Projekt Big- Semi und Unstructured Data

Projektteam	Anna Werlen, Leunora Ristemi, Vivien Aregger
Dozent	Marc Schaaf
Ort, Datum	Olten, 31.10.2025

Inhaltsverzeichnis

Inhaltsverzeichnis II

1 Scenario 1

1.1 Stakeholders 1

1.2 Triggers 2

2 Analytische Fragen 3

1. Welche Produktkategorien generieren den höchsten Umsatz? 3

2. Wie viel Umsatz pro Kunde (Customer Lifetime Value) wurde über alle Bestellungen hinweg erzielt? 3

3 Scenario / Data / Technology 5

3.1 Technische Umsetzung (Mongo DB Aggregationen) 5

3.1.1. Welche Produktkategorien generieren den höchsten Umsatz? 5

3.1.2. Wie viel Umsatz pro Kunde (Customer Lifetime Value) wurde über alle Bestellungen hinweg erzielt? 9

4 Event Generation / Event Processing 18

5 Resultate und Diskussion 26

5.1 Analytische Frage 1 26

5.2 Analytische Frage 2 26

5.3 Resultate gesamt 27

6 Reflexion 28

7 Anhang 29

1 Scenario

GlowLab ist ein fiktiver Online-Shop für hochwertige Kosmetikprodukte wie Seren, Cremes und Masken. Das Unternehmen verkauft direkt an Endkund:innen über einen Webshop in mehreren Ländern (z. B. Schweiz, Deutschland, Österreich).

Ziel ist es, durch datenbasierte Analysen zu verstehen,

- welche Produktkategorien den höchsten Umsatz generiert
- und wie viel Umsatz pro Kunde (Customer Lifetime Value) über alle Bestellungen hinweg erzielt wurde

Diese Informationen unterstützen GlowLab dabei, Sortiment, Marketingstrategien und Kampagnen gezielter auf Zielgruppen abzustimmen.

1.1 Stakeholders

Stakeholder	Rolle / Interesse	Informations- bzw. Analysebedarf
Management / Geschäftsleitung	Strategische Steuerung & Umsatzmaximierung	Welche Kategorien und Kundinnen sind am wertvollsten?
Marketing Team	Kampagnensteuerung	Wer sind unsere umsatzstärksten Kunden?
		Wie lässt sich die Kundenbindung verbessern?
E-Commerce Operations	Produktmanagement & Preisstrategie	Welche Kategorien generieren den höchsten Umsatz?
Business Intelligence	Datenintegration & KPI-Berechnung	Aufbau einer konsistenten Datenbasis für CLV-Analysen

1.2 Triggers

Kontext	Beschreibung
Reporting	CLV- und Umsatzberichte pro Kategorie
Kampagnenplanung (Marketing)	Zielkundenanalyse für Rabatt- oder Loyalty-Aktionen
Sortimentsbewertung	Entscheidung über Produktlinien basierend auf Umsatzbeiträgen
Velocity-Anwendungen (Echtzeit)	Überwachung von Event-Daten

2 Analytische Fragen

1. Welche Produktkategorien generieren den höchsten Umsatz?

   Das kann als Grundlage für Produktstrategie & Sortimentsplanung genutzt werden.

Kategorie	Gesamtumsatz (CHF)	Verkaufte Menge
Cremen	681.42	26
Seren	380.52	18
Masken	300.60	20

Interpretation: Cremen dominieren das Umsatzvolumen. Das Sortiment und Marketing sollten sich verstärkt auf Cremen konzentrieren.
Mögliche Massnahmen:

- Fokus auf Upselling und Cross-Selling von Cremen-Produkten,
- Erweiterung der Produktlinie (neue Varianten, Grössen oder Düfte),
- gezielte Promotion-Kampagnen, da hohe Nachfrage und Zahlungsbereitschaft bestehen.

2. Wie viel Umsatz pro Kunde (Customer Lifetime Value) wurde über alle Bestellungen hinweg erzielt?

Kunde	Bestellungen	Umsatz (CHF)	Durchschnittlicher Umsatz pro Bestellung
Emma Durand (ID 5)	5	336.50	67.30
Jonas Keller (ID 10)	4	228.26	57.07
Fabio Rossi (ID 6)	4	207.20	51.80
Gloria Martinez (ID 7)	5	131.52	26.30
Clara Case (ID 3)	4	123.20	30.80
Isabella Johnson (ID 9)	3	121.22	40.41
Alberto Rossi (ID 8)	3	83.70	27.90
Ben Beispiel (ID 2)	2	44.66	22.33
David Huber (ID 4)	3	43.28	14.43
Anna Muster (ID 1)	2	43.00	21.50

Interpretation: Nur 2–3 Kunden (z. B. Emma Durand, Jonas Keller, Fabio Rossi) machen über die Hälfte des gesamten Umsatzes aus.

Das ist genau das, was die Pareto-Regel beschreibt: Ein kleiner Anteil der Kunden (ca. 20 %) generiert den grössten Teil des Umsatzes (ca. 80 %).

Kundenbindung und personalisierte Angebote für diese Gruppe könnten den CLV weiter erhöhen. Kunden mit niedrigem CLV sollten gezielt mit Anreizen zu Wiederkäufen (Rabatte, Newsletter, Cross-Selling) aktiviert werden.

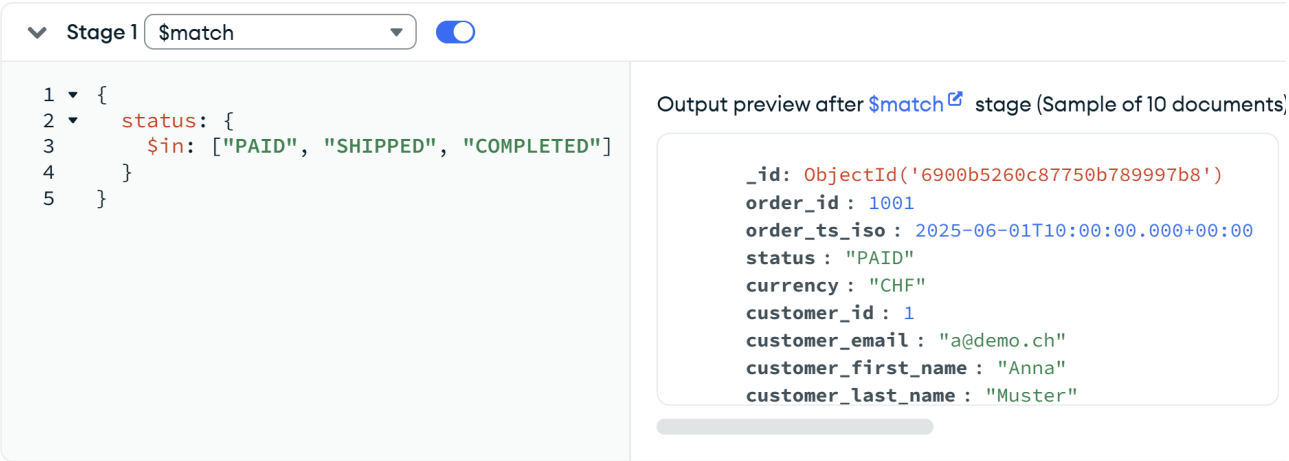
3 Scenario / Data / Technology

Schritt	Beschreibung
MySQL	Normalisierte Tabellenstruktur (3NF) mit Customers, Orders, Products, Order_Items, Categories
Export	SQL-Join erstellt flache CSV-Dateien für die Collections `customers`, `products` und `order_items` (jeweils 1 Zeile = 1 Kunde, 1 Produkt bzw. 1 Bestellposition).
MongoDB	CSV importiert in group2 zugeordnet
Aggregations in MongoDB Compass	Umsatz pro Kategorie und CLV pro Kunde
Velocity (Kafka)	Simulation von Events zur Echtzeitüberwachung

3.1 Technische Umsetzung (Mongo DB Aggregationen)

3.1.1. Welche Produktkategorien generieren den höchsten Umsatz?

In Stage 1 werden mit $match alle gültigen Verkäufe gefiltert. Also nur Bestellungen, die als bezahlt, versendet oder abgeschlossen markiert sind:



In Stage 2 werden mit $addFields Datentypen vereinheitlicht (Decimal). Rabatte werden berücksichtigt. Sollte kein Rabatt vorhanden sein, wird als Standardwert 0 gesetzt. Ausserdem wird hier ein Währungsfaktor (fx) berechnet, um alle Beträge in Schweizer Franken (CHF) umzurechnen:

Stage 2 $addFields

```
1  {
2    qtyN: {
3      $convert: {
4        input: "$qty",
5        to: "decimal",
6        onError: 0,
7        onNull: 0
8      }
9    },
10   unitPriceN: {
11     $convert: {
12       input: "$unit_price",
13       to: "decimal",
14       onError: 0,
15       onNull: 0
16     }
17   },
18   discountN: {
19     $convert: {
20       input: {
21         $ifNull: ["$discount_amount", 0]
22       },
23       to: "decimal",
24       onError: 0
25     }
26   },
27   fx: {
28     $switch: {
29       branches: [
30         {
31           case: {
32             $eq: ["$currency", "CHF"]
33           },
34           then: {
35             $toDecimal: "1.00"
36           }
37         },
38         {
39           case: {
40             $eq: ["$currency", "EUR"]
41           },
42           then: {
43             $toDecimal: "0.95"
44           }
45         }
46       ],
47       default: {
48         $toDecimal: "1.00"
49       }
50     }
51   }
52 }
```

Output preview after $addFields stage (Sample of 10 docun

```
_id: ObjectId('6900b5260c87750b789997b8')
order_id : 1001
order_ts_iso : 2025-06-01T10:00:00.000+00:00
status : "PAID"
currency : "CHF"
customer_id : 1
customer_email : "a@demo.ch"
customer_first_name : "Anna"
customer_last_name : "Muster"
customer_country : "CH"
line_no : 1
product_id : 102
qty : 1
unit_price : 21.5
discount_amount : 0
line_total : 21.5
item_number : "SERUM-VITC-30ML"
product_name : "Vitamin C Serum 30ml"
product_brand : "GlowUp"
category_id : 1
category_name : "Seren"
qtyN : 1
unitPriceN : 21.5000000000000
discountN : 0
fx : 1.00
```

In Stage 3 wird gruppiert nach Seren, Cremes und Masken (Aggregation je Produktkategorie).

- Revenue = Summe aus Einkäufe (Menge * Preis – Rabatt) je Kategorie
- Umrechnung in CHF
- Items_sold = Gesamtanzahl verkaufter Einheiten je Kategorie

Stage 3 $group

```
{
  _id: "$category_name",
  revenueSum: {
    $sum: {
      $multiply: [
        {
          $subtract: [
            {
              $multiply: ["$qtyN", "$unitPriceN"]
            },
            "$discountN"
          ]
        },
        "$fx"
      ]
    }
  },
  items_sold: {
    $sum: "$qtyN"
  }
}
```

Output preview after $group stage (Sample of 3 documents)

```
_id: "Masken"
revenueSum : 300.5990000000000000
items_sold : 20
```

In Stage 4 wird das Endergebnis formatiert, indem der Umsatz auf zwei Nachkommastellen gerundet wird:

Stage 4 $project

```
{
  _id: 1,
  revenue: {
    $round: ["$revenueSum", 2]
  },
  items_sold: 1
}
```

Output preview after $project stage (Sample of 3 documents)

```
_id: "Masken"
items_sold : 20
revenue : 300.60
```

Am Schluss werden die Kategorien mit $sort absteigend nach revenue sortiert, damit die umsatzstärksten Kategorien oben stehen:

Stage 5 $sort

```
1 ▾ {
2     revenue: -1
3   }
```

Output preview after $sort stage (Sample of 3 documents)

```
_id: "Cremes"
items_sold : 26
revenue : 681.42
```

Somit ergibt sich die Lösung zu Frage 1: Die Produktkategorie Cremes generiert den höchsten Umsatz mit rund 681.4 CHF. Das Ergebnis entspricht erwartungsgemäss der SQL-Abfrage.

PIPELINE OUTPUT PREVIEW

Sample of 3 documents

OUTPUT OPTIONS

```
_id: "Cremes"
items_sold : 26
revenue : 681.42
```

```
_id: "Seren"
items_sold : 18
revenue : 380.52
```

```
_id: "Masken"
items_sold : 20
revenue : 300.60
```

3.1.2. Wie viel Umsatz pro Kunde (Customer Lifetime Value) wurde über alle Bestellungen hinweg erzielt?

Gleich wie beim ersten Query werden in Stage 1 mit $match alle gültigen Verkäufe gefiltert. Also nur Bestellungen, die als bezahlt, versendet oder abgeschlossen markiert sind, werden berücksichtigt:

Stage 1 $match

```
1 {
2   status: {
3     $in: ["PAID", "SHIPPED", "COMPLETED"]
4   }
5 }
```

Output preview after $match stage (Sample of 10 documents)

```
_id: ObjectId('6900b5260c87750b789997b8')
order_id : 1001
order_ts_iso : 2025-06-01T10:00:00.000+00:00
status : "PAID"
currency : "CHF"
customer_id : 1
customer_email : "a@demo.ch"
customer_first_name : "Anna"
customer_last_name : "Muster"
```

Mit $addFields in Stage 2 werden Datentypen vereinheitlicht. Mengen (qty) und Preise (unit_price) werden in Dezimalzahlen (Double) konventiert. Rabatte werden berücksichtigt. Sollte kein Rabatt vorhanden sein, wird als Standardwert 0 gesetzt. Das Bestelldatum wird in ein echtes Datum umgewandelt.

Stage 2 $addFields

```
1 {
2   qtyN: {
3     $toDouble: "$qty"
4   },
5   priceN: {
6     $toDouble: "$unit_price"
7   },
8   discountN: {
9     $toDouble: {
10      $ifNull: ["$discount_amount", 0]
11    }
12  },
13  orderDate: {
14    $toDate: "$order_ts_iso"
15  }
16 }
```

Output preview after $addFields stage (Sample of 10 documents)

```
_id: ObjectId('6900b5260c87750b789997b8')
order_id : 1001
order_ts_iso : 2025-06-01T10:00:00.000+00:00
status : "PAID"
currency : "CHF"
customer_id : 1
customer_email : "a@demo.ch"
customer_first_name : "Anna"
customer_last_name : "Muster"
customer_country : "CH"
line_no : 1
```

In Stage 3 werden für jede Kombination aus Kunde und Währung Kennzahlen berechnet:

- Revenue = Summe aller Einkäufe (Menge * Preis – Rabatt)
- Units: Gesamtmenge der gekauften Produkte
- Orders: Liste aller Bestell-IDs des Kunden
- First & last purchase: liefert Bestellverlauf des Kunden

Stage 3 $group

```
1  {
2    _id: {
3      customer_id: "$customer_id",
4      currency: "$currency"
5    },
6    revenue: {
7      $sum: {
8        $subtract: [
9          {
10           $multiply: ["$qtyN", "$priceN"]
11         },
12         "$discountN"
13       ]
14     }
15   },
16   units: {
17     $sum: "$qtyN"
18   },
19   orders: {
20     $addToSet: "$order_id"
21   },
22   first_purchase: {
23     $min: "$orderDate"
24   },
25   last_purchase: {
26     $max: "$orderDate"
27   }
28 }
```

Output preview after $group stage (Sample of 10 documents)

```
▸ _id: Object
  revenue : 131.51999999999998
  units : 5
▸ orders : Array (5)
  first_purchase : 2025-06-01T10:30:00.000+00:…
  last_purchase : 2025-06-28T10:00:00.000+00:00
```

Die Anzahl der Bestellungen wird in Stage 4 als neues Feld order_count aus der Länge des orders-Arrays berechnet.

Stage 4 $addFields

```
1  {
2    order_count: {
3      $size: "$orders"
4    }
5  }
```

Output preview after $addFields stage (Sample of 10 documents)

```
▸ _id: Object
  revenue : 207.2
  units : 8
▸ orders : Array (4)
  first_purchase : 2025-06-01T10:25:00.000+00:…
  last_purchase : 2025-06-24T10:00:00.000+00:00
  order_count : 4
```

In Stage 5 werden über customer_id zusätzliche Daten aus der Customer-Collection geholt (z. B. Name, E-Mail, Land, Geburtsjahr).

Stage 5 $lookup

```
1 {
2   from: "customer",
3   localField: "_id.customer_id",
4   foreignField: "customer_id",
5   as: "cust"
6 }
```

Output preview after $lookup stage (Sample of 10 documents)

```
▸ _id: Object
  revenue : 15.9
  units : 1
▸ orders : Array (1)
  first_purchase : 2025-06-01T10:15:00.000+00…
  last_purchase : 2025-06-01T10:15:00.000+00:…
  order_count : 1
▸ cust : Array (1)
```

$unwind wandelt das Ergebnisarray cust in ein einzelnes eingebettetes Dokument um, um direkt auf Felder zugreifen zu können.

Stage 6 $unwind

```
1 {
2   path: "$cust",
3   preserveNullAndEmptyArrays: true
4 }
```

Output preview after $unwind stage (Sample of 10 documents)

```
  order_count : 4
▾ cust : Object
    _id : ObjectId('6900b5450c87750b78999804')
    customer_id : 3
    email : "c@demo.ch"
    first_name : "Cara"
    last_name : "Case"
    created_at_iso : 2025-02-10T00:00:00.000+…
    country : "CH"
    birth_year : 1992
```

Stage 7 sorgt dafür, dass die finale Ausgabe nur relevante Felder enthält:

- Kunden-ID, Währung, Umsatz, Anzahl Bestellungen, verkaufte Einheiten
- Durchschnittlicher Bestellwert (avg_order_value) = Umsatz ÷ Anzahl Bestellungen
- Kundendaten (Name, E-Mail, Land etc.)

Stage 7 $project

```
1  {
2    _id: 0,
3    customer_id: "$_id.customer_id",
4    currency: "$_id.currency",
5    revenue: {
6      $round: ["$revenue", 2]
7    },
8    units: 1,
9    order_count: 1,
10   avg_order_value: {
11     $round: [
12       {
13         $cond: [
14           {
15             $gt: ["$order_count", 0]
16           },
17           {
18             $divide: ["$revenue", "$order_
19           },
20           0
21         ]
22       },
23       2
24     ]
25   },
26   first_purchase: 1,
27   last_purchase: 1,
28   email: "$cust.email",
29   first_name: "$cust.first_name",
30   last_name: "$cust.last_name",
31   country: "$cust.country",
32   birth_year: "$cust.birth_year"
33 }
```

Output preview after $project stage (Sample of 10 documents)

```
units : 1
first_purchase : 2025-06-14T10:00:00.000+00:…
last_purchase : 2025-06-14T10:00:00.000+00:00
order_count : 1
customer_id : 2
currency : "EUR"
revenue : 22.9
avg_order_value : 22.9
email : "b@demo.ch"
first_name : "Ben"
last_name : "Beispiel"
country : "DE"
birth_year : 1995
```

Am Ende werden die Kunden absteigend nach Gesamtumsatz sortiert, sodass die umsatzstärksten Kunden zuerst angezeigt werden.

Stage 8 $sort

```
1 {
2   revenue: -1
3 }
```

Output preview after $sort stage (Sample of 10 documents)

```
units : 15
first_purchase : 2025-06-01T10:20:00.000+00…
last_purchase : 2025-06-23T10:00:00.000+00:…
order_count : 5
customer_id : 5
currency : "CHF"
revenue : 336.5
avg_order_value : 67.3
email : "e@demo.fr"
```

Somit ergibt sich die Lösung zu Frage 2: Emma hat den grössten CLV mit 336.50 CHF, gefolgt von Jonas mit 228.26 CHF und Fabio mit 207.20 CHF. Dies entspricht den Resultaten auf mySQL.

PIPELINE OUTPUT PREVIEW

Sample of 10 documents

OUTPUT OPTIONS


```
units : 15
first_purchase : 2025-06-01T10:20:00.000+00:00
last_purchase : 2025-06-23T10:00:00.000+00:00
order_count : 5
customer_id : 5
currency : "CHF"
revenue : 336.5
avg_order_value : 67.3
email : "e@demo.fr"
first_name : "Emma"
last_name : "Durand"
country : "FR"
birth_year : 1982
```

```
units : 12
first_purchase : 2025-06-30T10:00:00.000+00:00
last_purchase : 2025-07-03T10:00:00.000+00:00
order_count : 4
customer_id : 10
currency : "CHF"
revenue : 228.26
avg_order_value : 57.06
email : "j@demo.ch"
first_name : "Jonas"
last_name : "Keller"
country : "CH"
birth_year : 1952
```

```
units : 8
first_purchase : 2025-06-01T10:25:00.000+00:00
last_purchase : 2025-06-24T10:00:00.000+00:00
order_count : 4
customer_id : 6
currency : "CHF"
revenue : 207.2
avg_order_value : 51.8
email : "f@demo.it"
first_name : "Fabio"
last_name : "Rossi"
country : "IT"
birth_year : 1975
```

```
units : 5
first_purchase : 2025-06-01T10:30:00.000+00:00
last_purchase : 2025-06-28T10:00:00.000+00:00
order_count : 5
customer_id : 7
currency : "CHF"
revenue : 131.52
avg_order_value : 26.3
email : "g@demo.es"
first_name : "Gloria"
last_name : "Martinez"
country : "ES"
birth_year : 1968
```

```
units : 8
first_purchase : 2025-06-01T10:10:00.000+00:00
last_purchase : 2025-06-18T10:00:00.000+00:00
order_count : 4
customer_id : 3
currency : "CHF"
revenue : 123.2
avg_order_value : 30.8
email : "c@demo.ch"
first_name : "Cara"
last_name : "Case"
country : "CH"
birth_year : 1992
```

```
units : 6
first_purchase : 2025-06-01T10:40:00.000+00:00
last_purchase : 2025-06-30T10:00:00.000+00:00
order_count : 3
customer_id : 9
currency : "CHF"
revenue : 121.22
avg_order_value : 40.41
email : "i@demo.us"
first_name : "Isabella"
last_name : "Johnson"
country : "CH"
birth_year : 2003
```

```
units : 3
first_purchase : 2025-06-01T10:35:00.000+00:00
last_purchase : 2025-06-27T10:00:00.000+00:00
order_count : 3
customer_id : 8
currency : "CHF"
revenue : 83.7
avg_order_value : 27.9
email : "a@demo.it"
first_name : "Alberto"
last_name : "Rossi"
country : "IT"
birth_year : 1960
```

```
units : 2
first_purchase : 2025-06-01T10:00:00.000+00:00
last_purchase : 2025-06-12T10:00:00.000+00:00
order_count : 2
customer_id : 1
currency : "CHF"
revenue : 43
avg_order_value : 21.5
email : "a@demo.ch"
first_name : "Anna"
last_name : "Muster"
country : "CH"
birth_year : 1998
```

```
units : 2
first_purchase : 2025-06-18T10:00:00.000+00:00
last_purchase : 2025-06-19T10:00:00.000+00:00
order_count : 2
customer_id : 4
currency : "EUR"
revenue : 28.82
avg_order_value : 14.41
email : "d@demo.at"
first_name : "David"
last_name : "Huber"
country : "AT"
birth_year : 1988
```

```
units : 1
first_purchase : 2025-06-01T10:05:00.000+00:00
last_purchase : 2025-06-01T10:05:00.000+00:00
order_count : 1
customer_id : 2
currency : "CHF"
revenue : 22.9
avg_order_value : 22.9
email : "b@demo.ch"
first_name : "Ben"
last_name : "Beispiel"
country : "DE"
birth_year : 1995
```

4 Event Generation / Event Processing

Event types

Wir haben in unserem Projekt mehrere Event typen überlegt und unter anderem den PurchaseEvent erstellt. Mit diesem Event war es uns möglich, beide analytischen Fragen zu beantworten.

Event-Typ	Wann er auftritt	Beispiel-Key	Beispiel-Value	Analytische Relevanz
PurchaseEvent	Wenn ein Kunde ein Produkt kauft	customer_12	category=cremes;price=199.99;age=35	Grundlage für Umsatzanalyse nach Kategorie & Kundensegment
(optional) **CartEvent**	Wenn ein Produkt in den Warenkorb gelegt wird	customer_5	category=seren;price=39.99;age=25	Zeigt Kaufinteresse & Kundenverhalten vor dem Kauf
(optional) **ViewEvent**	Wenn ein Kunde ein Produkt ansieht	customer_9	category=masken;age=30	Unterstützt Marketinganalysen zum Interesse an Produktgruppen

Event Generator

kontinuierlich auf dem Kafka-Topic group2__orders. Für jedes Event wird ein Kunde (customer_ID) als Key verwendet, während die Value-Information die Produktkategorie, den Kaufpreis und die Altersgruppe des Kunden enthält.

Die Werte werden zufällig erzeugt, sodass unterschiedliche Kunden, Produktkategorien, Preise und Altersgruppen abgebildet werden und eine realistische, vielfältige Datenbasis entsteht. Durch die Endlosschleife mit einem festen Zeitintervall von zwei Sekunden wird ein kontinuierlicher Datenstrom erzeugt, der die Grundlage für die Echtzeit-Analyse im Event Processor bildet. Diese Struktur stellt sicher, dass alle relevanten Informationen für die Beantwortung der analytischen Fragen (Wie im vorherigen Kapitel definiert) vorhanden sind: Kategorie und Preis für die Umsatzanalyse sowie Altersgruppe (Für mögliche zukünftige Analysefragen) und Kategorie für die Analyse

des Kaufverhaltens verschiedener Kundensegmente.

```java
package event_generator;

import java.time.Duration;
import java.util.Properties;
import java.util.Random;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;

public class EventGenerator {

    public static void main(String[] args) {

        /// Name of our group:
        String group = "group2";

        // Kafka topic
        String topic = group + "__orders";

        // connect to Kafka
        Properties props = new Properties();
        props.put("bootstrap.servers", "192.168.111.10:9092");
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        Producer<String, String> producer = new KafkaProducer<>(props);

        Random rand = new Random();

        // Example data
        String[] categories = { "seren", "masken", "cremes", "peelings", "makeup" };
        int[] ageGroups = { 18, 25, 35, 45, 60 };

        for (;;) { // endless loop
            String customer = "customer_" + (rand.nextInt(20) + 1);
            String category = categories[rand.nextInt(categories.length)];
            double price = Math.round((rand.nextDouble() * 300 + 10) * 100.0) / 100.0;
            int age = ageGroups[rand.nextInt(ageGroups.length)];

            // Event value (simple text format)
            String value = "category=" + category + ";price=" + price + ";age=" + age;

            ProducerRecord<String, String> record = new ProducerRecord<>(topic, customer, value);

            producer.send(record, (RecordMetadata metadata, Exception exception) -> {
                if (exception != null) {
                    exception.printStackTrace();
                } else {
                    System.out.printf("Sent event(key=%s value=%s)%n", customer, value);
                }
            });

            try {
                Thread.sleep(Duration.ofSeconds(2));
            } catch (InterruptedException e) {}
        }
    }
}
```

EventProcessorBasic.java

Der Event Prozessor liest die vom Generator erzeugten Events in Echtzeit aus dem Kafka-Topic group2__orders und verarbeitet sie mit Kafka Streams. Zunächst werden die Events gefiltert, sodass nur Käufe mit einem Preis über 100.- weiterverarbeitet werden, um sich auf besonders relevante High-Value-Käufe zu konzentrieren und gleichzeitig die Umsetzung eines Filters zu üben. Die gefilterten Events werden sowohl auf der Konsole ausgegeben als auch nach Produktkategorie

aggregiert, wobei die Gesamtsumme der Umsätze pro Kategorie berechnet wird. Dadurch wird die Datenmenge reduziert, der Fokus auf umsatzstarke Kategorien gelegt und die Grundlage für die Beantwortung der ersten analytische Frage, «Welche Produktkategorien generieren den höchsten Umsatz?», geschaffen.

Die Architektur trennt klar Datenproduktion und -verarbeitung, simuliert ein realistisches Streaming-Szenario und ermöglicht Erweiterungen für weiterführende Aggregationen oder eine spätere Integration einer Datenbank.

EventProcessorBasic.java

```java
package event_processor;

import java.util.Properties;
import java.util.concurrent.CountDownLatch;


import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.Topology;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.KTable;
import org.apache.kafka.streams.kstream.Materialized;
import org.apache.kafka.streams.processor.WallclockTimestampExtractor;

public class EventProcessorBasic {

    public static void main(String[] args) {
        String group = "group2"; //// Our Group Name

        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "stream_processor-" + group);
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "192.168.111.10:9092");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG, WallclockTimestampExtractor.class);
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest");

        System.out.println("*** Waiting for purchase events...");
```

```java
31
32        final StreamsBuilder builder = new StreamsBuilder();
33
34        // Listen to the purchase topic
35        KStream<String, String> source = builder.stream(group + "__orders");
36
37        // Filter High-Value-Purchases (>100€) und Ausgabe
38        KStream<String, String> highValuePurchases = source.filter((key, value) -> {
39            try {
40                String[] parts = value.split(";");
41                double price = Double.parseDouble(parts[1].split("=")[1]);
42                return price > 100;
43            } catch (Exception e) {
44                return false;
45            }
46        });
47
48        highValuePurchases.foreach((key, value) -> {
49            System.out.println("High-value purchase -> Key: " + key + " | Value: " + value);
50        });
51
52        // Umsatz pro Kategorie aggregieren
53        KTable<String, Double> revenuePerCategory = highValuePurchases
54            .mapValues(value -> {
55                String[] parts = value.split(";");
56                String category = parts[0].split("=")[1];
57                double price = Double.parseDouble(parts[1].split("=")[1]);
58                return new CategoryPrice(category, price);
59            })
```

```java
            .groupBy((key, catPrice) -> catPrice.category)
            .aggregate(
                () -> 0.0,
                (aggKey, newValue, aggValue) -> aggValue + newValue.price,
                Materialized.with(Serdes.String(), Serdes.Double())
            );

        // Ausgabe der Aggregation
        revenuePerCategory.toStream().foreach((category, total) -> {
            System.out.println("Kategorie: " + category + " | Umsatz: " + total);
        });

        final Topology topology = builder.build();
        final KafkaStreams streams = new KafkaStreams(topology, props);
        final CountDownLatch latch = new CountDownLatch(1);

        Runtime.getRuntime().addShutdownHook(new Thread("streams-shutdown-hook") {
            @Override
            public void run() {
                streams.close();
                latch.countDown();
            }
        });

        try {
            streams.start();
            latch.await();
        } catch (Throwable e) {
            System.exit(1);

        }
        System.exit(0);
    }

    // Hilfsklasse für Kategorie + Preis
    public static class CategoryPrice {
        public String category;
        public double price;

        public CategoryPrice(String category, double price) {
            this.category = category;
            this.price = price;
        }
    }
}
```

Console output

```
Console
EventProcessorBasic [Java Application] /Users/vivien/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.aarch64_21.0.8.v20250724-1412/jre/bi
*** Waiting for purchase events...
WARN  StateDirectory:138 - Using an OS temp directory in the state.dir property can cause failures with writing the
High-value purchase -> Key: customer_9 | Value: category=peelings;price=221.32;age=35
High-value purchase -> Key: customer_12 | Value: category=masken;price=181.04;age=35
High-value purchase -> Key: customer_14 | Value: category=seren;price=200.34;age=18
High-value purchase -> Key: customer_15 | Value: category=makeup;price=254.51;age=18
High-value purchase -> Key: customer_17 | Value: category=makeup;price=175.96;age=18
High-value purchase -> Key: customer_10 | Value: category=masken;price=192.27;age=18
High-value purchase -> Key: customer_6 | Value: category=makeup;price=215.38;age=18
High-value purchase -> Key: customer_6 | Value: category=makeup;price=131.45;age=25
High-value purchase -> Key: customer_5 | Value: category=masken;price=289.7;age=18
High-value purchase -> Key: customer_15 | Value: category=masken;price=150.4;age=45
High-value purchase -> Key: customer_3 | Value: category=makeup;price=200.42;age=45
High-value purchase -> Key: customer_3 | Value: category=cremes;price=125.24;age=35
High-value purchase -> Key: customer_8 | Value: category=masken;price=185.51;age=60
High-value purchase -> Key: customer_12 | Value: category=makeup;price=284.57;age=18
High-value purchase -> Key: customer_7 | Value: category=makeup;price=239.87;age=45
High-value purchase -> Key: customer_1 | Value: category=masken;price=303.59;age=18
High-value purchase -> Key: customer_14 | Value: category=seren;price=140.86;age=18
High-value purchase -> Key: customer_8 | Value: category=makeup;price=231.82;age=35
High-value purchase -> Key: customer_19 | Value: category=seren;price=241.48;age=25
High-value purchase -> Key: customer_8 | Value: category=seren;price=216.45;age=18
High-value purchase -> Key: customer_6 | Value: category=makeup;price=149.3;age=60
High-value purchase -> Key: customer_12 | Value: category=seren;price=120.95;age=25
High-value purchase -> Key: customer_2 | Value: category=makeup;price=158.95;age=45
High-value purchase -> Key: customer_7 | Value: category=masken;price=188.01;age=25
High-value purchase -> Key: customer_9 | Value: category=makeup;price=252.35;age=18
High-value purchase -> Key: customer_18 | Value: category=cremes;price=268.04;age=45
```

```
Console
<terminated> EventProcessorBasic [Java Application] /Users/vivien/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.aarch64_21.0.8.v202507
High-value purchase -> Key: customer_15 | Value: category=masken;price=204.44;age=35
High-value purchase -> Key: customer_1 | Value: category=masken;price=280.59;age=45
High-value purchase -> Key: customer_17 | Value: category=seren;price=213.87;age=45
High-value purchase -> Key: customer_19 | Value: category=masken;price=224.4;age=35
High-value purchase -> Key: customer_11 | Value: category=masken;price=236.53;age=35
High-value purchase -> Key: customer_16 | Value: category=seren;price=104.24;age=45
High-value purchase -> Key: customer_7 | Value: category=cremes;price=251.09;age=35
High-value purchase -> Key: customer_17 | Value: category=cremes;price=262.36;age=60
High-value purchase -> Key: customer_14 | Value: category=seren;price=277.9;age=25
High-value purchase -> Key: customer_14 | Value: category=makeup;price=108.77;age=60
High-value purchase -> Key: customer_3 | Value: category=seren;price=225.96;age=18
High-value purchase -> Key: customer_4 | Value: category=peelings;price=196.48;age=25
High-value purchase -> Key: customer_7 | Value: category=peelings;price=294.36;age=18
High-value purchase -> Key: customer_3 | Value: category=peelings;price=117.42;age=18
High-value purchase -> Key: customer_4 | Value: category=seren;price=113.43;age=35
High-value purchase -> Key: customer_7 | Value: category=seren;price=147.42;age=35
High-value purchase -> Key: customer_7 | Value: category=cremes;price=220.86;age=18
High-value purchase -> Key: customer_17 | Value: category=cremes;price=152.9;age=60
High-value purchase -> Key: customer_15 | Value: category=makeup;price=261.67;age=25
High-value purchase -> Key: customer_16 | Value: category=masken;price=236.52;age=45
High-value purchase -> Key: customer_18 | Value: category=seren;price=110.81;age=60
High-value purchase -> Key: customer_2 | Value: category=masken;price=150.42;age=35
Kategorie: peelings | Umsatz: 20886.38999999999
Kategorie: cremes | Umsatz: 21594.090000000022
Kategorie: makeup | Umsatz: 25720.15
Kategorie: seren | Umsatz: 23467.840000000007
Kategorie: masken | Umsatz: 24505.69
```

EventProcessorCustomerRevenue.java

Der *EventProcessorCustomerRevenue* liest die vom Generator erzeugten Kaufereignisse in Echtzeit aus dem Kafka-Topic group2__orders. Zunächst werden die eingehenden Events auch hier gefiltert, sodass nur Käufe mit einem Preis über 100.- berücksichtigt werden. Anschliessend wird für jeden Kunden der gesamte Umsatz berechnet, indem alle zugehörigen Preise im Stream mithilfe von

groupByKey() und einer Aggregation summiert werden. Das Ergebnis ist eine fortlaufend aktualisierte Tabelle (*KTable*), die für jeden Kunden den aktuellen Gesamtumsatz anzeigt.

Diese Verarbeitung beantwortet direkt die zweite analytische Frage, nämlich **«Wie viel Umsatz pro Kunde wurde über alle Bestellungen hinweg erzielt?»**. Der Code demonstriert dabei das Prinzip einer Echtzeitaggregation mit Kafka Streams, bei der sich der Umsatzwert automatisch aktualisiert, sobald neue Events eintreffen. Dadurch entsteht ein dynamisches, kontinuierlich berechnetes Ergebnis, das für Dashboards oder weitere Analysen genutzt werden könnte.

```
1  package event_processor;
2
3  import java.util.Properties;
4  import java.util.concurrent.CountDownLatch;
5
6  import org.apache.kafka.clients.consumer.ConsumerConfig;
7  import org.apache.kafka.common.serialization.Serdes;
8  import org.apache.kafka.streams.KafkaStreams;
9  import org.apache.kafka.streams.StreamsBuilder;
10 import org.apache.kafka.streams.StreamsConfig;
11 import org.apache.kafka.streams.Topology;
12 import org.apache.kafka.streams.kstream.KStream;
13 import org.apache.kafka.streams.kstream.KTable;
14 import org.apache.kafka.streams.kstream.Materialized;
15 import org.apache.kafka.streams.processor.WallclockTimestampExtractor;
16
```

```
17 public class EventProcessorCustomerRevenue {
18
19     public static void main(String[] args) {
20         String group = "group2";
21
22         Properties props = new Properties();
23         props.put(StreamsConfig.APPLICATION_ID_CONFIG, "customer_revenue_processor-" + group);
24         props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "192.168.111.10:9092");
25         props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
26         props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());
27         props.put(StreamsConfig.DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG, WallclockTimestampExtractor.class);
28         props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest");
29
30         System.out.println("*** Berechne Umsatz pro Kunde...");
31
32         final StreamsBuilder builder = new StreamsBuilder();
33
34         // Stream von allen Bestellungen
35         KStream<String, String> source = builder.stream(group + "__orders");
36
37         // Optional: nur High-Value-Purchases berücksichtigen
38         KStream<String, String> highValuePurchases = source.filter((key, value) -> {
39             try {
40                 double price = Double.parseDouble(value.split(";")[1].split("=")[1]);
41                 return price > 100; // Filterung optional wie vorhher
42             } catch (Exception e) {
43                 return false;
```

```java
            }
        });

        // Umsatz pro Kunde aggregieren
        KTable<String, Double> revenuePerCustomer = highValuePurchases
            .mapValues(value -> Double.parseDouble(value.split(";")[1].split("=")[1]))
            .groupByKey() // Key = Kunde
            .aggregate(
                () -> 0.0,
                (customer, newPrice, aggValue) -> aggValue + newPrice,
                Materialized.with(Serdes.String(), Serdes.Double())
            );

        // Ausgabe der Ergebnisse
        revenuePerCustomer.toStream().foreach((customer, total) -> {
            System.out.println("Kunde: " + customer + " | Gesamtumsatz: " + total);
        });

        final Topology topology = builder.build();
        final KafkaStreams streams = new KafkaStreams(topology, props);
        final CountDownLatch latch = new CountDownLatch(1);

        // Shutdown-Hook
        Runtime.getRuntime().addShutdownHook(new Thread("streams-shutdown-hook") {
            @Override
            public void run() {
                streams.close();
                latch.countDown();
            }
        });

        try {
            streams.start();
            latch.await();
        } catch (Throwable e) {
            System.exit(1);
        }
        System.exit(0);
    }
}
```

Console Output

```
Console
EventProcessorCustomerRevenue [Java Application] /Users/vivien/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.aarch64_21.0.8.v20250724
*** Berechne Umsatz pro Kunde...
WARN  StateDirectory:138 - Using an OS temp directory in the state.dir property can cause failures with writing the
Kunde: customer_13 | Gesamtumsatz: 3533.1500000000005
Kunde: customer_18 | Gesamtumsatz: 4510.650000000001
Kunde: customer_19 | Gesamtumsatz: 4265.12
Kunde: customer_11 | Gesamtumsatz: 3529.7400000000002
Kunde: customer_3 | Gesamtumsatz: 5946.0999999999985
Kunde: customer_6 | Gesamtumsatz: 5021.570000000001
Kunde: customer_20 | Gesamtumsatz: 4582.64
Kunde: customer_15 | Gesamtumsatz: 4124.03
Kunde: customer_17 | Gesamtumsatz: 3980.6800000000003
Kunde: customer_2 | Gesamtumsatz: 3857.4499999999994
Kunde: customer_7 | Gesamtumsatz: 4633.34
Kunde: customer_9 | Gesamtumsatz: 4857.110000000001
Kunde: customer_14 | Gesamtumsatz: 4503.11
Kunde: customer_16 | Gesamtumsatz: 5118.550000000001
Kunde: customer_12 | Gesamtumsatz: 4226.639999999999
Kunde: customer_10 | Gesamtumsatz: 3954.05
Kunde: customer_1 | Gesamtumsatz: 3525.91
Kunde: customer_8 | Gesamtumsatz: 4720.249999999999
Kunde: customer_17 | Gesamtumsatz: 4122.990000000001
Kunde: customer_13 | Gesamtumsatz: 3976.6700000000005
Kunde: customer_6 | Gesamtumsatz: 5219.01
Kunde: customer_14 | Gesamtumsatz: 4623.66
Kunde: customer_9 | Gesamtumsatz: 5067.31
Kunde: customer_4 | Gesamtumsatz: 3997.4300000000003
```

5 Resultate und Diskussion

5.1 Analytische Frage 1

Welche Produktkategorien generieren den höchsten Umsatz?

Verwendeter Prozessor**:** EventProcessorBasic

Erwartete Resultate:

Der Processor filtert alle Kauf-Events und berücksichtigt nur solche mit einem Preis über 100 CHF. Auf der Konsole erscheinen regelmässig Meldungen wie:

High-value purchase -> Key: customer_7 | Value: category=cremes;price=149.99;age=35
Kategorie: cremes | Gesamtumsatz: 820.45
Kategorie: masken | Gesamtumsatz: 640.20
Kategorie: makeup | Gesamtumsatz: 550.30

Diese Ausgaben zeigen, welche Produktkategorien besonders häufig und mit hohem Wert gekauft werden.

Beantwortung der analytischen Frage:

Aus der Aggregation der Umsätze pro Produktkategorie lässt sich erkennen, **welche Kategorien den grössten Umsatzbeitrag liefern**. In unserem Fall könnte sich z. B. zeigen, dass „Cremes“ oder „Masken“ die umsatzstärksten Produktkategorien sind. Diese Erkenntnis bildet die Grundlage für **Sortimentsplanung und Produktstrategie**, da sich daraus ableiten lässt, in welche Kategorien mehr investiert oder welche gezielt beworben werden sollten.

5.2 Analytische Frage 2

Wie viel Umsatz pro Kunde (Customer Lifetime Value) wurde über alle Bestellungen hinweg erzielt?

Verwendeter Prozessor: EventProcessorCustomerRevenue

Erwartete Resultate:

Der Prozessor liest alle Kauf-Events und summiert den Umsatz pro Kunde. Auf der Konsole erscheinen fortlaufend aktualisierte Zeilen wie:

Kunde:	customer_3	|	Gesamtumsatz:	520.75
Kunde:	customer_7	|	Gesamtumsatz:	310.50
Kunde:	customer_12	|	Gesamtumsatz:	955.30

Jedes Mal, wenn ein neuer Kauf für denselben Kunden eintrifft, wird der Gesamtwert automatisch aktualisiert.

Beantwortung der analytischen Frage:

Diese Aggregation beantwortet direkt die zweite analytische Frage, indem sie zeigt, **welche Kunden über alle Bestellungen hinweg den höchsten Gesamtumsatz erzielen**. Damit lässt sich der sogenannte **Customer Lifetime Value (CLV)** bestimmen. Unternehmen können diese Information nutzen, um **wertvolle Kunden gezielt zu binden**, etwa durch personalisierte Angebote oder Treueprogramme.

5.3 Resultate gesamt

In unserem Beispiel sind die Resultate gleichmässig über alle Produktkategorien und Kunden verteilt, da die Ereignisse zufällig generiert werden. Jede Produktkategorie und Altersgruppe wird mit derselben Wahrscheinlichkeit ausgewählt, und auch die Kaufpreise liegen zufällig zwischen 10 und 300 Franken. Dadurch entstehen keine echten Umsatzschwerpunkte oder dominante Kundensegmente, wie sie in realen Geschäftsdaten üblich wären. Das Ziel dieser Simulation liegt daher nicht in der inhaltlichen Aussagekraft der Daten, sondern im technischen Verständnis der Datenverarbeitung mit Apache Kafka und Kafka Streams.

Als abschliessende Diskussion stellen wir uns die Frage, wie sich die Ergebnisse verändern würden, wenn die verwendeten Daten nicht zufällig generiert, sondern auf realen Kundentransaktionen basieren würden. Echte Daten könnten deutlich aussagekräftigere Muster im Kaufverhalten offenbaren, beispielsweise saisonale Trends, bevorzugte Produktkategorien oder Unterschiede zwischen Kundensegmenten. Dadurch liessen sich wesentlich tiefere Einblicke in das tatsächliche Umsatzverhalten gewinnen.

Zudem wäre es interessant zu untersuchen, wie sich die Ergebnisse verändern würden, wenn viele jüngere Kundengruppen zwar häufiger, aber mit kleineren Beträgen einkaufen, ohne den Filter von 100 CHF. Dies könnte die analytischen Resultate stark beeinflussen und zeigen, dass nicht nur hohe Einzelumsätze, sondern auch häufige kleinere Käufe entscheidend für den Gesamterfolg sind. Diese Überlegung verdeutlicht, wie wichtig ein gutes Data Management und die Arbeit der Analystinnen und Analysten im Backend sind, um auf Basis von Daten fundierte Entscheidungen zu treffen und gezielt auf Kundenbedürfnisse reagieren zu können.

6 Reflexion

Durch die Bearbeitung des Projekts im Modul Big, Semi- and Unstructured Data konnten wir ein praktisches Verständnis dafür entwickeln, wie Datenströme in Echtzeit mit Apache Kafka erzeugt, verarbeitet und analysiert werden können. Besonders wertvoll war das eigenständige Arbeiten mit Kafka Streams, da wir so ein tieferes Verständnis für Event-getriebene Architekturen und deren Bedeutung in modernen Datenanalyse-Systemen gewonnen haben.

Die Zusammenarbeit im Team verlief sehr gut. Wir haben uns gegenseitig unterstützt und gemeinsam an den Lösungen gearbeitet. Durch den Austausch konnten wir verschiedene Perspektiven einbringen und voneinander lernen, was den Projektfortschritt deutlich gefördert hat.

Ebenfalls spannend war es, die Unterschiede zwischen klassischen relationalen Datenbanken wie MySQL und dokumentenorientierten Systemen wie MongoDB kennenzulernen. Dadurch wurde deutlich, wie unterschiedlich Datenmodelle, Abfragekonzepte und Skalierungsstrategien in strukturierten und unstrukturierten Datenumgebungen umgesetzt werden.

Insgesamt hat uns das Projekt nicht nur technisches Wissen vermittelt, sondern auch gezeigt, wie wichtig gute Kommunikation, Planung und Teamarbeit in der Praxis sind. Es hat uns geholfen, die theoretischen Konzepte aus dem Unterricht in konkrete Anwendungsbeispiele zu übertragen und so ein ganzheitliches Verständnis für den Umgang mit modernen Datensystemen zu entwickeln.

7 Anhang

Code zum Kopieren

Event Generator

```
package event_generator;

import java.time.Duration;
import java.util.Properties;
import java.util.Random;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;

public class EventGenerator {

    public static void main(String[] args) {

        /// Name of our group:
        String group = "group2";

        // Kafka topic
        String topic = group + "__orders";

        // connect to Kafka
        Properties props = new Properties();
        props.put("bootstrap.servers", "192.168.111.10:9092");
        props.put("key.serializer", "org.apache.kafka.common.seri-
alization.StringSerializer");
        props.put("value.serializer", "org.apache.kafka.common.se-
rialization.StringSerializer");

        Producer<String, String> producer = new KafkaPro-
ducer<>(props);

        Random rand = new Random();

        // Example data
        String[] categories = { "seren", "masken", "cremes",
"peelings", "makeup" };
        int[] ageGroups = { 18, 25, 35, 45, 60 };

        for (;;) { // endless loop
            String customer = "customer_" + (rand.nextInt(20) +
1);
```

```
            String category = categories[rand.nextInt(catego-
ries.length)];
            double price = Math.round((rand.nextDouble() * 300 +
10) * 100.0) / 100.0;
            int age = ageGroups[rand.nextInt(ageGroups.length)];

            // Event value (simple text format)
            String value = "category=" + category + ";price=" +
price + ";age=" + age;

            ProducerRecord<String, String> record = new Produc-
erRecord<>(topic, customer, value);

            producer.send(record, (RecordMetadata metadata, Excep-
tion exception) -> {
                if (exception != null) {
                    exception.printStackTrace();
                } else {
                    System.out.printf("Sent event(key=%s
value=%s)%n", customer, value);
                }
            });

            try {
                Thread.sleep(Duration.ofSeconds(2));
            } catch (InterruptedException e) {}
        }
    }
}
```

EventProcessorBasic.java

```
package event_processor;

import java.util.Properties;
import java.util.concurrent.CountDownLatch;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.Topology;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.KTable;
import org.apache.kafka.streams.kstream.Materialized;
```

```
import org.apache.kafka.streams.processor.WallclockTimestampEx-
tractor;

public class EventProcessorBasic {

    public static void main(String[] args) {
        String group = "group2"; //// Our Group Name

        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG,
"stream_processor-" + group);
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
"192.168.111.10:9092");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_TIMESTAMP_EXTRAC-
TOR_CLASS_CONFIG, WallclockTimestampExtractor.class);
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "lat-
est");

        System.out.println("*** Waiting for purchase events...");

        final StreamsBuilder builder = new StreamsBuilder();

        // Listen to the purchase topic
        KStream<String, String> source = builder.stream(group +
"__orders");

        // Filter High-Value-Purchases (>100€) und Ausgabe
        KStream<String, String> highValuePurchases = source.fil-
ter((key, value) -> {
            try {
                String[] parts = value.split(";");
                double price = Double.parseDou-
ble(parts[1].split("=")[1]);
                return price > 100;
            } catch (Exception e) {
                return false;
            }
        });

        highValuePurchases.foreach((key, value) -> {
            System.out.println("High-value purchase -> Key: " +
key + " | Value: " + value);
        });
```

```
    // Umsatz pro Kategorie aggregieren
        KTable<String, Double> revenuePerCategory = highValuePur-
chases
            .mapValues(value -> {
                String[] parts = value.split(";");
                String category = parts[0].split("=")[1];
                double price = Double.parseDou-
ble(parts[1].split("=")[1]);
                return category + ":" + price; // Return String
instead of object
            })
            .groupBy((key, catPriceString) -> {
                String category = catPriceString.split(":")[0];
                return category;
            })
            .aggregate(
                () -> 0.0,
                (aggKey, newValue, aggValue) -> {
                    double price = Double.parseDou-
ble(newValue.split(":")[1]);
                    return aggValue + price;
                },
                Materialized.with(Serdes.String(), Serdes.Dou-
ble())
            );

        // Ausgabe der Aggregation
        revenuePerCategory.toStream().foreach((category, total) ->
{
            System.out.println("Kategorie: " + category + " | Um-
satz: " + total);
        });

        final Topology topology = builder.build();
        final KafkaStreams streams = new KafkaStreams(topology,
props);
        final CountDownLatch latch = new CountDownLatch(1);

        Runtime.getRuntime().addShutdownHook(new Thread("streams-
shutdown-hook") {
            @Override
            public void run() {
                streams.close();
                latch.countDown();
            }
        });

        try {
```

```
            streams.start();
            latch.await();
        } catch (Throwable e) {
            System.exit(1);
        }
        System.exit(0);
    }

    // Hilfsklasse für Kategorie + Preis
    public static class CategoryPrice {
        public String category;
        public double price;

        public CategoryPrice(String category, double price) {
            this.category = category;
            this.price = price;
        }
    }
}
```

EventProcessorCustomerRevenue.java

```
package event_processor;

import java.util.Properties;
import java.util.concurrent.CountDownLatch;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.Topology;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.KTable;
import org.apache.kafka.streams.kstream.Materialized;
import org.apache.kafka.streams.processor.WallclockTimes-
tampExtractor;

public class EventProcessorCustomerRevenue {

    public static void main(String[] args) {
        String group = "group2";

        Properties props = new Properties();
```

```
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "custo-
mer_revenue_processor-" + group);
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
"192.168.111.10:9092");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_TIMESTAMP_EXTRAC-
TOR_CLASS_CONFIG, WallclockTimestampExtractor.class);
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "la-
test");

        System.out.println("*** Berechne Umsatz pro Kunde...");

        final StreamsBuilder builder = new StreamsBuilder();

        // Stream von allen Bestellungen
        KStream<String, String> source = builder.stream(group +
"__orders");

        // Optional: nur High-Value-Purchases berücksichtigen
        KStream<String, String> highValuePurchases = source.fil-
ter((key, value) -> {
            try {
                double price = Double.parseDouble(va-
lue.split(";")[1].split("=")[1]);
                return price > 100; // Filterung optional wie vor-
hher
            } catch (Exception e) {
                return false;
            }
        });

        // Umsatz pro Kunde aggregieren
        KTable<String, Double> revenuePerCustomer = highValue-
Purchases
            .mapValues(value -> Double.parseDouble(va-
lue.split(";")[1].split("=")[1]))
            .groupByKey() // Key = Kunde
            .aggregate(
                () -> 0.0,
                (customer, newPrice, aggValue) -> aggValue + new-
Price,
                Materialized.with(Serdes.String(),
Serdes.Double())
            );
```

```
        // Ausgabe der Ergebnisse
        revenuePerCustomer.toStream().foreach((customer, total) ->
{
            System.out.println("Kunde: " + customer + " | Gesam-
tumsatz: " + total);
        });

        final Topology topology = builder.build();
        final KafkaStreams streams = new KafkaStreams(topology,
props);
        final CountDownLatch latch = new CountDownLatch(1);

        // Shutdown-Hook
        Runtime.getRuntime().addShutdownHook(new Thread("streams-
shutdown-hook") {
            @Override
            public void run() {
                streams.close();
                latch.countDown();
            }
        });

        try {
            streams.start();
            latch.await();
        } catch (Throwable e) {
            System.exit(1);
        }
        System.exit(0);
    }
}
```