Prof. Dr. Bernhard Hollunder
Fakultät Informatik
bernhard.hollunder@hs-furtwangen.de

Hochschule Furtwangen
Robert-Gerwig-Platz 1
D-78120 Furtwangen

Wintersemester 2025/26

Streaming Systems

Praktikum

Bearbeitungszeitraum: 9. Oktober 2025 – 22. Januar 2026

Version vom 25. September 2025

Rahmenbedingungen

In dem Praktikum "Streaming Systems" geht es vor allem um das praktische Arbeiten mit den in der Vorlesung vorgestellten Konzepten, Technologien und Frameworks. Neben der praktischen Umsetzung der gestellten Aufgaben sollen Sie Ihre Lösungen auch in schriftlicher Form dokumentieren. Erstellen Sie hierzu eine Ausarbeitung, in der Sie die gewählten Lösungsansätze und erzielten Ergebnisse aufgabenweise darstellen und bewerten. Sie können sich an folgenden Fragen orientieren:

- Welche Lösungs- und Umsetzungsstrategie wurde gewählt?
- Wie schätzen Sie die Leistungsfähigkeit der Lösung ein? Wurden neben der Basisfunktionalität zusätzliche Funktionen realisiert?
- Welche nicht-funktionalen Anforderungen (z.B. Skalierbarkeit und Durchsatz) wurden untersucht? Mit welchem Ergebnis?
- Wie haben Sie sich von der Korrektheit und Vollständigkeit der Lösung überzeugt? Gibt es eine systematische Teststrategie?
- Wie können die berechneten Ergebnisse auf geeignete Weise dargestellt werden? Gibt es naheliegende Visualisierungsmöglichkeiten?
- Bei welchen Aufgaben haben Sie ChatGPT oder vergleichbare KI-Systeme eingesetzt? Wie hoch ist der Anteil des maschinell generierten Codes und wie schätzen Sie die Qualität ein? Wie aufwändig war es, den generierten Code anzupassen und Fehler zu korrigieren? Bewerten Sie den Produktivitätsgewinn durch Einsatz einer KI.
- Welche zusätzlichen Frameworks und Bibliotheken wurden eingesetzt? Warum?

Beachten Sie, dass einige Aufgaben bewußt nicht vollständig spezifiziert wurden. Seien Sie kreativ und schließen Sie mögliche Lücken auf sinnvolle Weise.

Die Aufgaben sollten in Zweierteams bearbeitet werden.

Die Abschlusspräsentationen aller Lösungen finden gruppenweise nach individueller Absprache ab dem 22. Januar statt. Präsentationen von Zwischenergebnissen erfolgen in den Praktikumsstunden.

Die Gesamtnote für das Praktikum setzt sich je zur Hälfte zusammen aus den lauffähigen Lösungen der Aufgaben sowie der schriftlichen Ausarbeitung. Achten Sie auf "Lesbarkeit" der Dokumentation sowie auf Berücksichtigung der oben formulierten Fragen.

Aufgabenübersicht

Die angegebenen Bearbeitungstermine dienen zur zeitlichen Orientierung.

1. Installation von *Apache ActiveMQ* und *Apache Kafka* sowie *Getting Started*, 9. Oktober
2. JMS Messaging, 16. und 23. Oktober
3. *Apache Kafka* Messaging, 30. Oktober und 6. November
4. *CQRS / Event Sourcing* mit *Apache Kafka* als *Event Store*, 13., 20. und 27. November
5. Datenanalyse "Verkehrsüberwachung" mit Apache Beam, 4. und 11. Dezember
6. *Complex Event Processing* "Verkehrsüberwachung" mit EPL und Esper, 18. Dezember und 8. Januar
7. Weitere Funktionalität zur "Verkehrsüberwachung", 15. und 22. Januar
8. Read-Process-Write-Pattern (optional)
9. Vergleichende Analyse – Lessons Learned (optional)

Aufgabe 1 [9. Oktober]

In den kommenden Praktikumsaufgaben werden u.a. die Message Broker *Apache ActiveMQ* und *Apache Kafka* eingesetzt. In dieser Aufgabe sollen Sie diese Systeme installieren und sich mit der Installation bzw. Konfiguration vertraut machen. Nutzen Sie hierzu einschlägige Beschreibungen[1].

Nach der Installation der Systeme überprüfen Sie die Korrektheit der Installation, in dem Sie jeweils eine einfache Anwendung "zum Laufen bringen". Ein tieferes Verständnis der Systeme ist hierzu nicht erforderlich. Dieses wird in den nachfolgenden Vorlesungs- und Praktikumsstunden aufgebaut.

Aufgabe 2 [16. und 23. Oktober]

In dieser Aufgabe sollen Messwerte eines LiDAR-Sensors verarbeitet werden. Um den Versuchsaufbau zu vereinfachen, wurden die erfassten Messpunkte in einer Datei abgelegt. Zum technischen Hintergrund: Es handelt sich um Daten, die von einem terrestrischen statischen LiDAR-Sensor erzeugt wurden. Ein LiDAR-Scan erzeugt eine Punktwolke im 2D-Raum, die bei horizontaler Drehung um 360 Grad entstanden ist. Ein Punkt gibt den Abstand von dem stationären LiDAR zu einer Begrenzung an, beispielsweise die Entfernung zu einer Wand.

Ein Punkt in einem 2D-LiDAR-Scan wird durch eine Polarkoordinate repräsentiert und wird durch einen Winkel sowie die Entfernung relativ zu dem Pol (d.h. Position des LiDARs) definiert. Des Weiteren gibt ein dritter Wert in einem Datensatz die Qualität der Messung (hohe Unsicherheit bis hohe Genauigkeit) an.

In der Datei befinden sich Einträge mit folgender Struktur:

```
{"winkel": 0.53, "distanz": 1859.50, "qualitaet": 24}
{"winkel": 1.08, "distanz": 1859.50, "qualitaet": 23}
{"winkel": 1.64, "distanz": 1858.00, "qualitaet": 21}
{"winkel": 2.19, "distanz": 1857.00, "qualitaet": 22}
{"winkel": 2.80, "distanz": 1859.00, "qualitaet": 22}
{"winkel": 3.36, "distanz": 1860.50, "qualitaet": 21}
{"winkel": 3.92, "distanz": 1862.00, "qualitaet": 22}
{"winkel": 4.48, "distanz": 1864.00, "qualitaet": 22}
```

[1] Vgl. `https://www.mastertheboss.com/jboss-frameworks/activemq/getting-started-with-activemq-artemis/`, `https://kafka.apache.org/quickstart`.

```
{"winkel": 5.06, "distanz": 1864.00, "qualitaet": 21}
...
{"winkel": 358.31, "distanz": 1866.25, "qualitaet": 24}
{"winkel": 358.98, "distanz": 1865.00, "qualitaet": 24}
{"winkel": 359.72, "distanz": 1864.75, "qualitaet": 24}
{"winkel": 0.41, "distanz": 1864.25, "qualitaet": 24}
{"winkel": 1.09, "distanz": 1863.00, "qualitaet": 23}
{"winkel": 1.78, "distanz": 1862.75, "qualitaet": 22}
...
```

Der Winkel wird in Grad in dem Bereich 0 - 360 angegeben. Die Datei enthält eine Folge von Scans. Der nachfolgende Scan beginnt wieder bei 0 Grad. Die Distanz wird in Millimeter angegeben. Die Qualität eines erfassten Punkt wird in dem Bereich 0 (hohe Unsicherheit) bis 50 (hohe Genauigkeit) angegeben.

Die Datei `Lidar-scans.txt` befindet sich in dem Felix-Ordner.

Lösen Sie die folgenden Aufgaben mit einem JMS-Message System wie etwa *Apache ActiveMQ Artemis*.

1. Einlesen der Datei. Für jeden Messpunkt soll eine Nachricht erzeugt und in einen Topic (z.B. Scans) geschrieben werden. Erstellen Sie hierzu eine Produzenten-Applikation, die die Einträge in der Datei ausliest und als Nachrichten an den Broker sendet.

2. Eine Konsumenten-Applikation soll die unter 1. erzeugten Nachrichten einlesen und folgendermaßen verarbeiten. Alle Werte des ersten Scans sollen mit der Zahl 1 ausgezeichnet werden, die Zahl 2 für den nächsten Scan, usw. Ein konsumierter Messwert soll nun zusammen mit der Scan-Zahl in einen weiteren Topic geschrieben werden. Hinweis: Ein neuer Scan beginnt, wenn der Winkel einer Folgemessung wieder bei Null Grad aufwärts beginnt.

3. Abstandsberechnung: Es soll der Abstand zwischen jeweils zwei Messpunkten berechnet werden. Die geometrischen Grundlagen hierzu sind wie folgt. Gegeben seien zwei Punkte A und B im Polarkoordinatensystem:

   - Punkt A: Winkel α_1 und Radius r_1
   - Punkt B: Winkel α_2 und Radius r_2

   Zunächst können die beiden Punkte in kartesische Koordinaten $P_1(x_1, y_1)$ und $P_2(x_2, y_2)$ konvertiert werden:

   - $x_1 = r_1 * \cos(\alpha_1)$, $y_1 = r_1 * \sin(\alpha_1)$
   - $x_2 = r_2 * \cos(\alpha_2)$, $y_2 = r_2 * \sin(\alpha_2)$

   Der Abstand zwischen den Punkten P_1 und P_2 ergibt sich aus folgender Formel:

   $$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

   Für die in 2. publizierten Nachrichten sollen die Abstände berechnet werden und jeweils zusammen mit der Scan-Nummer in einen weiteren Topic (z.B. Distanzen) geschrieben werden.

   Hinweis: Von der Abstandsberechnung sollen alle Messungen, bei denen die Qualität kleiner einem konfigurierbaren Wert (z.B. 15) sind, ausgeschlossen werden.

4. Schließlich sollen die Entfernungen aus dem Topic von 3. ausgelesen werden und die Gesamtdistanz eines Scans berechnet werden. Alle Werte mit derselben Scan-Nummer werden somit addiert werden. Die jeweiligen Gesamtlängen sollen mit der Angabe der Scan-Nummer in folgendem Format in einer Datei abgelegt werden:

```
{"scan": 1, "distance": 10688.28}
{"scan": 2, "distance": 10315.19}
{"scan": 3, "distance": 10337.11}
...
```

Überlegen Sie sich geeignete Datenstrukturen zur Repräsentation und Verarbeitung der erforderlichen Informationen. Achten Sie darauf, dass Sie ausgewählte Datenstrukturen und Berechnungen (Repräsentation von Punkten, Berechnung von Abständen, etc.) auslagern und für die folgende Aufgabe 3 direkt nutzen können.

Wie können Sie die Korrektheit Ihrer Lösung überprüfen?

Aufgabe 3 [30. Oktober und 6. November]

In dieser Aufgabe sollen Sie die Datenverarbeitung von Aufgabe 2 mit *Apache Kafka* als Messaging-Broker realisieren.

Hinweis: Sie können das Hinzufügen der Scan-Zahl, die Abstandsberechnung, die Berechnung der Gesamtdistanz eines Scans sowie das Erstellen der Ausgabedatei in *einer* Konsumenten-Anwendung umsetzen.

Vergleichen Sie die berechneten Werte mit denen aus Aufgabe 2.

Schließlich sollen die Datenpunkte eines Scans visualisiert werden. Verwenden Sie hierzu ein Werkzeug Ihrer Wahl wie etwa Python *Plotly* oder *Grafana*.

Aufgabe 4 [13., 20. und 27. November]

Ein elektrisch angetriebenes Fahrzeug kennt seinen Startpunkt und die jeweils relative Bewegung zum letzten Standort. Dies kann beispielsweise über die Rotation der Räder abgeleitet werden. Einfachheitshalber befindet sich das Fahrzeug auf einem Punkt in einer Ebene (d.h. in einem 2D-Koordinatensystem). Der Weg von dem aktuellen Punkt des Fahrzeuges zum nächsten wird als Bewegungsvektor $\vec{v} = \begin{pmatrix} x \\ y \end{pmatrix}$ dargestellt. Durch diesen Vektor kann ausgehend von der aktuellen Position $A = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$ die nächste Position B durch Vektoraddition $B = \begin{pmatrix} x_1 + x \\ y_1 + y \end{pmatrix}$ bestimmt werden.

Das Fahrzeug kennt selbst nicht seine absolute Position (abgesehen vom Startpunkt) und meldet kontinuierlich die Bewegungsvektoren seiner Fahrt an ein weiteres System zur Auswertung. In dem folgenden Szenario bewegen sich mehrere Fahrzeuge, die jeweils einen eindeutigen Namen haben.

Zu den Schnittstellen:

(1) Die Position eines Fahrzeuges bzw. ein Bewegungsvektor wird durch folgenden Typ repräsentiert:

```
public class Position implements Serializable, Comparable<Position> {
  ...
}
```

Fragen am Rande: Warum sollte `Position` die Schnittstelle `Comparable` implementieren? `Position` ist ein "reines" Datenobjekt. Dies würde doch für eine `record` Umsetzung sprechen. Was spricht dagegen?

(2) Die folgende Schnittstelle dient dazu, Fahrzeuge zu erstellen und diese wieder zu löschen. Auch wird die Schnittstelle von Fahrzeugen regelmäßig genutzt, um ihre relativen Bewegungsdaten zu melden:

```
public interface VehicleCommands {
   void createVehicle(String name, Position startPosition) throws Exception;
   void moveVehicle(String name, Position moveVector) throws Exception;
   void removeVehicle(String name) throws Exception;
}
```

Verändert ein Fahrzeug seine Position, so wird dies über die `moveVehicle(...)`-Methode unter Angabe des Fahrzeugnamens und des Bewegungsvektors gemeldet.

(3) Über eine Query-Schnittstelle können Informationen zu den Fahrzeugen abgefragt werden. Es ist auch möglich, die Fahrzeuge zu bestimmen, die sich aktuell auf einer bestimmten Position befinden.

```
public interface Query {
   public VehicleDTO getVehicleByName(String name);
   public Enumeration<VehicleDTO> getVehicles();
   public Enumeration<VehicleDTO> getVehiclesAtPosition(Position position);
}
```

(4) Beachten Sie, dass für die Rückgabe von Daten auf der Query-Seite der Typ `VehicleDTO` verwendet wird. Dieser Typ wird ausschließlich auf der *Read Side* verwendet, um Anfragen an Fahrzeuge und deren Positionen zu beantworten. `VehicleDTO` ist ein "reines" Datenobjekt und hat abgesehen von *Gettern* keine weiteren Funktionalitäten. `getPosition()` liefert die aktuelle Position des Fahrzeugs. Wie oft sich ein Fahrzeug bewegt hat, kann über `getNumberOfMoves()` abgefragt werden.

```
public interface VehicleDTO {
   public String getName();
   public Position getPosition();
   public int getNumberOfMoves();
}
```

Erstellen Sie – basierend auf diesen Schnittstellen-Vorgaben – eine *Event Sourcing* Anwendung gemäß der Komponenten der Grobarchitektur von Folie "CQRS with Event Sourcing". Diese Anwendung soll sukzessive in mehreren Versionen erstellt werden.

Version 1

Bei der Umsetzung dieser Basisversion können Sie wie folgt vorgehen:

- Erstellen Sie eine oder mehrere Klassen für die erforderlichen *Commands* sowie den *Command Handler*. Sie benötigen ein Domänenmodell, um Validierungen durchführen zu können. Soll beispielsweise ein weiteres Objekt mit einem bereits vergebenen Namen angelegt werden, so wird dieser Befehl nicht ausgeführt. Wird ein Befehl nicht ausgeführt bzw. abgelehnt, so soll dies über eine Exception dem Aufrufer mitgeteilt werden. Ein `moveVehicle(...)`-Befehl soll abgelehnt werden, wenn der Bewegungsvektor der Null-Vektor ist.

- Einfachheitshalber soll in dieser Version 1 das Domänenmodell nicht auf Basis der im *Event Store* abgelegten Ereignisse aufgebaut werden. Stattdessen kann eine Map zur Verwaltung der aktuell vergebenen Fahrzeugnamen verwendet werden, weitere Informationen werden im Domänenmodell (zunächst) nicht benötigt. Wird ein Kommando akzeptiert, werden die erforderlichen Ereignisse erzeugt. Ein Beispiel für ein Ereignistyp ist etwa `VehicleCreatedEvent`. Im einfachsten Fall wird ein Kommando auf ein entsprechendes Ereignis abgebildet. Wird ein Befehl akzeptiert, muss das Domänenmodell

entsprechend aktualisiert werden.

- Die erzeugten Ereignisse werden im *Event Store* abgelegt. Nutzen Sie *Apache Kafka* zur Speicherung der Ereignisse. Der *Command Handler* nimmt hierbei die Rolle des Nachrichtenproduzenten ein. Wird ein Befehl angenommen und durchgeführt, werden hieraus Ereignisse abgeleitet, die dem *Event Store* übergeben werden. Die Projektion konsumiert die erzeugten Ereignisse und aktualisiert das *Query Model*.
- Implementieren Sie die `Query` Schnittstelle. Überlegen Sie sich eine geeignete Datenstruktur für das zugrundeliegende *Query Model*. Das *Read Repository* kann beispielsweise Maps verwalten, um Anfragen auf einfache Weise beantworten zu können.
- Nun benötigen Sie noch eine Projektion, die die im *Event Store* eingegangenen Ereignisse auswertet und das *Query Model* aktualisiert.

Erstellen Sie eine Client-Anwendung, die über die Schnittstelle `VehicleCommands` Fahrzeug-Objekte erzeugt und deren Positionen über `moveVehicle(...)` ändert. Auch sollten Fahrzeuge entfernt werden. Über eine zweite Anwendung können Sie die `Query` Schnittstelle nutzen, um den Effekt der durchgeführten Befehle zu prüfen. Alternativ – sicherlich auch ein besserer Ansatz – können Sie JUnit-Tests realisieren.

Achten Sie auf eine strikte Trennung zwischen dem Domänenmodell und dem *Query Model*, sodass beide Komponenten unabhängig voneinander weiterentwickelt werden können.

Überlegen Sie sich, wie Sie auf sinnvolle Weise die Korrektheit und Vollständigkeit Ihrer Lösung prüfen können. Setzen Sie ein Testkonzept um.

Version 2

Nun soll das Domänenmodell (genau genommen relevante Ausschnitte davon) auf Basis der im *Event Store* gespeicherten Ereignisse dynamisch bei der Abarbeitung eines Befehls aufgebaut werden. Hier bietet es sich an, geeignete Funktionen zum Laden eines Aggregats (d.h. eines Fahrzeuges) oder der aktuell vergebenen Fahrzeugnamen zu realisieren. Dies bedeutet auch, dass die in Version 1 verwendete Map zur Speicherung der Objektnamen auf der *Write Side* nicht mehr benötigt wird.

Passenderweise persistiert *Apache Kafka* alle Nachrichten. Zum dynamischen Aufbau können somit alle aufgetretenen / gespeicherten Ereignisse beginnend von der Position 0 ausgewertet werden, um einen Befehl zu validieren.

Version 3

Erweitern Sie Ihr System um folgende Funktionalitäten:

1. Nach einer bestimmten Anzahl von Bewegungen (z.B. 5) eines Fahrzeuges soll dieses entfernt werden. Wird also das fünfte Mal der `moveVehicle(...)` Befehl für ein Fahrzeug aufgerufen, wird diese Bewegung nicht ausgeführt. Stattdessen wird das Fahrzeug gelöscht, der Name wird freigegeben und kann nachfolgend wieder vergeben werden.
2. Bewegt sich ein Fahrzeug auf eine Position, auf die es bereits gewesen ist, soll das Fahrzeug entfernt werden. Der Name des Fahrzeugs wird freigegeben und kann nachfolgend wieder vergeben werden.
3. Betrachten Sie nun folgende Erweiterung. Bei der Durchführung von `moveVehicle(...)` bei einem Fahrzeug soll überprüft werden, ob sich auf der neuen Position bereits ein Fahrzeug befindet. In diesem Fall soll letzteres entfernt werden. Prüfen Sie folgende Optionen zur Umsetzung:
   - Nutzung der Query-Schnittstelle zur Positionsüberprüfung.

- Anpassen des Algorithmus zum dynamischen Aufbau des Domänenmodells.
- Erweiterung des Domänenmodells um ein weiteres Aggregat ("Positionsaggregat").
- Regelmäßiges Wegschreiben von Zwischenständen ("Snapshots") in einen persistenten Speicher.

Diskutieren und dokumentieren Sie die jeweiligen Vor- und Nachteile. Setzen Sie mindestens eine Variante um.

Version 4 (Optional)

Erstellen Sie nun eine zweite Projektion, die das Query-Modell in eine Datenbank wie etwa *Redis* abzulegt.

Aufgabe 5 [4. und 11. Dezember]

In den folgenden Aufgaben soll mit Hilfe unterschiedlicher Programmiermodelle die Verarbeitung kontinuierlicher Datenströme untersucht werden. Hierzu wird ein Szenario aus der Domäne "Verkehrsüberwachung" betrachtet. Zum einen sind die zu lösenden Aufgaben in diesem Kontext schnell zu verstehen, ohne dass tiefgehendes Domänenwissen aufgebaut werden muss. Zum anderen sind die durchzuführenden Datenanalysen und -aufbereitungen repräsentativ für eine Vielzahl anderer Fachdomänen.

Die auf einem Streckenabschnitt installierten Sensoren erzeugen Datensätze der folgenden Form:

```
2025-05-13T18:06:57.024Z 2 10.3
2025-05-13T18:06:57.131Z 2 8.9
2025-05-13T18:06:57.530Z 2 11.0
2025-05-13T18:06:57.829Z 1 5.5
2025-05-13T18:06:58.486Z 2 10.6
2025-05-13T18:06:58.977Z 1 6.2
...
```

Eine Zeile soll folgendermaßen interpretiert werden: Der führende Zeitstempel gibt den Zeitpunkt der Datenerzeugung an. Die folgende Zahl legt den eindeutigen Namen eines Sensors fest. Die nachfolgende Gleitkommazahl gibt die zu dem Zeitpunkt gemessene Geschwindigkeit des Sensors in der Einheit Meter pro Sekunde (m/s) an. In dem ersten Datensatz hat der Sensor 2 die Geschwindigkeit 10.3 m/s am 13.5.2025 um 18:06:57 erfasst.

Die Datei `Trafficdata.txt` befindet sich in dem Felix-Ordner.

Erstellen Sie zunächst ein Programm, das die Datensätze aus der Datei ausliest und für jeden Eintrag eine Nachricht in einen *Apache Kafka* Topic einstellt. Die Datensätze sollen vom Typ String dargestellt werden. Achten Sie darauf, dass der in einem Datensatz enthaltene Zeitstempel in dem erzeugten *Record* eingetragen wird (Verwendung der Ereigniszeit).

Eine *Apache Beam* Pipeline soll diese Datensätze über den KafkaIO-Adapter einlesen und die Durchschnittsgeschwindigkeit im km/h für jede Messstelle (d.h. Sensor) und Zeitfenstern (*Batch Window*) von 10 Sekunden berechnen. Die berechneten Geschwindigkeitswerte sollen in folgendem Format in eine Datei geschrieben werden:

```
[18:13:38.013, 18:13:48.013) SensorId=1 AverageSpeed=26.28
[18:13:38.013, 18:13:48.013) SensorId=2 AverageSpeed=23.5
[18:13:48.013, 18:13:58.013) SensorId=1 AverageSpeed=22.36
...
```

Hinweise zum Vorgehen:

1. Es bietet sich an, einen als String gelesenen Datensatz in einen Typ (beispielsweise `SpeedEvent`) mit einer entsprechenden *Transforms* zu überführen. Bei geeigneter Konfiguration des KafkaIO-Adapters können die Zeitstempel (d.h. die Ereigniszeit) automatisch übernommen werden.

2. Nachfolgend können verschiedene *Transforms* erstellt werden, um die geforderten Berechnungen durchzuführen: Filter, um negative Geschwindigkeitswerte von der weiteren Bearbeitung auszuschließen, `GroupByKey` für die Gruppierung der Geschwindigkeitswerte nach Sensor, `Combine` für die Ermittlung der Durchschnittsgeschwindigkeit, `Window` für die Erstellung von Zeitfenstern, etc.

3. Der Typ `IntervalWindow` hat die Methoden `start()` und `end()` zur Abfrage des Start- und Endzeitpunkts eines Zeitfensters. Das Zeitfenster kann als Argument in der Methode `processElement( ... , BoundedWindow window, ...)` deklariert werden. Bei Aufruf der Methode wird das aktuelle Zeitfenster vom Laufzeitsystem bereit gestellt.

Was müsste in Ihrem Programm geändert werden, wenn ein *Sliding Window* der Länge von 10 Sekunden mit der Wiederholungsgrate von 5 Sekunden genutzt werden sollte? Probieren Sie es aus.

Aufgabe 6 [18. Dezember und 8. Januar]

Die Berechnung der Durchschnittsgeschwindigkeiten der Sensoren aus Aufgabe 5 soll nun mittels CEP und Esper / EPL umgesetzt werden. Erstellen Sie eine (oder mehrere) geeignete EPL-Anfrage(n) und *Listener*, die die geforderten Berechnungen durchführen und die ermittelten Geschwindigkeitswerte ausgeben.

Hinweise zum Vorgehen:

- Ihre CEP-Anwendung sollte zunächst die Datensätze aus dem *Apache Kafka* Topic auslesen und als Objekte von einem Java-Typ (z.B. `SpeedEvent`) internalisieren. Es bietet sich an, den Zeitstempel der Ereigniszeit in einem Attribut bei `SpeedEvent` abzulegen.

- Definieren Sie nun eine EPL-Anfrage, die die Durchschnittsgeschwindigkeiten berechnet und Objekte von einem weiteren Typ (beispielsweise `AverageSpeedEvent`) erzeugt. Verwenden Sie `#ext_timed_batch(...)` für die Festlegung der Zeitfenster auf Basis der Zeitstempel in `SpeedEvent`.

- Der für die EPL-Anfrage registrierte *Listener* soll die berechneten Werte auf der Konsole ausgeben oder alternativ in eine Datei schreiben.

Vergleichen Sie die berechneten Durchschnittsgeschwindigkeiten mit denen von Aufgabe 5. Gibt es Unterschiede? Was könnten die Gründe hierfür sein?

Aufgabe 7 [15. Januar und 22. Januar]

In dieser Aufgabe wird die Idee des *Complex Event Processing* vertieft. Exemplarisch soll eine stärkere Geschwindigkeitsveränderung auf einem Streckenabschnitt betrachtet werden. Ein solches Ereignis könnte auf eine Staubildung oder die Auflösung eines Staus auf einem Streckenabschnitt hinweisen.

Ergänzen Sie Ihre Lösung von Aufgabe 6 wie folgt. Unterscheiden sich die Durchschnittsgeschwindigkeiten zweier aufeinanderfolgender `AverageSpeedEvent`s um mehr als 20 km/h, soll hierfür ein weiteres Ereignis (*complex event*) erzeugt werden. Definieren Sie hierzu einen entsprechenden Ereignistyp und EPL-Anfrage. Hilfreich bei der Definition der EPL-Anfrage ist die Verwendung des Temporaloperators `->`. Überprüfen Sie die Korrektheit der Lösung.

Wie müsste die Lösung mit *Apache Beam* von Aufgabe 5 erweitert werden, um diese Aufgabenstellung zu lösen? Beschreiben Sie ein mögliches Umsetzungskonzept. Dieses müssen Sie nicht umsetzen.

Aufgabe 8 (optional)

Implementieren Sie das Read-Process-Write-Pattern mit *Apache Kafka*. Betrachten Sie insbesondere Fehlersituationen, die zeigen, dass alle Datensätze tatsächlich genau einmal verarbeitet werden.

Aufgabe 9 (optional)

In den bisherigen Praktikumsaufgaben wurden vorgegebene Aufgabenstellungen mit unterschiedlichen Technologien und Frameworks bearbeitet. Das Ergebnis war jeweils eine lauffähige Implementierung. Hierbei haben Sie das jeweilige Programmiermodell angewendet und die Arbeitsweise bzw. Fähigkeiten der Technologien praktisch erproben können.

In dieser abschließenden Aufgabe sollen Sie ein noch tieferes Verständnis bezüglich der in der Vorlesung eingeführten Konzepte, Technologien, Frameworks und Programmiermodelle erwerben.

Konkret sollen Sie folgende Systeme

- Java Messaging System (JMS)
- Apache Kafka
- Apache Beam
- Complex Event Processing (CEP) mit EPL / Esper

miteinander vergleichen. Damit verbunden sollen Sie eine Bewertung der aus Ihrer Sicht jeweiligen Stärken und Schwächen vornehmen bzw. den jeweiligen Erfüllungsgrad von Kriterien einschätzen.

Bei der Bewertung können Sie sich anhand der folgenden Kriterien orientieren:

1. Repräsentation von Ereignissen: Wie und mit welchen Datenstrukturen werden Ereignisse repräsentiert? Wie flexibel schätzen Sie die Datenstrukturen ein? Werden Vererbungsstrukturen unterstützt? Welche Metainformationen werden mitgeführt?
2. Wie werden Ereignisse / Nachrichten erzeugt und wie können diese dem Broker übermittelt werden? Welche Schritte sind hierzu erforderlich? Können Ereignisse gebündelt übertragen werden?
3. Wie kann eine Konsumentenanwendung Ereignisse / Nachrichten von dem Broker entgegennehmen? Welche Stategien werden hierzu angeboten?
4. Werden Ereignisse / Nachrichten dauerhaft gespeichert?
5. Welche Auslieferungs- und Verarbeitungsgarantien werden unterstützt?
6. Welche Maßnahmen müssen ergriffen werden, damit Ereignisse / Nachrichten gegen einen unberechtigten Zugriff gesichert werden können?
7. Welche Programmiermodelle werden angeboten, um eintreffende Nachrichten / Ereignisse zu verarbeiten? Werden zustandsbehaftete Transformationen wie etwa Aggregationsfunktionen unterstützt und können Nachrichten / Ereignisse zeitlich gruppiert werden? Ist eine Auswertung nach Ereigniszeit möglich?

8. Welche Skalierungsmöglichkeiten werden vom System unterstützt, um mitwachsen zu können hinsichtlich Datenvolumen, Datenfrequenz sowie Anzahl von Produzenten und Konsumenten?

9. Gibt es Konzepte hinsichtlich einer Ausfallsicherheit des Systems? Wenn ja, wie ist die grundsätzliche Arbeitsweise.

10. Typsicherheit: Wie typsicher ist das Programmiermodell? Wie können Fehler bei der Interpretation einer Nachricht erkannt werden? Z.B. erwartet ein Konsument einen Ereignistyp A, erhält jedoch eine Instanz vom Typ B.

11. In welchen Programmiersprachen können die Produzenten- und Konsumentenanwendungen erstellt werden?

Geben Sie auch eine persönliche Einschätzung zu den Stärken, Schwächen und dem zukünftigen Potenzial der genannten Technologien.