n oder n:m?
Schlüssel festlegen	Wodurch wird jeder Datensatz eindeutig identifiziert?
Attribute zuordnen	Welche Eigenschaften gehören zu welcher Tabelle?

Leitbeispiel: Stadtbibliothek

Sachverhalt:

> Eine Stadtbibliothek verwaltet Bücher, Autorinnen und Autoren, Mitglieder und Ausleihen. Ein Buch hat einen Autor. Ein Autor kann mehrere Bücher geschrieben haben. Ein Mitglied kann im Laufe der Zeit mehrere Bücher ausleihen. Ein Buch kann im Laufe der Zeit mehrfach ausgeliehen werden. Zu jeder Ausleihe werden Ausleihdatum und Rückgabedatum gespeichert. Wenn ein Buch noch nicht zurückgegeben wurde, bleibt das Rückgabedatum leer.

Zuerst markieren wir konkrete Dinge und verallgemeinern sie:

Konkrete Nennung	Entitätstyp / Tabelle
Franz Kafka, Agatha Christie	`Autor`
Der Prozess, Faust I, Mord im Orientexpress	`Buch`
Lena Weber, Max Huber	`Mitglied`
Ausleihe eines Buchs durch ein Mitglied	`Ausleihe`

Die relationale Struktur lautet:

Tabelle	Aufgabe
`Autor`	speichert Autorendaten
`Buch`	speichert Buchtitel und verweist auf `Autor`

Tabelle	Aufgabe
`Mitglied`	speichert Bibliotheksmitglieder
`Ausleihe`	verbindet `Buch` und `Mitglied`, speichert Ausleih- und Rückgabedatum

Das vollständige Stadtbibliothek-Modell ist als Vorlage direkt in DataBlick integriert — es lässt sich mit einem Klick öffnen und im Schema-Diagramm nachvollziehen.

… Das Theoriekapitel behandelt außerdem: Primär- und Fremdschlüssel, Kardinalitäten, Normalisierung (1NF bis 3NF), UML-artige Datenbankdiagramme und die Umsetzung in DataBlick.

Teil 2: SQL in DataBlick und SQLite

Einführung: Von der GUI zur Sprache

Hinter jeder Abfrage, die du im QBE-Entwurf von DataBlick zusammengeklickt hast, stecken SQL-Befehle. DataBlick erzeugt sie automatisch im Hintergrund. In Teil 2 lernen wir, diesen Code direkt zu lesen und zu schreiben.

Das lohnt sich aus mehreren Gründen. Einige Abfragen lassen sich im Abfrageentwurf gar nicht oder nur umständlich formulieren; in SQL geht es kürzer und klarer. SQL-Code ist außerdem kopierbar, dokumentierbar und wiederverwendbar. Wer eine Abfrage als Text vorliegen hat, kann sie später verändern, vergleichen oder in einem anderen Datenbanksystem erneut aufbauen.

Zugleich ist wichtig: SQL ist keine völlig neue Welt. In Teil 1 hast du bereits gefiltert, sortiert, Tabellen verbunden und gruppiert. SQL ist die Sprache hinter diesen Arbeitsschritten. Wir wechseln also nicht das Thema, sondern die Darstellungsform.

DataBlick verwendet **SQLite** — ein echtes relationales Datenbanksystem mit klarer SQL-Syntax. Einige Schreibweisen unterscheiden sich von anderen Systemen und Schulbuchbeispielen; diese Unterschiede werden in den Lektionen jeweils ausdrücklich markiert.

Lektion 1: Der SQL-Editor in DataBlick

Lernziel: Du kannst den SQL-Code einer QBE-Abfrage lesen, sie in eine SQL-Abfrage kopieren, in DataBlick eine SQL-Abfrage öffnen und einfache Anpassungen direkt im Code vornehmen.

Von der Abfrage zum Code

Öffne eine Abfrage aus Teil 1 — zum Beispiel eine Abfrage auf die Tabelle `Kunden`, in der Berliner Kunden mit hohen Gesamtausgaben angezeigt werden. Öffne dann den Reiter **SQL** in der QBE-Ansicht.

Der Code, den DataBlick erzeugt hat, sieht etwa so aus:

```
SELECT Kunden.Vorname, Kunden.Nachname, Kunden.Stadt,
       Kunden.Gesamtausgaben
FROM Kunden
WHERE Kunden.Stadt = 'Berlin' AND Kunden.Gesamtausgaben > 1000;
```

Das ist genau das, was du in der Entwurfsansicht eingestellt hast — nur als Text. Die drei wichtigsten Teile:

SQL-Teil	Bedeutung
`SELECT`	Welche Felder sollen ausgegeben werden?
`FROM`	Aus welcher Tabelle kommen die Daten?
`WHERE`	Welche Bedingung muss erfüllt sein?

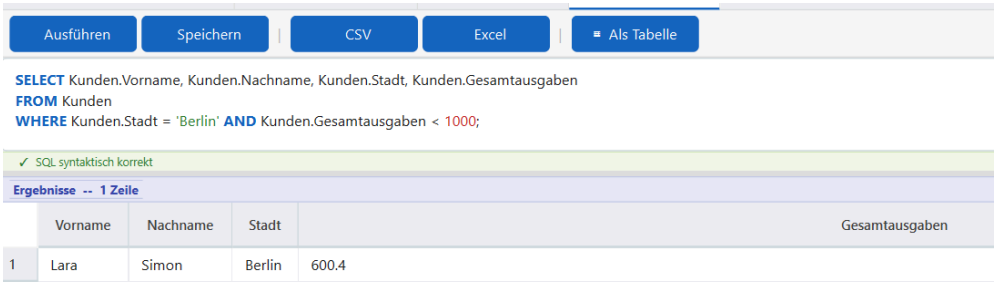
Die SQL-Ansicht als Lesebereich

Die SQL-Ansicht einer QBE-Abfrage ist in DataBlick **schreibgeschützt** — du kannst den Code lesen und kopieren, aber nicht direkt bearbeiten. Der Grund: DataBlick müsste frei geschriebenes SQL wieder in das grafische QBE-Raster übersetzen, was nicht für jede Abfrage möglich ist.

Wenn du die Abfrage anpassen möchtest, gehst du so vor:

1. In der SQL-Ansicht den Code mit **Strg+A, Strg+C** kopieren.
2. Den SQL-Editor öffnen (**Strg+Q** oder Menü *Abfragen → SQL-Editor*).
3. Den Code dort einfügen, anpassen und mit **F5** ausführen.

Alternativ kannst du die QBE-Abfrage über *Rechtsklick → Als SQL-Abfrage speichern* dauerhaft umwandeln. SQL-Abfragen sind immer bearbeitbar.



Bearbeiteter SQL-Code im SQL-Editor von DataBlick

Du siehst dabei zwei Unterschiede zur QBE-Ansicht: Die Syntax wird während der Eingabe rudimentär geprüft, und nach dem Ausführen erscheinen Code und Ergebnistabelle gemeinsam im selben Fenster.

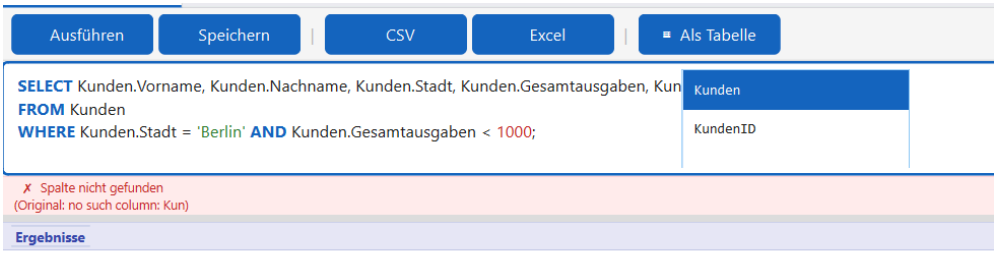
IntelliSense — Autovervollständigung im SQL-Editor

Sobald du drei oder mehr Zeichen getippt hast, schlägt DataBlick passende Begriffe vor. Die Vorschläge umfassen drei Kategorien:

- **SQL-Schlüsselwörter** wie `SELECT`, `WHERE`, `GROUP BY`, `HAVING` (kursiv dargestellt)
- **SQL-Funktionen** wie `COUNT`, `ROUND`, `STRFTIME` (ebenfalls kursiv)
- **Tabellen- und Feldnamen** aus der aktuell geöffneten Datenbank (normal dargestellt)

Die Vorschläge sind kontextsensitiv: Nach `FROM` und `JOIN` erscheinen nur Tabellennamen. Nach einem Punkt hinter einem Tabellennamen — z. B. `Bestellung.` — erscheinen sofort die Felder dieser Tabelle, ohne Mindestzeichenzahl.

Mit der Pfeiltaste wählst du einen Vorschlag aus, mit `Tab` oder `Return` übernimmst du ihn. `Esc` schließt die Liste.



IntelliSense-Dropdown im SQL-Editor mit Feldvorschlägen

IntelliSense verhindert viele Tippfehler, bevor sie entstehen. Wenn du den Feldnamen `Rueckgabedatum` nicht genau weißt, tippst du `Rue` und lässt dir die Möglichkeiten anzeigen.

Automatische Syntaxprüfung

Während du tippst, prüft DataBlick den SQL-Text fortlaufend auf offensichtliche Syntaxfehler. Solange alles in Ordnung ist, bleibt der Bereich unterhalb des Editors leer. Sobald ein Problem erkannt wird — eine fehlende Klammer, ein falsch geschriebenes Schlüsselwort — erscheint ein roter Hinweis direkt darunter. Während du noch eingibst, ist das häufig der Fall — kein Grund zur Beunruhigung.

Die Syntaxprüfung findet viele formale Fehler und oft auch unbekannte Tabellen- oder Feldnamen. Die fachliche Richtigkeit der Abfrage prüft sie nicht.

SQLite-Schreibweisen, die immer wieder auftreten

Thema	SQLite/DataBlick-Syntax	Hinweis
Textwerte	`'Berlin'`	Immer in einfache Anführungszeichen
Datumswerte	`'2023-01-01'`	ISO-Format; DataBlick speichert Datumswerte als sortierbaren ISO-Text
Wildcards in LIKE	`LIKE 'M%'`	% steht für beliebig viele Zeichen, _ für genau ein Zeichen
Ergebnis begrenzen	`LIMIT 5`	Steht am Ende der Abfrage

Mini-Aufgabe

Öffne eine Abfrage aus Teil 1 und schaue dir den erzeugten SQL-Code an. Kopiere sie dann als eigene SQL-Abfrage. Ändere im SQL-Editor einen Wert in der `WHERE`-Bedingung — zum Beispiel eine Stadt — und führe die Abfrage aus.

Notiere: Was hat sich im SQL-Code geändert? Hat sich die ursprüngliche QBE-Abfrage dadurch auch verändert?

Lektion 2: SELECT — Spalten auswählen und benennen

> 📋 **Datenmodell: Kunden & Bestellungen** | Tabelle | Felder | |—|—| | Kunden | KundenID · Vorname · Nachname · Stadt · Email · Anmeldedatum · Gesamtausgaben | | Bestellungen | BestellID · **KundenID** · Bestelldatum · Betrag · Produkt |

Lernziel: Du kannst eine SELECT-Abfrage von Grund auf schreiben, Spalten gezielt auswählen, mit Aliasnamen versehen und das Ergebnis auf eine bestimmte Anzahl von Zeilen begrenzen.

Die Grundstruktur

```
SELECT Spalte1, Spalte2
FROM Tabelle;
```

Das Semikolon am Ende ist in DataBlick optional, aber gute Praxis. Großschreibung der SQL-Schlüsselwörter ist ebenfalls optional, verbessert aber die Lesbarkeit deutlich.

Alle Spalten auswählen:

```
SELECT *
FROM Kunden;
```

Das * bedeutet „alle Spalten". Für schnelle Übersichten ist das praktisch. In fertigen Abfragen ist eine konkrete Feldliste besser — das Ergebnis ändert sich sonst unbemerkt, wenn später neue Spalten zur Tabelle hinzukommen.

Bestimmte Spalten auswählen:

```
SELECT Vorname, Nachname, Stadt
FROM Kunden;
```

Aliasnamen mit AS

Spalten können im Ergebnis umbenannt werden:

```
SELECT Vorname, Nachname, Gesamtausgaben AS Umsatz
FROM Kunden;
```

Im Ergebnis erscheint die Spalte `Gesamtausgaben` unter dem Namen `Umsatz`. Aliasnamen sind besonders bei berechneten Spalten und Aggregatfunktionen wichtig.

Eindeutige Werte: DISTINCT

Mit `DISTINCT` werden doppelte Zeilen aus dem Ergebnis entfernt:

```
SELECT DISTINCT Stadt
FROM Kunden;
```

Ohne `DISTINCT` erscheint jede Stadt so oft, wie es Kunden aus dieser Stadt gibt. Mit `DISTINCT` erscheint sie genau einmal. Besonders nützlich, wenn du wissen willst, welche Werte überhaupt vorkommen.

`DISTINCT` wirkt auf die gesamte ausgegebene Zeile: `SELECT DISTINCT Vorname, Stadt` zeigt jede Kombination aus Vorname und Stadt genau einmal, nicht jeden Vornamen einmal.

Ergebnisse begrenzen: LIMIT

SQLite kennt kein `TOP`. Stattdessen steht `LIMIT` am Ende der Abfrage:

```
SELECT Vorname, Nachname, Gesamtausgaben
FROM Kunden
ORDER BY Gesamtausgaben DESC
LIMIT 5;
```

Ohne `ORDER BY` ist nicht sinnvoll festgelegt, welche Zeilen angezeigt werden. Deshalb kombiniert man `LIMIT` fast immer mit einer Sortierung. Bei Gleichständen kann eine zweite Sortierspalte die Auswahl stabil machen:

```
SELECT Vorname, Nachname, Gesamtausgaben
FROM Kunden
ORDER BY Gesamtausgaben DESC, Nachname ASC
LIMIT 5;
```

Berechnete Spalten

Einfache Rechenoperationen können direkt in `SELECT` stehen:

```
SELECT Vorname, Nachname, Gesamtausgaben * 0.19 AS Mehrwertsteuer
FROM Kunden;
```

Texte werden in SQLite mit `||` zusammengesetzt:

```
SELECT Vorname || ' ' || Nachname AS VollerName, Stadt
FROM Kunden;
```

Ergebnisse -- 200 Zeilen

	VollerName	Stadt
1	Anna Müller	Berlin
2	Peter Schmidt	München
3	Sabine Klein	Hamburg

Ergebnis mit zusammengesetzter Namensspalte

Damit die Mehrwertsteuer immer mit zwei Nachkommastellen erscheint:

```
SELECT Vorname, Nachname,
       PRINTF('%.2f', Gesamtausgaben * 0.19) || ' €' AS Mehrwertsteuer
FROM Kunden;
```

> **Hinweis — unerwartete Nachkommastellen:** `ROUND()` gibt eine Dezimalzahl zurück. Beim Verketten mit `||` wandelt SQLite diese intern in Text um — nach der Regel „keine überflüssigen Nullen". Dabei erscheint `285.10` als `285.1`, aber

> `475.00` als `475.0`. Das sieht inkonsistent aus. `PRINTF('%.2f', ...)` erzeugt direkt Text mit exakt zwei Nachkommastellen und umgeht dieses Problem.

Mini-Aufgabe

Erstelle eine Abfrage, die Vor- und Nachname als eine Spalte `Name` ausgibt, daneben die Stadt und die Gesamtausgaben. Begrenze das Ergebnis auf die drei Kunden mit den höchsten Gesamtausgaben.

Selbstkontrolle

Die Spaltenüberschrift für den zusammengesetzten Namen lautet `Name`. Es erscheinen genau 3 Zeilen. Die Werte in `Gesamtausgaben` sind absteigend sortiert.

Lektion 6: GROUP BY und Aggregatfunktionen *(Ausschnitt)*

Lernziel: Du kannst Daten in SQL gruppieren, Aggregatfunktionen einsetzen und mit `HAVING` auf aggregierte Werte filtern.

Bisher haben alle Abfragen einzelne Zeilen zurückgegeben — eine Zeile pro Kunde, pro Produkt, pro Bestellung. Manchmal interessiert aber nicht die Einzelzeile, sondern eine Zusammenfassung: Wie viele Bestellungen hat ein Kunde insgesamt? Was ist der durchschnittliche Bestellwert? Wie hoch ist der Gesamtumsatz pro Stadt?

Genau dafür gibt es Aggregatfunktionen. Sie fassen mehrere Zeilen zu einem einzigen Wert zusammen. Mit `GROUP BY` lässt sich festlegen, nach welchem Kriterium gruppiert werden soll — so erhält man nicht den Gesamtumsatz aller Kunden zusammen, sondern den Umsatz pro Stadt oder pro Kundengruppe.

Aggregatfunktionen

Funktion	Bedeutung
`COUNT(*)`	Anzahl aller Zeilen
`SUM(Feld)`	Summe
`AVG(Feld)`	Durchschnitt — oft sinnvoll mit `ROUND(AVG(...), 2)`
`MIN(Feld)`	Kleinster Wert
`MAX(Feld)`	Größter Wert

GROUP BY — Gruppierung nach Datum

Oft will man Umsätze nicht pro Kunde, sondern pro Zeitraum auswerten. In SQLite extrahiert `strftime()` Teile eines Datums als Text:

- `strftime('%Y', Bestelldatum)` → Jahr, z. B. `'2024'`
- `strftime('%m', Bestelldatum)` → Monat mit führender Null, z. B. `'03'`

Damit lässt sich der monatliche Umsatz berechnen — gruppiert nach Jahr und Monat, sortiert chronologisch:

```sql
-- Monatlicher Bestellumsatz, chronologisch sortiert
SELECT strftime('%Y', Bestellungen.Bestelldatum) AS Jahr,
       strftime('%m', Bestellungen.Bestelldatum) AS Monat,
       COUNT(*)                                  AS
        Anzahl_Bestellungen,
       ROUND(SUM(Bestellungen.Betrag), 2)        AS Monatsumsatz
FROM Bestellungen
GROUP BY Jahr, Monat
ORDER BY Jahr ASC, Monat ASC;
```

Die Reihenfolge in `GROUP BY` bestimmt, wie kombiniert wird: zuerst Jahr, dann Monat. `ORDER BY` bestimmt die Anzeigefolge — hier chronologisch aufsteigend.

Möchte man den Monat als Wort ausgeben statt als Zahl, braucht man `CASE WHEN` — eine Technik aus Lektion 7:

```sql
SELECT strftime('%Y', Bestellungen.Bestelldatum) AS Jahr,
       CASE strftime('%m', Bestellungen.Bestelldatum)
           WHEN '01' THEN 'Januar'   WHEN '02' THEN 'Februar'
           WHEN '03' THEN 'März'     WHEN '04' THEN 'April'
           WHEN '05' THEN 'Mai'      WHEN '06' THEN 'Juni'
           WHEN '07' THEN 'Juli'     WHEN '08' THEN 'August'
           WHEN '09' THEN 'September' WHEN '10' THEN 'Oktober'
           WHEN '11' THEN 'November' WHEN '12' THEN 'Dezember'
       END AS Monat,
       COUNT(*)                              AS Anzahl_Bestellungen,
       ROUND(SUM(Bestellungen.Betrag), 2) AS Monatsumsatz
FROM Bestellungen
GROUP BY Jahr, strftime('%m', Bestellungen.Bestelldatum)
ORDER BY Jahr ASC, strftime('%m', Bestellungen.Bestelldatum) ASC;
```

> **Achtung:** `GROUP BY` und `ORDER BY` müssen weiterhin auf `strftime('%m', ...)` (die Zahl) laufen, nicht auf den Monatsnamen — sonst sortiert SQLite alphabetisch, und April käme vor Februar.

Ergebnisse -- 20 Zeilen

	Jahr	Monat	Anzahl_Bestellungen	Monatsumsatz
1	2023	Januar	10	2160.0
2	2023	Februar	10	1185.0
3	2023	März	10	1065.0
4	2023	April	10	1080.0
5	2023	Mai	10	1080.0

Ergebnis der Monatsauswertung mit ausgeschriebenen Monatsnamen

… Lektion 6 behandelt außerdem: HAVING als Filter nach der Gruppierung, die logische Ausführungsreihenfolge WHERE → GROUP BY → HAVING, und Transferbeispiele aus Schulverwaltung und Stadtbibliothek.

Ende der Leseprobe.

Der vollständige Kurs umfasst: - Theoriekapitel: 8 Lektionen (Modellierung, Kardinalitäten, Normalisierung, UML-Diagramme) - Teil 1: DataBlick-Grundlagen, CSV-Import, QBE-Abfragen, Fremdschlüssel - Teil 2: SQL-Lektionen 1–9 plus 6 Vertiefungen (CTEs, Fensterfunktionen, Pivot-Tabellen, strftime, Unterabfragen, UNION) - Teil 3: ***DML (Data Manipulation Language)*** *und Praxisprojekte*