Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Finn V. Dohrn

Evaluierung von parallelen Datenströmen in komplexen Event Prozessen anhand des Frameworks „Kafka Streams“

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Finn V. Dohrn

Evaluierung von parallelen Datenströmen in komplexen Event Prozessen anhand des Frameworks „Kafka Streams“

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr-Ing. Olaf Zukunft
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 31. Juli 2017

Finn V. Dohrn

Thema der Arbeit

Evaluierung von parallelen Datenströmen in komplexen Event Prozessen anhand des Frameworks „Kafka Streams“

Stichworte

Apache Kafka, Big Data, CEP, Komplexe Eventprozesse, parallele Datenströme

Kurzzusammenfassung

Das Ziel der vorliegenden Bachelorarbeit war es das Verhalten von parallelen Datenströmen in komplexen Event Prozessen zu simulieren und zu evaluieren. Dazu wurden mit dem Framework „Kafka Streams“ vier verschiedene Szenarien eines möglichen Echtzeit Analyse-Dashboards realisiert. Es wurden verschiedene Events generiert, dann mit einander verknüpft und anschließend ausgewertet. Die Ergebnisse bestätigen die schnelle und zuverlässige Verknüpfung von Events. Deswegen ist „Kafka Streams“ besonders geeignet verschiedene Informationen in Echtzeit zu vereinen. Die Bachelorarbeit ist sowohl für Studenten die sich im Bereich *Big Data* vertieft haben als auch für Informatiker die in diesem Bereich arbeiten.

Finn V. Dohrn

Title of the paper

Evaluation of parallel data streams in complex event processing with the help of the framework "Kafka Streams"

Keywords

Apache Kafka, Big Data, complex event processing, CEP, Apache Kafka, parallel data streams

Abstract

It was the aim of the present bachelor thesis to evaluate and simulate the behaviour of parallel data streams in complex event processes. In addition, were realise with the Framework "Kafka Streams" four different scenarios of a possible real time analysis-dashboard. Differenz events were generated, then join together with each other and subsequently evaluted. The results confirm the quick and reliable join of events. So "Kafka Streams" is particularly suitably to unite different information on real-time. This bachelor thesis is for students as well as for information scientists which deepened in the area of *Big Data*.

Danksagung

An dieser Stelle möchte ich all jenen danken, die mich im Rahmen dieser Bachelorarbeit begleitet haben. Ganz besonders möchte ich Herrn Zorn der Firma inovex danken, der meine Arbeit durch seine fachliche und persönliche Unterstützung begleitet hat. Der Firma inovex danke ich für das Vertrauen.

Darüber hinaus möchte ich mich bei meinem ältesten Bruder bedanken, der mich nicht nur zur Informatik gebracht hat und somit indirekt zu einem Studium bewogen hat, sondern mir auch am Schluss bei den Korrekturen an der Seite stand.

Einen großen Dank möchte ich auch meiner Partnerin aussprechen, die mich während der ganzen Arbeit mit viel Geduld moralisch unterstützt hat.

Inhaltsverzeichnis

1. Einleitung

In der heutigen vernetzten Welt findet ein ständiger Austausch von Daten statt. So generieren Geräte, Sensoren und Industrieanlagen, aber auch klassische Software wie ein e-Commerce System immense Daten unterschiedlichster Art die es zu bewältigen gilt – und das möglichst mit hoher Geschwindigkeit bis hin zur Echtzeit.

Big Data oder *Smart Data* sind typische Begriffe um diese charakteristischen Eigenschaften zu beschreiben. Gleichermaßen dienen Technologien aus diesen Bereichen dazu, diese vernetzten Daten zu verarbeiten und zu analysieren. Dabei steigt auch der Bedarf an Datenverarbeitung in Echtzeit – der sofortigen Reaktion auf ein erkanntes Muster in einer großen Menge von Signalen um dadurch Entscheidungen zu fällen.

Deswegen wird ein Großteil der heutigen Verarbeitung solcher Daten der vernetzten Welt durch Software erledigt, die permanent in Betrieb ist. Es kann so zeitnah auf eingehende Ereignisse (engl. *events*) reagiert werden, die den Zustand einer Anwendung verändern, um eine mögliche Prognose dieser Veränderung aufzeigen zu können.

(vgl. Hedtstück, 2017; Estrada and Ruiz, 2016).

1.1. Motivation

Die Softwaretechnologie Complex Event Processing (abgek. CEP) kann diese Art, Datenströme zu erkennen, zu verarbeiten und darauf zu reagieren, ermöglichen. Treffen mehrere Ereignisse ein, erzeugt aus der Umwelt, spricht man auch von einem *Ereignis Strom* (engl. *event stream*). Es lässt sich eine Teilmenge dieser Ereignisse erkennen, dessen Eigenschaften und Beziehungen ein bestimmtes Muster aufweisen. Das versetzt die CEP-Software wiederum in die Lage, die Fortsetzung der Anwendung zu beeinflussen und ein vorgegebenes Ziel zu verfolgen. Die Verarbeitung eines solchen Stroms wird auch als *Event Stream Processing* bezeichnet.

Ein Echtzeit Analyse-Dashboard für eine e-Commerce Webapplikation wie einen Online Shop, stellt einen typischen Anwendungsfall dar. Neben den Klicks, die von Benutzern generiert werden, existieren auch Transaktionen wie Suchanfragen oder Käufe die von der Applikation gesammelt werden können. Diese Daten können genutzt werden um in Echtzeit Informationen

zu verknüpfen um so Erkenntnisse über „Top Produkte“ oder „Top Suchanfragen“ zu finden und darauf wirtschaftlich zu reagieren. Eine weitere nützliche Anwendung kann das Verknüpfen mit bereits existierenden Daten, wie Benutzerdaten, sein um so beispielsweise Betrugsfälle präventiv aufzudecken und darauf zu reagieren. Dafür generieren die Anwender diese Klick- und Transaktionsereignisse die wiederum von einem CEP-System verarbeitet werden und nach der Verarbeitung als Wert im Dashboard angezeigt werden.

Dafür ist in den letzten Jahren eine ganze Reihe neuer *Big Data*-Technologien entstanden. *Apache Kafka* ist eine dieser Technologien, dass insbesondere die Verarbeitung von multiplen Datenströmen mit Hilfe eines besonderen Datenverteilungsmechanismus für Datenströme, dem verteilten Log, ermöglicht. Nur dadurch kann die große Datenmenge, die in aktuellen Anwendungen produziert wird, schnell verarbeitet werden.

Mit Hilfe der in *Apache Kafka* implementierten Streams lassen sich viele Probleme des klassischen *Big Data* lösen. Dafür entstehen neue Herausforderung durch die Echtzeitanforderungen. Zum Beispiel kann eine Webapplikation die Klicks der Nutzer protokollieren, aber auch gleichzeitig Aktionen wie das Speichern aufnehmen. Es gibt also einen großen Datenstrom von Klicks und parallel viele kleinere Datenströme die technische Transaktionen beschreiben. Die Anforderung liegt darin, viele verschiedene Informationen nicht nur schnell, sondern in nahe zu Echtzeit zu verarbeiten, um sofort auf solche Nutzeraktionen reagieren zu können.

Interessant ist nun das Verhalten, der verschiedenen und parallelen Datenströme, wenn man sie verknüpft um damit später einen gemeinsamen Zusammenhang zu erkennen und daraus neue Erkenntnisse für die Anwendungslogik zu ziehen.

(vgl. Estrada and Ruiz, 2016, Seite 2ff.).

1.2. Zielsetzung

Ziel dieser Arbeit ist es die Grenzen und Möglichkeiten des Dashboard-Szenarios zu simulieren und im Anschluss zu analysieren. Dafür wird das Verhalten von parallelen Eventströmen untersucht und evaluiert. Die Software generiert verschiedene Eventströme einer simulierten Webapplikation – die benötigten Ereignisse konkreter Testsituationen. Mit Hilfe von Zufallszahlen werden diese Ereignisse aber möglichst einem realen Szenario, das in der Motivation kurz erwähnt wurde, nahe gelegt.

In der Arbeit werden konkret vier Anwendungsfälle implementiert und anschließend diskutiert. Im ersten Anwendungsfall soll die Auswirkung verschieden großer Auswertungsfenster (siehe 2.4.4). und Arten evaluiert werden. Dafür werden zwei Eventströme, die mit zufällig

generierten Events gefüllt sind, miteinander verknüpft um anschließend die Verluste der nicht verknüpfbaren Events in Abhängigkeit der Fenstergröße und Typ zu bestimmen.

Der zweite Anwendungsfall untersucht das Verhalten von zwei sehr unterschiedlich langen Eventströmen, in dem unterschiedliche Zeitstempel (siehe 2.4.3) zur Verarbeitung verwendet werden. Zu dem wird die Größe der temporär angelegten Hilfstabelle untersucht.

Ein weiteres Ziel der Arbeit ist das Finden einer effizienten Implementierung, die das Verknüpfen von mehreren Ereignisströmen ermöglicht.

Zum Abschluss der Arbeit sollen die Daten für das Betrugsfall-Szenario geeignet vorbereitet werden, in dem ein Eventstream mit einer persistenten Benutzerdaten-Tabelle verknüpft wird, um mehr Informationen für den Erkennungsalgorithmus zur Verfügung zu stellen.

Mit der Evaluierung sollen präzise Aussagen zum Verhalten von verknüpften Kafka Streams getroffen werden können und die vorangestellten Annahmen zu bestätigen. Mit Hilfe dieser Arbeit kann diese Technologie dann zielgerichteter in Anwendungen verwendet werden und neue Einsatzgebiete offen legen.

1.3. Aufbau der Arbeit

Kapitel 1 Einleitung Im ersten Kapitel findet man die Motivation für diese Arbeit. Es wird kurz erklärt welche Bedeutung die Datenverarbeitung von vielen Daten in kurzer Zeit für heutige Technologien hat. Außerdem wird das Ziel dieser Arbeit beschrieben.

Kapitel 2 Grundlagen Im Grundlagen Kapitel werden die wichtigsten Grundbegriffe von komplexen Eventprozessen eingeführt. Das Verknüpfen von zwei Streams wird in der Theorie vorgestellt.

Kapitel 3 Anforderungsanalyse Das Kapitel startet mit einer Beschreibung des Anwendungsszenarios. Mit Hilfe der Anforderungsanalyse werden dann die Bedingungen, die für die Umsetzung der Arbeit erforderlich sind geklärt und erläutert. Dafür werden die Anforderungen in funktionale, nicht-funktionale und andere Anforderungen unterteilt.

Kapitel 4 Design Im vierten Kapitel geht es um das Design der Software. Dabei wird vorab geklärt weshalb sich für *Apache Kafka* mit Kafka Streams entschieden wurde und wie diese Technologien funktionieren. Es wird der Aufbau eines Events im allgemeinen gezeigt und wie die Use Cases der späteren Evaluierung ablaufen. Außerdem wird eine Übersicht über die Architektur der Software gegeben.

Kapitel 5 Realisierung Das darauffolgende Kaptiel erläutert die Realisierung der Software. Es wird beschrieben wie die Software mit Hilfe des Frameworks Kafka Streams implementiert wurde. Die daraus entstandenen verknüpften Streams dienen als Grundlage der Evaluierung.

Kapitel 6 Evaluierung Anschließend wird im sechsten Kapitel evaluiert. Dafür werden die Testergebnisse der verschiedenen simulierten Situationen ausgewertet und in Kontext mit den Anwendungsszenario gebracht, um voran gestellte Annahmen bestätigen oder widerlegen zu können.

Kapitel 7 Fazit und Ausblick Im letzten Kapitel wird die Arbeit zusammen gefasst und die Erkenntnisse bewertet. In dem Ausblick wird dann erläutert, inwiefern die Arbeit noch vertieft werden könnte um weitere Erkenntnisse über das Verhalten paralleler Eventströme zu erhalten um auf dieser Bachelorarbeit aufzubauen.

2. Grundlagen

Bevor sich die Arbeit konkret mit dem Szenario auseinander setzt, werden noch ein paar Grundbegriffe eingeführt, die zum Verständnis der Arbeit notwendig sind. Dieses Kapitel soll helfen die im Programmcode verwendeten Methoden und Parameter besser zu verstehen um so das Gesamtszenario nachvollziehen zu können.

2.1. Big Data

Der Ursprung dieser Arbeit liegt im *Big Data* Umfeld. Der abstrakte Oberbegriff *Big Data* steht für jegliche Art und Anzahl von Daten, die nicht mehr mit traditionellen Analyseverfahren bewältigbar sind und deswegen neue Techniken und Technologien benötigen. Eine einheitliche Definition für diesen Begriff gibt es in der vorherrschenden Literatur nicht, da es sich um ein relativ jungen Begriff handelt.

Weniger umstritten sind jedoch die Eigenschaften von *Big Data*, die auch auf die verwendeten Daten in dieser Arbeit zu treffen. Doug Laney, beschreibt diese Eigenschaften in einem dreidimensionalen „3V-Modell“, dass die Herausforderungen des Datenwachstums beschreiben soll (vgl. Laney, 2001):

1. *volume* definiert die Anforderung, mit immensen Mengen von Daten, die in solchen *Big Data* Anwendungen üblicherweise generiert werden, umzugehen.

2. *velocity* bezeichnet die hohe Geschwindigkeit in der diese Daten generiert, analysiert und verarbeitet werden können.

3. *variety*: Die letzte Eigenschaft steht für die Vielfalt der Datentypen und Quellen. Die meisten Daten die heute vorkommen sind unstrukturiert, unterliegen also keinem bestimmten Schema. Die Herausforderung ist hier mit geeigneten Algorithmen die Daten strukturiert einordnen zu können.

Dieses „3V-Modell“ deckt sich gut mit der Anwendung von Complex Event Processing. Neben vielen zu verarbeiteten Ereignissen, unterschiedlichster Art, zählt aber vor allem die Geschwindigkeit für CEP-Systeme – nämlich Echtzeit.

2.2. Complex Event Processing

Das *Complex Event Processing* (CEP) identifiziert mehrerer Ereignisse eines nicht abbrechenden Flusses von Ereignissen, die zu unterschiedlichen Zeitpunkten stattfinden können. Die identifizierten Ereignisse zeigen ein bestimmtes gemeinsames Muster auf, aus denen Erkenntnisse für die Zukunft gezogen werden, um darauf zielgerichtet reagieren zu können. Ziel eines CEP-Systems ist es die Events möglichst schnell zu verarbeiten, um Wartesituationen zu vermeiden.

Im Gegensatz zu den klassischen Datenbanktechnologien, aus denen bestehende Daten verarbeitet werden, existieren die Daten beim CEP zum Zeitpunkt der Anfrage des Datenmusters noch nicht. Aus dem nicht abbrechenden Strom von Ereignissen werden nach einander die Daten, die zu einem Muster passen, heraus extrahiert.

Das Grundprinzip eines CEP-Systems zeigt die Abbildung 2.1. Verschiedene Quellen Q_n, $n \in \mathbb{N}$, auch *Ereignis-Produzenten* genannt, produzieren durchgehend Ereignisse unterschiedlichster Art die das System erreichen. Unmittelbar nach dem Eintreffen in dem System werden die Ereignisse gefiltert. Unwichtige Ereignisse die für das CEP nicht interessant sind, werden entfernt. Die noch übrig gebliebenen relevanten Ereignisse werden dann von einem Programm aufbereitet. Das heißt, sie werden für die Weiterverarbeitung vorbereitet. Die aufbereiten Ereignisse gehen dann durch die Mustererkennung. Dort werden anhand eines gesuchten Musters Ereignisse identifiziert. Je nach dem ob es sich um eine *detection oriented CEP* oder *computation oriented CEP* handelt, sind die Muster vorher bekannt (*detection oriented*) oder es müssen mit Hilfe einer Wissensbasis und Maschinellen Lernen unbekannte Muster erlernt werden (*computation oriented*). Die erkannten Ereignisse können dann von Ereignis-Konsumenten K_n, $n \in \mathbb{N}$, verwendet werden, um eine gewünschte Reaktion zu bewirken. CEP-Systeme werden in der Regel verteilt auf mehreren Rechnern ausgeführt. Ein verteiltes System erhöht die Ausfallsicherheit und kann die Lasten gleichmäßig verteilen um eine zuverlässige hohe Geschwindigkeit zu garantieren.

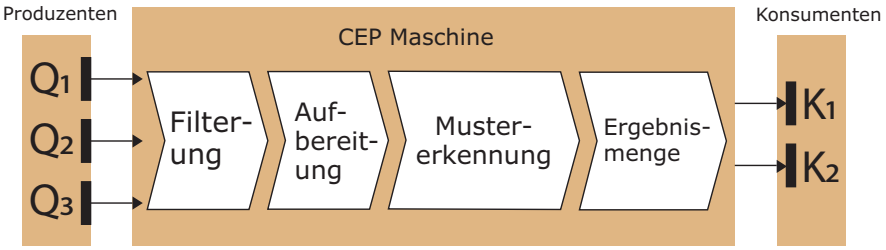

Abbildung 2.1.: Grundlegendes Prinzip einer CEP Maschine.

Ein Muster könnte das Erkennen eines Kaufabbruchs sein. Dafür würde man nach einer Sequenz folgender Ereignisse suchen: *Artikel ansehen*, *Artikel in Warenkorb legen* und *Artikel zeitnah wieder löschen*. Dabei haben diese Transaktionen eine gemeinsame Identifikationsnummer um die Zusammengehörigkeit zu erkennen.

In dieser Arbeit liegt der Fokus vor allem darin zwei oder mehrere Eventströme zu verknüpfen, und anschließend optional zu filtern und aufzubereiten, nicht in der Mustererkennung. Deswegen wird auf das Erkennen von Mustern nicht weiter eingegangen.

(vgl. Hedtstück, 2017; Bruns and Dunkel, 2015).

2.3. Events

2.3.1. Prozesse und Ereignisse

Die Grundlage der Verarbeitung von komplexen Eventprozessen bilden die Prozesse und Events. Ein *Prozess* wird als ein dynamisches System bezeichnet, dessen Ablaufzeit unbegrenzt ist und eine nicht abbrechende Folge von Ereignissen erzeugt.

In einem *dynamischen System* können sich die Zustände der darin liegenden Objekte über einen Zeitraum ändern. Zu den Zuständen der *Objekte* gehören Attribute sowie Beziehungen. Objekte besitzen Eigenschaften, die man auch *Attribut* nennt. Das kann beispielsweise ein Name oder eine Transaktions-ID sein. Als *Beziehung* bezeichnet man eine Gemeinsamkeit die zwischen zwei oder mehreren Objekten besteht.

Eine Veränderung dieser Attribute und Beziehungen, also des Zustandes, verursacht ein Ereignis. Dies findet zu einem Zeitpunkt und nicht über einen Zeitraum statt, dem sogenannten Zeitstempel (engl. *timestamp*). Es beansprucht also keine Zeit. Ein *Ereignis* ist also eine sprunghafte Änderung des Zustandes und verändert mindestens einen Attributwert oder eine Beziehung. Ereignisse lassen sich auch in gleiche *Ereignistypen* kategorisieren. Ein Ereignis aus dieser Menge wird dann als *Instanz* dieses Typs bezeichnet.

Ein Beispiel Ereignis sieht man in Abbildung 2.2. Es besitzt einen Zeitstempel, einen Typen sowie eine eindeutige Identifikationsnummer. Daneben kann es optionale Attribute besitzen wie Name, Suchbegriff, besuchte Webseite oder BenutzerID.

Ein Ereignis das einen eindeutigen Zeitstempel besitzt wird auch als *atomares Ereignis* bezeichnet. Abgrenzt werden muss der Begriff von *Aktivität*, die in einer Zeitdauer abläuft, also Zeit benötigt.

Sobald sich ein Ereignis aus mehreren Teilerereignissen zusammensetzt, spricht man nicht mehr von einem *atomaren Ereignis* sondern von einem komplexen Ereignis.

(vgl. Hedtstück, 2017, Seite 12f).

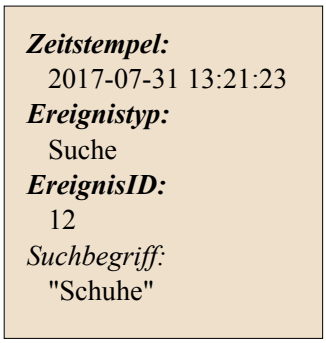

Abbildung 2.2.: Beispiel eines Suchereignises.

2.3.2. Komplexe Ereignisse

Eine endliche Menge von *atomaren* Ereignissen, die auf Grund eines gemeinsamen Musters zu einander in Beziehung stehen, bezeichnet man als *komplexes Ereignis*. Dabei kann die Menge aus eintreffenden, von einem Muster identifizierten oder daraus generierten Ereignissen bestehen (vgl. Zimmer and Unland, 1999).

Innerhalb eines Ereignisstroms kann es mehrere komplexe Ereignisse geben, die jeweils zu einem Ereignismuster passen. Deshalb werden komplexe Ereignisse auch als *Ereignismusterinstanzen* bezeichnet.

Ein exemplarisches Muster könnte eine Folge von Such und Kauf-Ereignissen sein, die direkt hintereinander erfolgen um so einen Kauf zu erkennen der über die Suchmaschine eingeleitet wurde.

Die Aufgabe des CEP-System ist es nun, solche Instanzen von Mustern aus einer unendlichen fließenden Menge von *atomaren* Ereignissen zu erkennen und gemäß einer Operation (Konjunktion, Disjunktion, Negation und vieles mehr) zu der Menge der gesuchten Ereignismuster bei hinzuzufügen.

(vgl. Hedtstück, 2017, Seite 10f.).

2.4. Ereignisströme

Diese endlos fließende Menge von *atomaren* Ereignissen wird als *Ereignisstrom* (engl. *event stream*) bezeichnet. Da jedes Objekt dieser Ereignismenge neben einer eindeutigen Identifikationsnummer und dem Typ auch einen Zeitstempel besitzt, kann man ohne Probleme mehrere Ströme verschiedener Quellen zu einem einzigen Strom zusammenfassen – sofern diese ein einheitliches Zeitsystem verwenden. Mit Hilfe der optionalen Parameter, können die verschiedenen Quellen dann identifiziert werden.

In folgenden Abbildungen verläuft der Stream immer von links nach rechts. Konkret bedeutet das: die Ereignisinstanzen werden von links eingelesen und verlassen rechts die Verarbeitung. Ereignisse werden immer mit e_n durch nummeriert, wo bei $n \in \mathbb{N}$ gilt. Es gilt weiterhin, dass ein Event e_n älter ist als e_{n+1}. Die Indizes geben also eine zeitliche Reihenfolge vor. Die Lücken zwischen den Events entstehen, da nicht jedes Event sofort hintereinander eintritt sondern zeitlich etwas verzögert sein kann.

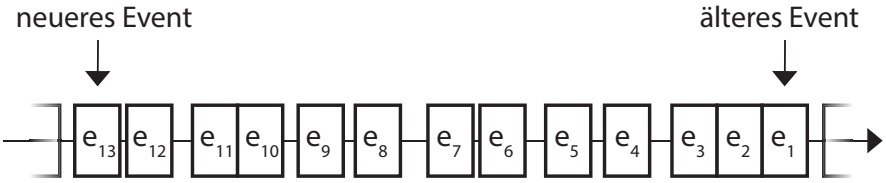

Abbildung 2.3.: Beispiel eines Eventstrom.

Eine Software die Verwaltung und Verarbeitung von Streams bereitstellt, wird als *Stream Data Platform* bezeichnet, die hier auch einfach als Plattform abgekürzt wird. Software-Lösungen die eine Interaktion mit dieser Plattform bieten, nennt man *Stream Processing Systeme*.

(vgl. Hedtstück, 2017, Seite 21 f.).

2.4.1. Event Cloud

Oftmals wird noch zusätzlich eine *Event Cloud* bei komplexen Event Prozessen verwendet. Sie empfängt alle Ereignisse aus unterschiedlichen Quellen die in einem vorgegeben Zusammenhang stehen. Beispielsweise alle Ereignisse eines Unternehmens.

Mit Hilfe des *Publish-Subscribe* Design Pattern lässt sich mit dem CEP-System interagieren. Das Muster funktioniert, so das Ereignis-Produzenten, also die Quellen der Ereignisse, ihre Ereignisse in die Cloud schicken. Das CEP-System teilt dann mit an welchen Ereignistypen oder Quellen es interessiert ist. Je nach Schnittstelle greift die CEP-Software dann selber auf die bereitgestellten Ereignisse zu (*Pull-Prinzip*) oder die Cloud schickt ausgewählte Ereignisse von selbst in das CEP-System (*Push-Prinzip*).

(vgl. Hedtstück, 2017, Seite 24).

2.4.2. Filterung und Aufbereitung

Mit Hilfe einer Filter-Software können vorab irrelevante Ereignisse oder Duplikate beim Eintreffen der Events in das CEP-System aussortiert werden. Um die Verarbeitung zu beschleunigen, wird in den meisten Fällen nur auf den Ereignistyp und die optionalen Attribute geprüft.

Nach dem die Ereignisse vor gefiltert wurden, werden sie durch ein Programm, den Präprozessor, für die Weiterbearbeitung aufbereitet. Aufgaben dieser Aufbereitung sind das entfernen unwichtiger Attribute, hinzufügen fehlender Informationen und das anpassen von Attributwerten an die CEP-Engine. So können beispielsweise Werte vom imperialen Maßsystem auf das metrische System umgerechnet werden.

Das Prinzip der Filterung und Aufbereitung eines Eventstroms wird in Abbildung 2.4 illustriert. Es zeigt wie ein Stream von Daten in das CEP-System ankommt, dann gefiltert wird und im Anschluss von dem Präprozessor aufbereitet wird.

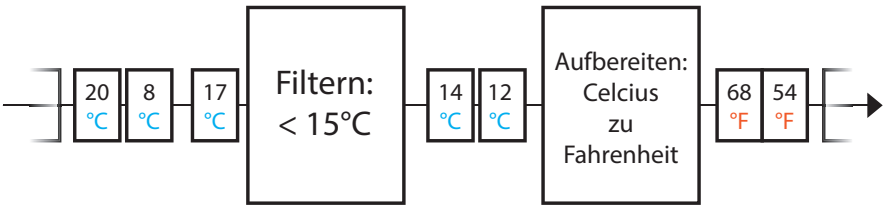

Abbildung 2.4.: Filterung und Aufbereitung von Temperaturdaten.

(vgl. Hedtstück, 2017; Akidau, 2015a).

2.4.3. Systemzeit vs. Eventzeit

Eine weitere wichtige Aufgabe des Präprozessors ist die zeitliche Einordnung von Ereignissen für die spätere Musterkennung. Denn das CEP Programm ordnet dem ankommenden Ereignisobjekt ein Ankunft-Ereignis zu.

Es wird also zwischen dem Zeitstempel des Ereignisobjekts und dem Zeitstempel des Ankunft-Ereignisses unterschieden. Das Ereignisobjekt besitzt selber den *expliziten Zeitstempel*. Es ist der Zeitpunkt in dem das Ereignis erstellt wurde. Deswegen spricht man auch von Realzeit (engl. *event time*).

Das Ankunft-Ereignis wiederum hat den *impliziten Zeitstempel*. Dieser Zeitstempel stammt von der Systemzeit (engl. *processing time*) des CEP-System ab, zum Zeitpunkt der Zuordnung des Ankunft-Ereignis mit dem Ereignisobjekts. Also bei der Ankunft des Ereignis.

Die unterschiedlichen Zeitstempel können auf Grund von Verzögerungen entstehen. Beispielsweise wenn die Verarbeitung der CEP-Maschine zu langsam ist und daher die Ereignisse in einer Warteschlange warten müssen bevor sie einen impliziten Zeitstempel bekommen. Beide Zeitstempel stehen aber in einer klaren Relation zu einander. Und zwar gilt:

$$\text{explizite Zeit} \leq \text{implizite Zeit} \tag{2.1}$$

Innerhalb der Ereignisströme kann die zeitliche Reihenfolge der expliziten Zeitstempel von dem der impliziten Zeitstempel abweichen. Das kann bei der Verwendung unterschiedlicher Ereignis-Produzenten passieren. Ein Produzent könnte einen Vorgang verzögern, so dass ein später eingetretenes Ereignis früher zum CEP-System geschickt wird.

Der Präprozessor hat dann die Aufgabe diesen Datenverkehr konfliktfrei zu halten und die zeitliche Behandlung zu korrigieren. Dafür wird beispielsweise ein Zwischenspeicher verwendet in dem die Ereignisobjekte dynamisch sortiert werden bevor sie die Mustererkennung erreichen. Es können also jederzeit neue Objekte zum Speicher hinzugefügt werden die dann einsortiert werden.

(vgl. Hedtstück, 2017; Akidau, 2015a).

2.4.4. Auswertungsfenster

Nach der Filterung und Aufbereitung ist die Menge der Ereignisse in einem Ereignisstrom immer noch unendlich. Damit eine Verarbeitung möglich ist, muss der Suchraum auf eine endliche Menge beschränkt werden. Das geschieht durch so genannte Auswertungsfenster.

Ein Auswertungsfenster ist ein gleitendes Zeit- oder Längenfenster das immer nur eine bestimmte Anzahl atomarer Ereignisse als Auswahl von Ereignisinstanzen für die Verarbeitung zulässt. Ein Fenster mit einer endlichen Menge von Ereignissen nennt man dann auch *Fensterinstanz*.

Ein Zeitfenster (engl. *Time Window*) hat immer eine feste Zeitdauer. Es werden nur Ereignisse berücksichtigt, die mit ihrem Zeitstempel innerhalb eines Zeitintervalls liegen (beispielsweise ein Fenster von 30 Sekunden). Das Prinzip des *Time Window* wird in Abbildung 2.5 illustriert. Das Δt steht für das gewählte Zeitintervall.

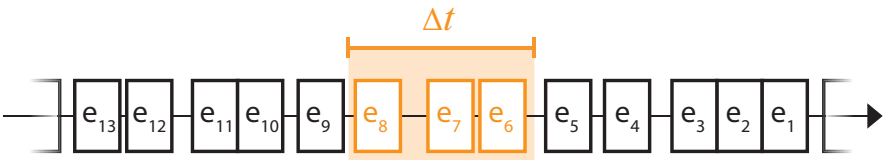

Abbildung 2.5.: Das Time-Window Prinzip

Ein Längenfenster (engl. *Count Window*) wiederum, schaut nicht auf die Zeitstempel der Ereignisse, sondern lässt nur eine gewisse Anzahl Ereignisse zu. Das bedeutet, das Fenster nimmt solange Ereignisse in sein Auswertungsbereich auf, bis die maximale Anzahl erreicht ist (beispielsweise maximal 3 Ereignisse). In Abbildung 2.6 lässt sich das Prinzip gut nachvollziehen.

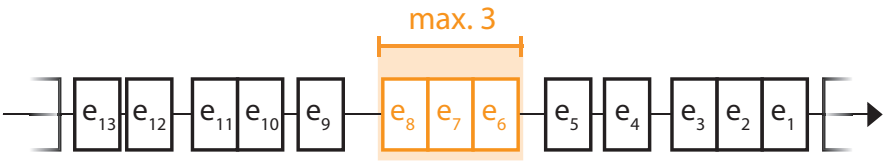

Abbildung 2.6.: Das Count Window Prinzip

Diese beiden Auswertungsfenster können mit verschiedenen Verschleierungsstrategien verwendet werden. Im Folgenden wird eine Auswahl von Strategien vorgestellt. Unter anderem werden Sliding Windows, Landmark Windows und Partitioning erläutert.

Eine andere Auswahlstrategie für Ereignisse neben den Auswertungsfenstern ist der *Event Consumption Mode*, der Instanzen anhand von passenden Ereignismustern einschränkt. Dies wird hier aber nicht näher erläutert, da es für die Arbeit keine Relevanz hat.

(vgl. Hedtstück, 2017; Akidau, 2015b).

Sliding Windows

Sliding Windows, auch als *Hopping Window* bezeichnet, sind Zeit- oder Längenfenster bei dem sich Anfangs- und Endpunkt nach einem ausgewählten Verschiebungsfaktor (engl. *slide size*) bewegen. Anschaulich heißt das, welche Ereignisse in das Fenster aufgenommen und welche entfernt werden. In der Software muss das ganze als dynamische Datenstruktur, beispielsweise als Queue, umgesetzt werden. Je nach dem wie groß dieser Faktor ist, gibt es für die aufeinanderfolgenden Fensterinstanzen drei Möglichkeiten:

1. Die Fensterinstanzen überlappen sich (Rolling Window),
2. eine Fensterinstanz folgt auf die andere Fensterinstanz (Tumbling Window),
3. es gibt eine Lücke zwischen den Instanzen.

(vgl. Akidau, 2015a, Seite 27).

Rolling Window Wenn der Verschiebungsfaktor kleiner als die Fenstergröße ist, spricht man von einem *Rolling Window*. In Abbildung 2.7 wird das Prinzip des Rolling Windows deutlich. Es liegt ein Stream mit Ereignissen vor. Das Längenfenster hat die Größe 3. Die Ereignisse e_1 bis e_5 wurden bereits bearbeitet. Die aktuelle Fensterinstanz beinhaltet die Elemente e_6 bis e_8. Wird nun der Verschiebungsfaktor 1 gewählt, so wird das Ereignis e_6 aus dem Fenster entfernt und e_9 hinzugefügt, so dass das Fenster immer noch die Größe 3 behält. Die darauf folgende Instanz hätte würde dann das Element e_7 entfernen und e_{10} hinzufügen.

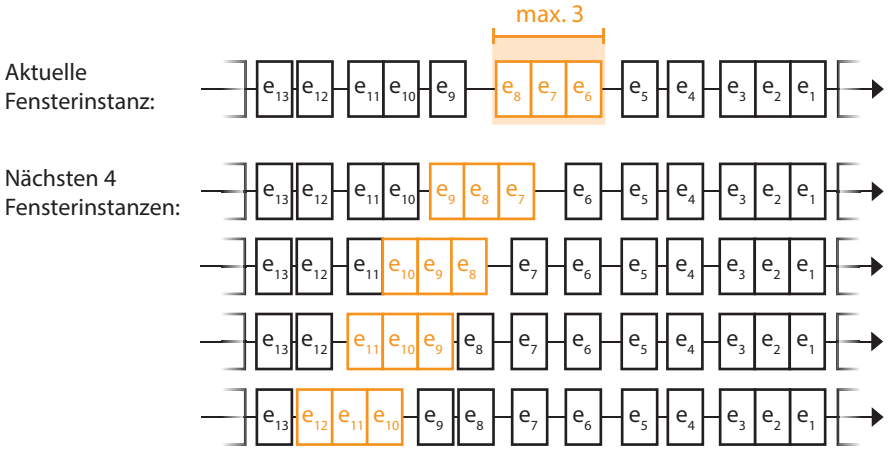

Abbildung 2.7.: Sliding Window mit Fenstergröße 3 und Verschiebefaktor 1 (Rolling Window)

(vgl. Hedtstück, 2017; Akidau, 2015b).

Tumbling Window Ist der Verschiebungsfaktor größer oder gleich der Fenstergröße, spricht man von einem *Tumbling Window*. Es wird nun der gesamte Inhalt entfernt, sobald das Fenster maximal gefüllt ist. Deswegen ist das Fenster zunächst leer und füllt sich nach und nach mit den Elementen e_9, e_{10} und e_{11}. Man sieht nochmal in Abbildung 2.8, das sich die Fenster nie überlappen. Die Fensterinstanzen sind dann disjunkt.

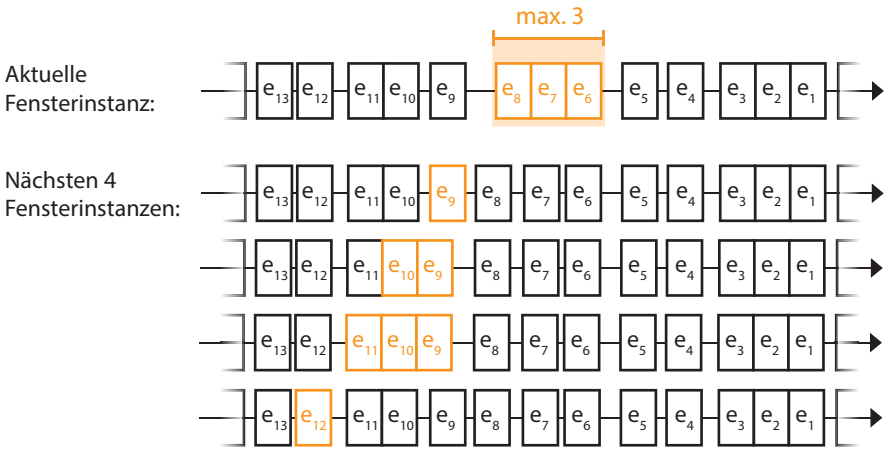

Abbildung 2.8.: Sliding Window mit Fenstergröße 3 und Verschiebefaktor 3 (Tumbling Window)

(vgl. Hedtstück, 2017, Seite 27).

Landmark Windows

Ein *Landmark Window* ist ähnlich wie ein Tumbling Window, arbeitet nur nicht mit Zeitintervallen sondern mit einem konkreten Zeitpunkt in dem das Fenster geschlossen wird – also keine neuen Ereignisse aufnimmt. Nach diesem Zeitpunkt wird eine neue Fensterinstanz geöffnet. Man sieht auch hier die disjunkte Eigenschaft von Tumbling Windows (siehe 2.4.4, Absatz „Tumbling Window“).

Ein beispielhaftes *Landmark Window* wäre „immer Sonntag um 23:59 Uhr“. Zu diesem Zeitpunkt würde das Fenster dann schließen und eine neue Instanz öffnen. In diesem Fall würde Sonntag um Mitternacht die Fensterinstanz schließen und eine neue Instanz bis nächsten Sonntag 23:59 Uhr öffnen. Dieser Vorgang wiederholt sich dann wöchentlich.

Je nach Literatur kann ein *Landmark Window* auch ein Fenster sein, das zu einem bestimmten Zeitpunkt geöffnet wird und nicht schließt. Diese Art Fenster sind aber unbrauchbar, da ihre Größe, aufgrund ihrer Eigenschaften, sehr stark wachsen.

(vgl. Hedtstück, 2017; Matysiak, 2012).

Session Windows

Ein *Session-Window* ist ein flexibles Fenster, dessen Größe von der Aktivität, zum Beispiel eines Nutzers, abhängt. Das Fenster schließt also sobald diese Aktivität fehlt und öffnet sobald eine neue Aktivität gestartet wird.

(vgl. Guy, 2016).

Partitioning

Für den Fall das nur atomare Ereignisse eines bestimmten Typs berücksichtigt werden sollen, werden die Eventströme nach ihren Typen in disjunkte Teilströme portioniert. Die Teilströme enthalten dann nur die relevanten atomaren Ereignisse worauf die weitere Verarbeitung von CEP angewandt werden kann. Das ganze wird dann *Partitioned Window* genannt.

Als reales Anwendungsszenario kann man sich vorstellen wie aus einem großen Stream mit Klick- und Viewevents jeweils ein Klick-Stream und ein Viewstream generiert wird.

(vgl. Hedtstück, 2017, Seite 28).

2.5. Verknüpfung paralleler Eventströme

Das Verknüpfen von paralleler Eventströmen lässt sich so erklären, dass Ereignisse aus zwei oder mehreren Teilströmen zu einem neuen Teilstrom von Ereignissen hinzugefügt werden.

Mögliche Paare von Events werden nach einem gemeinsamen Attribut selektiert. Es wird dann aus den selektierten Ereignissen der Ströme ein neues Ereignis erzeugt, das einen eigenen Zeitstempel, Ereignistyp und eine neue EreignisID bekommt. Die Menge der Events die für die Verknüpfung berücksichtigt werden, lassen sich über ein Auswertungsfenster eingrenzen. Optional können aus den zu ursprünglichen Ereignissen wahlweise Attribute für das neue Ereignis übernommen werden.

Zwei Streams die zu einem neuen Stream verknüpft werden, werden

in Abbildung 2.9 dargestellt. Das Attribut `UserID` steht dabei in Klammern. Es wird deutlich das sich die Events e_2 und e_a verknüpfen lassen, da sie innerhalb des Auswertungsfenster eine gemeinsame `UserID` haben nach der selektiert wird. Es wurde so ein neues Ereignis $e_{a/2}$ erzeugt, das in einem neuen laufenden Stream hinzugefügt wurde.

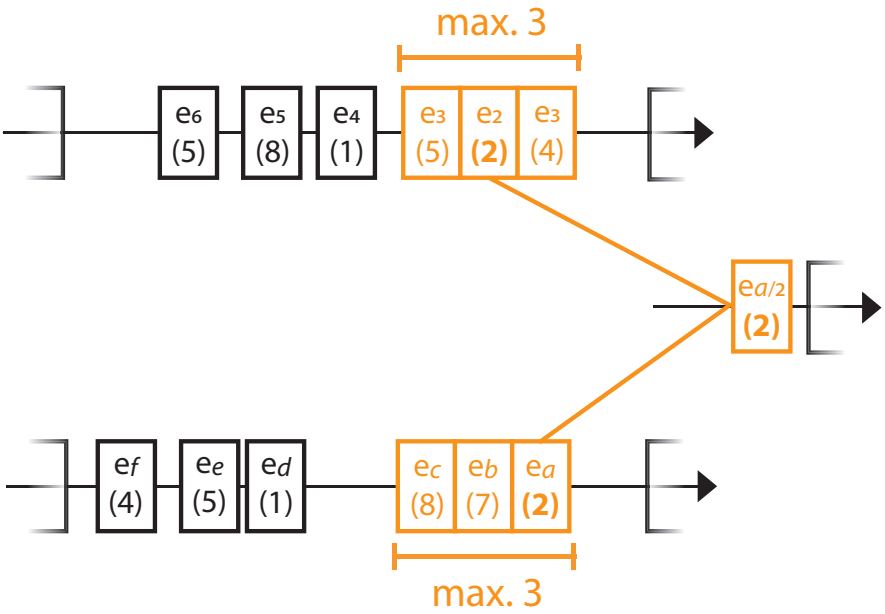

Abbildung 2.9.: Verknüpfung zweier Streams mit einem Auswertungsfenster der Größe 3

Dieses Verknüpfen von parallelen Streams ist der Fokus dieser Arbeit und soll anhand eines nahezu realen Anwendungsszenario evaluiert werden.

(vgl. Akidau, 2015a; Troßbach, 2017).

3. Anforderungsanalyse

Als erstes wird beschrieben welches Anforderungsszenario umgesetzt wird, um zu verstehen, was die Evaluierung bezwecken soll. Dafür wird auch geprüft welche Anforderungen dafür erfüllt sein müssen.

3.1. Anwendungsszenario

Als Anwendungsszenario wird ein Informations-Dashboard eines Online Shops simuliert. Auf diesem Online-Shop werden von den Benutzern (Ereignis-Produzenten) Klicks und Transaktionen als JSON (JavaScript Object Notation) erzeugt. Diese Daten sind zufällig generiert und werden in die CEP-Software geführt. Mit Hilfe von Zufallszahlen soll ein möglichst reales Verhalten einer Webplattform erzeugt werden. Die Daten die von den Nutzern einer Webplattform erzeugt werden, sind nämlich auch unvorhersehbar und unregelmäßig. Dafür werden beispielsweise zufällige Verzögerungen erzeugt. Weiterhin wird immer nur ein kleines Zeitfenster einer größeren Webanwendung simuliert. Daher bewegen sich die Zeiten die in dieser Arbeit benutzt wurden im Millisekunden Bereich. Es werden immer um die 5000 bis 7000 Events pro Sekunde über ein kleinen Zeitraum erzeugt. Diesen Durchsatz erwartet man in echten größeren Anwendungen durchgehend.

Das Dashboard hat mehrere Features die, mit Hilfe von komplexen Ereignisprozessen wie das Verknüpfen von Datenströmen realisiert werden. Optional kann dieses Dashboard eine Benutzeroberfläche haben (Ereignis-Konsument), welche die Ergebnisse visuell darstellt. Für das Anwendungsszenario werden die erzeugten Daten der Nutzer von der Software so vorbereitet das in Echtzeit angezeigt werden könnte, welche Produkte des Online-Shops aktuell am meisten gesucht und geklickt werden oder welche am seltensten. Durch die Vorbereitung der Daten könnte ein „Top-Produkte-Ranking“ oder „Flop-Produkte-Ranking“ umgesetzt werden. Dafür werden geeignete Auswertungsfenster und Zeitstempeltypen gesucht. Eine Erweiterung des Dashboards ist das Top-Ranking der geklickten Produkte nicht nur von einem Eventstrom (hier: Suchstream) abhängig zu machen, sondern zusätzlich jeweils von einer anderen Transaktion (zum Beispiel: Klick und Suche, Klick und Aufrufe, Klick und Bezahlungen).

Anwendungsfall	Bezeichnung
Use Case 1	Variable Verknüpfungs- und Auswertungsfenster
Use Case 2	Variable Eventgenerierung mit verschiedenen Zeitstempeltypen
Use Case 3	Effizentes One-To-Many-Join
Use Case 4	Verknüpfung von Stream und Änderungslog

Tabelle 3.1.: Die vier Anwendungsfälle und ihre Bezeichnung

Das Dashboard kann dann ein mehrere Top-Rankings gleichzeitig anbieten. Dafür wird eine effiziente Implementierung gesucht und evaluiert.

Weiterhin könnte die Software Betrugsversuche mit Hilfe von persistent hinterlegen Daten und einem Ereignisstream erkennen und anzeigen. Benutzer können mit Hilfe der `UserID` eindeutig identifiziert werden. Die Umsetzung einer konkreten Betrügererkennungssoftware ist nicht Teil dieser Arbeit und wird daher nicht umgesetzt, lediglich das Vorbereiten der Daten findet statt.

Auf diesem Szenario sollen vier Use Cases evaluiert werden. Diese lassen sich aus den Features des Dashboards ableiten. In der Tabelle **??** werden die einzelnen Anwendungsfälle beziffert und bezeichnet, auf denen sich in dieser Arbeit bezogen wird.

In dieser Arbeit wird die Weboberfläche als Anwendungsszenario nicht umgesetzt. Es geht nur um die Simulation und Verarbeitung der Ereignisse dahinter – also die Bereitstellung der verarbeiteten Daten.

Das Szenario sollte möglichst viele einzelne Quellen oder wenige Quellen mit sehr hoher Datenerzeugungsrate haben, um sehr viele verschiedene Daten zu generieren. Ein Anwendungsfall könnten Nutzer einer Applikation sein. Mehrere von Nutzer erzeugte Transaktionen eines Systems könnten dafür verwendet werden um sie mit Hilfe der CEP-Techniken zu verknüpfen und dann auszuwerten. Es können so die oben vorgestellten Features umgesetzt werden. Für eine sinnvolle Verwendung wird eine immense Datenmenge, persistent sowie nicht persistent, vorausgesetzt. Nur so können neue Rückschlüsse aus den Daten gezogen werden, die bei wirtschaftlichen Entscheidungen helfen.

Gerade das unvorhersehbare und unregelmäßige eintreffen von Ereignissen (Transaktionen) durch zufällig erzeugtes Nutzerverhalten machen die Evaluierung dieser Arbeit interessant. Die Wahl des richtigen Auswertungsfensters, Fenstergröße und Implementierung können die Effektivität der Verarbeitung solcher Anwendungen (hier das Dashboard) beeinflussen.

Das Szenario lässt sich also auf eine Vielzahl von Systemen anwenden, die auch diese Art Daten aus verschiedenen Quellen erzeugen. Es lässt sich aufgrund der unabhängigen CEP Komponentente, gut in bereits bestehende Systeme integrieren.

3.2. Rahmenbedingungen

Damit das oben beschriebene Anwendungsszenario realisiert werden kann, dass zum einen die Erzeugung der Zufallsevents und zum anderen die Verarbeitung und Analyse der Eventströme beinhaltet, benötigt man gewisse Anforderungen die in funktionale und nicht-funktionale Anforderungen zusammengefasst wurden.

3.2.1. Funktionale Anforderungen

Anforderung an die Eventerzeugung

Eine Anforderung an die Software ist das bereitstellen von mindestens zwei ungleichen Datenquellen in Form von Ereignisströmen, mit sehr vielen Ereignissen. Die Ereignisse müssen zu dem zufällig erzeugt werden. Das ist notwendig um ausreichend Daten zu haben und eine Evaluierung durch führen zu können. Diese Eventstreams müssen zu dem in einer sehr hohen Geschwindigkeit eintreffen, um der Echtzeit Verarbeitung möglichst nahe zu kommen. Damit eine spätere Auswertung möglich ist, ist es weiterhin notwendig das die Events innerhalb eines Streams zwischengespeichert werden und nicht sofort wieder gelöscht werden.

Anforderung an die Verarbeitung und Analyse

Eine weitere funktionale Anforderung ist, das sich Eventstreams verknüpfen lassen. Damit zwei oder mehr Eventstreams verknüpft werden können, ist es notwendig, dass die Streams aggregiert werden können. Die wichtigste Aggregationsfunktion die dazu notwendig ist, ist der `JOIN` den man auch aus den klassischen relationalen Datenbanken kennt. Ohne diese Anforderung wäre die Umsetzung des Szenarios nicht möglich. Das Verknüpfen erst bietet die Möglichkeiten Erkenntnisse aus mehreren einzelnen Quellen zu ziehen.

Die Streaming-Technologie muss konkret für die zu evaluierenden Anwendungswälle 1 und 2 auch die Möglichkeit bieten zwischen den Auswertungsfenstern (Tumbling Window/Time Window) und deren Größe sowie zwischen Zeitstempeltypen (Systemzeit oder Eventzeit) auswählen zu können. In einigen Fällen kann die Wahl des Auswertungsfenster gegebenenfalls zu schnelleren Ergebnissen führen.

Für den letzten Anwendungsfall wird zu dem das Speichern von Änderungen in einer persistenten Datenbank benötigt, die wiederum mit einem Stream aggregiert werden kann.

Die Anforderungen an die Stream Data Platform und das Stream Processing System sind folgende:

1. Verarbeitet Daten bei ihrem Eintreffen (möglichst in Echtzeit)

2. Lässt sich über Cluster skalieren
3. Speichert Daten mit gewisser Fehlertoleranz
4. Einfache, also möglichst wenig aufwendige Integration in bestehende Systeme

Die Verarbeitung in nahe zu Echtzeit ist notwendig, damit die aufkommenden Events sich sonst zurück stauen würden und so potentielle Kaufentscheidungen oder Betrüger zu spät erkannt werden würden. Eine Verarbeitung von Daten mit Verzögerungen ist für diese Arbeit uninteressant, da heutige Datenquellen immer höhere Datenmengen in kürzester Zeit generieren, die es zu verarbeiten gilt. Ab einer gewissen Datenmenge kann ein einzelnes Rechen System diese nicht mehr in der geforderten Zeit verarbeiten. Einer vertikale Skalierung (mehr Speicher, leistungsfähigere CPU) sind ökonomische und technische Grenzen gesetzt. Aus diesem Grund muss das System horizontal skalierbar (auf mehreren Rechen Systemen) sein. Viele heutige Systeme sind effektiv weil sie verteilt auf mehreren Rechnern in einem Cluster laufen und von daher beliebig horizontal skaliert werden können um höheren Anforderungen gerecht zu werden.

Die Geschwindigkeit der Verarbeitung gegenüber der Fehlertoleranz ist ein Kriterium, da die Priorisierung auf den Durchsatz der Daten liegt, nicht auf die vollständige Korrektheit. Bei der großen Menge der Daten mit denen gearbeitet wird, spielen einige Verluste keine entscheidende Rolle.

Aus Gründen der guten Wartbarkeit und Erweiterbarkeit sollte das System einfach integrierbar sein. Solche Systeme lassen sich gut austauschen und laufen dann als unabhängige Komponente. Bei einem Systemausfall, ist nur diese Komponente betroffen, nicht das ganze System.

3.2.2. Nicht-funktionale Anforderungen

Eine nicht-funktionale Anforderung ist die Skalierbarkeit des Systems. Gerade durch schnell anwachsende Nutzerzahlen, die gleichzeitig die Datenquellen darstellen, muss das System gut skalierbar sein um den erhöhten Datenaufkommen bewältigen zu können.

Außerdem sollten die Antwortzeiten möglichst klein sein. Die Effizienz ist daher eine entschiedene Anforderung. Es hat keinen Nutzen die Software zu verwenden, wenn die verarbeiteten Ereignisse, die zur Entscheidungshilfe dienen sollen, zu spät zurück gegeben werden.

Weiterhin sollte die Software anpassbar sein, um mögliche Erweiterungen nachträglich umzusetzen die für ein neues Szenario interessant sein könnte.

4. Design

In diesem Kapitel wird zu erst begründet weshalb sich für *Apache Kafka* und die API Kafka Streams entschieden wurde. Diese Technologien werden dann näher erläutert. Darauffolgend wird die Architektur der Software mit seinen verschiedenen Sichten vorgestellt. Dafür werden verschiedene UML-Diagramme verwendet. Es wird beschrieben wie die Tests aufgebaut sind um eine Evaluierung möglich zu machen und wie die Events konkret dafür aussehen.

4.1. Auswahl von Technologien

Unabhängig von der Wahl der Plattform, muss die Wahl einer geeigneten Programmiersprache getroffen werden. In dieser Arbeit wird die Programmiersprache Java verwendet, da sie weit Verbreitung findet und sich gut mit der gewählten Technologie verwenden lässt. Die Eventerzeugung liese sich auch in einer anderen Programmiersprache umsetzen.

4.1.1. Stream Data Platform

Eine geeignete Plattform, welche die Bedingungen erfüllt ist *Apache Kafka* in Verbindung mit dem Stream Processing System Kafka Streams. Mögliche Alternativen, die aber viel langsamer als *Apache Kafka* sind, wären RabbitMQ oder HornetQ. *Apache Kafka* ist ein Messaging-System das ursprünglich bei dem Internetportal LinkedIn entwickelt wurde, aber seit 2011 ein Software-Projekt der Apache Software Foundation ist.

Implementiert ist dieses verteilte System in Scala, eine funktionale Java Virtual Machine (JVM) Sprache, lässt sich jedoch über eine Java API (Application Programming Interface) auch innerhalb einer Java-Applikation verwenden. In dieser Arbeit wird die Java-Schnittstelle verwendet um das Anwendungsszenario umzusetzen. Da sowohl Scala als auch Java in Java-Bytecode umgewandelt werden, gibt es keinen Overhead beim Aufruf von der einen in die andere Sprache.

Apache Kafka zeichnet sich neben der Verteilbarkeit, vor allem durch die hohe Skalierbarkeit und den hohen Durchsatz von Daten aus. Denn im wesentlichen stellt *Apache Kafka* eine persistente Queue für Messages wie Logs bereit, die mit sehr hoher Geschwindigkeit gelesen

und verarbeitet werden können. Zu den Logs zählen auch Benutzeraktionen wie Seitenbesuche, Warenkorbtransaktionen oder Klicks. (vgl. Ghadir, 2013). Gespeichert werden diese Daten in Key-Value-Datenbanken – die schnellste aller NoSQL-Datenbanktypen. Ein Eintrag dieser Datenbank, also ein Ereignis, besteht dann aus einem Schlüssel (engl. *key*) und dazu gehörigem Wert (engl. *value*), wobei der Wert die Eigenschaften des Ereignis enthält.

Die Daten der Events werden als Binäre Daten angelegt, übertragen und verarbeitet. JSON ist ein einfaches textbasiertes Format von binären Daten das den Vorteil mit sich bringt, Attribute und Attributwerte strukturiert abzulegen. Das ermöglicht das bessere Lesen und damit Testen der Anwendung. Der wichtige Vorteil von JSON gegenüber XML ist die kompakte Syntax die eine schnellere Übertragung ermöglicht. Die Byterepräsentation wäre noch schneller, aber schlechter lesbar.

Apache Kafka ist ein verteiltes Log, das als Publish/Subscribe Messaging System genutzt werden kann. Diese Funktion übernehmen klassisch spezialisierte Message Queue (MQ) Systeme, wie beispielsweise RabbitMQ, ActiveMQ etc. Die Hauptunterschiede liegen in der horizontalen Skalierbarkeit für große Datenmengen. Beispielsweise betreibt Microsoft Office einen Kafka Cluster mit 1300 Brokern, der 1 Trillion Nachrichten pro Tag verarbeitet (siehe Ritchie, 2016, Slide 12). Des weiteren ist Apache Kafka im Gegensatz zu oben genannten Technologien ein persistentes Log, Nachrichten bleiben so lange erhalten, wie die „retention time“ das vorgibt und können erneut konsumiert werden. Klassische MQ-Systeme sind dafür nicht vorgesehen, eine Nachricht wird aus der Queue entfernt, sobald der letzte Subscriber sie abgeholt hat.

In dem *Publish-Subscribe*-Messaging werden die Nachrichten von einem *Sender* (Producer) zu einem Topic geschickt und können von mehreren *Empfängern* (Consumer) aus diesem Topic gelesen (subscribed) werden. So werden die *Empfänger* informiert, sobald eine neue Nachricht (hier: Ereignis) vorliegt. Die Nachricht wird dann konsumiert und nach nach Bedarf aus der Queue entfernt.

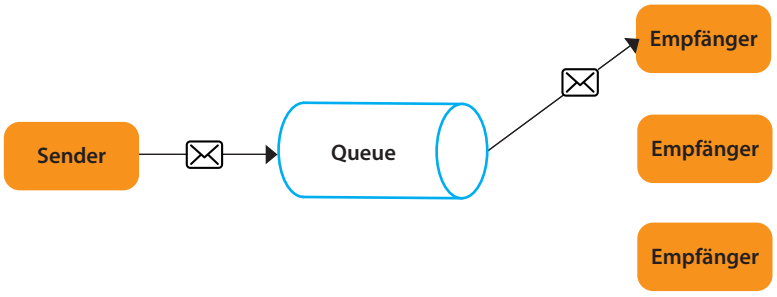

Abbildung 4.1.: Prinzip des Publish-Subscribe-Messaging.

Diese Technologie ist für viele Daten schneller als seine Alternativen. Das liegt unter anderem daran, dass jeder Konsumer für sich selber die Nummer der zuletzt gelesenen Nachricht verwaltet – nicht der Server. Für geringere Datenmengen ist es langsamer als andere Systeme. Der Performance Unterschied wird an Hand der Tabelle 4.1 mit den Messergebnissen von Quadri deutlich. (vgl. Quadri, 2016).

Technologie	Senden (msg/s)	Empfangen (msg/s)
Apache Kafka	33000	31000
HornertQ	17000	16000
SQS-1node	14000	4000
RabbitMQ-batch100	3000	3000

Tabelle 4.1.: Vergleich von Senden und Empfangen von Nachrichten pro Sekunde.

Es gibt auch keine klassische relationale Datenbank die für das Anwendungsszenario in Frage kommt. Diese Datenbanken sind aufgrund ihrer AKID Eigenschaften, also Atomarität, Konsistenz, Isolation und Dauerhaftigkeit viel zu langsam und kommen mit den in dieser Arbeit verwendeten Datenmengen nicht zurecht. Dagegen erfüllt *Apache Kafka* die drei Herrausforderungen des *Big Data* optimal (siehe 2.1, 3V-Modell) und arbeitet als ein Verteiltes System nach BASE (Basically Available, Soft state, Eventual consistency). Danach wird keine Konsistenzgarantie gegeben, nur das ein Datensatz irgendwann konsistent werden kann. *Apache Kafka* kann aufgrund seiner Skalierbarkeit mit Billionen (*volume*) unterschiedlichster Events (*variety*) pro Tag arbeiten und dabei eine sehr hohe Geschwindigkeit garantieren (*velocity*). (vgl. A7, 2017; Reed and Johnson, 2015).

4.1.2. Stream Processing System

Die Wahl des Stream Processing Systems fällt auf Kafka Streams. Zu den alternativen Stream Processing Bibliotheken gehören unter anderem Apache Flink und Apache Spark die hier näher beleuchtet werden, um die Wahl der Kafka Streams Bibliothek zu begründen. Dafür soll nun ein Vergleich zwischen Apache Flink, Apache Spark und Kafka Streams hergestellt werden, um die Vor- und Nachteile der einzelnen Stream Processing Technologien erkennbar zu machen.

(vgl. A7, 2017; Kreps, 2016).

Apache Spark Spark wird hauptsächlich für das Verarbeiten von Daten, die als Dateien vorliegen und im Arbeitsspeicher in Batches abgearbeitet werden (RDD-Datenstruktur), verwendet. Doch durch das Aufteilen in kleinere Stapel und anschließendem Sammeln der Daten

eines gewissen Zeitraumes, wird das Stream Processing ermöglicht. Spark wird oft für Machine Learning benutzt, weil die Lernalgorithmen des Machine Learnings meistens iterativ sind wofür Spark entwickelt wurde. Weiterhin unterstützt Spark die *MLlib* Bibliothek um typische Data Science Methoden anzuwenden.

Neben Tumbling Windows unterstützt Spark auch Sliding Windows. Session Windows werden aktuell nicht unterstützt. Entscheidend ist aber die fehlende Unterstützung der Verwendung von der Systemzeit für die Eventverarbeitung. Spark an sich unterstützt nur die Eventzeit. Zwar unterstützt die Weiterentwicklung Structured-Spark die Verwendung von Systemzeit, das wird aber über noch kleinere Stapeldateien realisiert, die immer noch minimal zwischengespeichert werden - weshalb keine reale Systemzeit erreicht wird. Außerdem befindet sich Spark Structured Steaming zum Zeitpunkt des Schreibens dieser Arbeit in einer Alpha Version.

Apache Flink Flink ist ein sehr mächtiges Framework für verteiltes Stream Processing. Das Framework kann sich gut von Fehlern erholen und damit den Anwendungszustand beibehalten. Es kann auf tausenden Knoten mit einem guten Durchsatz ausgeführt werden, das zu einer hohen Geschwindigkeit der Verarbeitung sorgt. Diese geringe Latenz ermöglicht Flink, im Gegensatz zu Spark, mit Systemzeit arbeiten zu können.

Flinks Datenstreams sind echte Streams. Das bedeutet die Elemente werden sofort in die zu verarbeitende Kette gehängt, sobald sie das Programm erreichen. Es können auch verspätete Daten aus alten Fenstern, die außerhalb des aktuellen Auswertungsfenster liegen, verarbeitet werden. Diesen Mechanismus nennt man *watermarks*. Mit einer Heuristik wird versucht, Zugriff auf alte Daten zu geben, ohne das aktuelle Auswertungsfenster zu verzögern.

Das erlaubt flexible Auswertungsfenster auf die Streams. Flink unterstützt die Fensterarten Tumbling-Window, Sliding-Window und Session-Windows.

Trotz seiner Vorteile ist Apache Flink gegenüber *Apache Kafka* eine Zusatztechnologie die aufwendig integriert werden muss. Das heißt es müssen die Flink Cluster selber angelegt werden und die Intergation mit *Apache Kafka* muss hergestellt werden.

Kafka Streams Apache Kafka bietet das voll-integrierbares Framework Kafka Streams an, das auch nicht-triviale Verarbeitungen wie das Aggregieren und Verknüpfen von Streams ermöglicht auszuführen. Über die Java-Schnittstelle lässt sich mit *Apache Kafka* interagieren.

Das Framework löst viele Stream Verarbeitungsprobleme wie das handhaben von Daten die außerhalb des zu betrachenden Fensters liegen, falls mögliche Latenzen eine Verzögerung verursachen. Das ist auch entscheidend für die Evaluierung in dieser Arbeit. Die Consumer und Producer API des Kafka Clients ermöglichen das Entwickeln von unabhängigen Ereig-

nisproduzenten und Konsumenten, die in dieser Arbeit genutzt werden. Die API von Kafka Streams ermöglicht das Verarbeiten von Streams.

Die Daten des verteilten Logs werden auf den Brokern im Dateisystem persistiert. Damit können nicht nur Daten des aktuell betroffenen Auswertungsfenster verarbeitet werden, sondern auch nachträglich Daten aus älteren Fenstern – der bereits oben beschriebene Mechanismus Watermarks. Noch eintreffende Events werden dagegen direkt verarbeitet. Es werden also beide Datenarten kombiniert, um einen Stream zu schaffen der mit geringer Latenz und hoher Speicherzuverlässigkeit arbeitet.

Das Framework Kafka Streams ermöglicht mit seiner API das Verwenden von Event- und Systemzeit. Auswertungsfenster können Tumbling-Windows, Sliding-Windows und Session-Windows sein.

(vgl. Warski, 2016; A7, 2017; Kreps, 2016).

Zusammenfassend liegen die Vorteile von Apache Kafka Streams gegenüber den anderen Stream Processing Systemen in der Skalierbarkeit und Performance. Andere mögliche Technologien wie RabbitMQ oder relationale Datenbanken sind zu langsam oder können aufgrund ihrer Eigenschaften nicht mit den riesigen Datenmengen umgehen. *Apache Kafka* passt ideal zu dem beschriebenen Anwendungsszenario (siehe 3.1).

Bei der Wahl eines geeigneten Stream Processing Systems gab es mehrere mögliche Alternativen. Apache Spark fällt aus der Wahl raus, da die benötigte Unterstützung von Systemzeit-Verarbeitung fehlt. Apache Flink sowie Kafka Streams selbst bringen alle Anforderungen mit. Apache Flink benötigt aber eine zusätzliche aufwendige Integration in *Apache Kafka* und ist für das Anwendungsszenario zu umfangreich. Kafka Streams hingegen ist eine einfach zu verwendende Bibliothek und läuft mit *Apache Kafka* selbst (ohne andere externe Abhängigkeiten).

In der Tabelle 4.2 werden nochmal die wichtigsten Eigenschaften für das Anwendungsszenario zusammen getragen und ob sie erfüllt sind.

Nach dieser Begründung weshalb *Apache Kafka* in Verbindung mit Kafka Streams genutzt wird, soll im nächsten Kapitel unter anderem diese Technologie näher vorgestellt werden.

	Apache Spark	Apache Flink	Kafka Streams
Auswertungsfenster			
Sliding Window	x	x	x
Tumbling Window	x	x	x
Session Window		x	x
Systemzeit vs. Eventzeit			
Eventzeit	x	x	x
Systemzeit		x	x
Offline-Daten Behandlung			
Watermarks		x	x
Einfache Integration in Apache Kafka			
Erfüllt			x

Tabelle 4.2.: Übersicht der Features von Flink, Spark und Kafka Streams.

4.2. Apache Kafka

Die Architektur von *Apache Kafka* besteht aus *Topics*, *Producers*, *Consumers*, und *Brokers*. Alle Nachrichten, also die Events, befinden sich in Partitionen die wiederum in den Brokern liegen. Um eine Nachricht zu lesen oder zu schicken, muss immer das jeweilige Topic einer Partition angegeben werden.

Apache Kafka speichert die Partitionen (inklusive Nachrichten) auf den Brokern in einem eigenen, binären Dateiformat. Für das Zwischenspeichern der Nachrichten wird von Kafka Streams eine *Key-Value-Datenbank* verwendet. Die Einträge dieser Datenbanken bestehen aus einem Schlüssel und einem dazu gehörigen Wert. Der Zugriff auf die Werte geschieht also über den Schlüssel. Diese Datenbank legt mehrere sogenannte *SST* (*Static Sorted Table* Dateien an. Die Einträge sind statisch und sortiert und ermöglichen einen schnellen Zugriff auf die Werte. Dateien werden nach Gebrauch wieder entfernt. So wird ständig Speicherplatz wieder freigegeben und nicht dauerhaft belegt.

Die Koordination zwischen Producer, dem Kafka Cluster und Consumer regelt der *Apache Zookeeper*. Das Prinzip von *Apache Kafka* wird an der Abbildung 4.2 deutlich.

Diese Stream Data Platform verteilt auf einem Cluster arbeiten. Dafür wird der oben erwähnte *Zookeeper* benötigt. Der *Apache Zookeeper* ist ein eigenes Projekt von Apache. Dieser bietet das verteilte Konfiguieren und Synchronisieren, sowie ein Namensregister für die einzelnen Knoten des verteilten Systems an. Es gibt also mehrere Knotenpunkte in denen diese Topics liegen können. Diese Knoten heißen *Broker*.

(vgl. Sookocheff, 2015; Jain, 2013; Foundation, 2017).

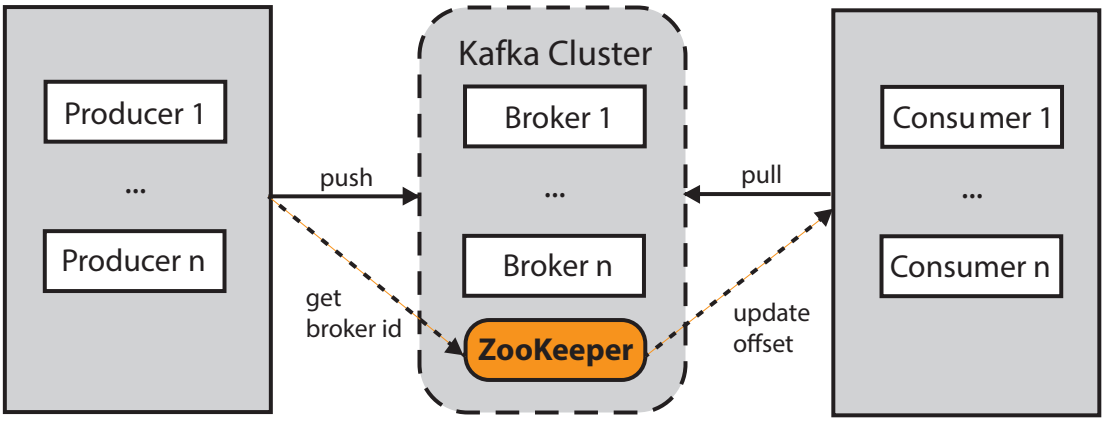

Abbildung 4.2.: Prinzip von Apache Kafka

4.2.1. Broker

Apache Kafka kann als verteiltes System mehrere Broker besitzen. Sie bilden zusammen ein *Apache Kafka* Cluster.

Jeder Broker hat dann eine gewisse Anzahl an Partitionen. Jeder dieser Partitionen kann entweder ein *Leader* oder eine *Kopie* für ein Topic sein. Wird eine Partition angelegt, wird sie auf die anderen Broker kopiert. Sobald eine Partition verändert wurde, überschreibt der Leader seine Replikationen mit den neuen Daten – die Daten werden auf den Knoten synchronisiert. Falls das nicht klappt, wird einer der Kopien als neuer Leader ausgesucht. All das passiert durch *Apache Kafka* automatisch.

Aufgrund dieser Replikation und der strengen Reihenfolgeeinhaltung kann *Apache Kafka* die Datenkosistenz und Verfügbarkeit garantieren. Dabei wird das komplexe Nachrichtenrouting und großartige Serververwaltungen vernachlässigt, was den Datendurchsatz erhöht. Eine ausführliche Begründung ist nicht Fokus dieser Arbeit, kann aber in der Quelle (Sookocheff, 2015) nach gelesen werden.

In der Abbildung 4.3 wird das Prinzip der Leadparition (deutsch: Leiterpartition) und Kopien nochmal deutlich. In diesem Beispiel besitzt der Broker 1 die zwei Leadpartition für 1 und 2 und Broker 2 hat den Leader für Partition 3.

(vgl. Sookocheff, 2015).

4.2.2. Topics und Partitions

Ein Topic von *Apache Kafka* liegt in einer oder mehreren *partitions*. In diesen Partitionen liegen die Nachrichten (hier die Ereignisse). Diese Daten der Partition lassen sich über ein

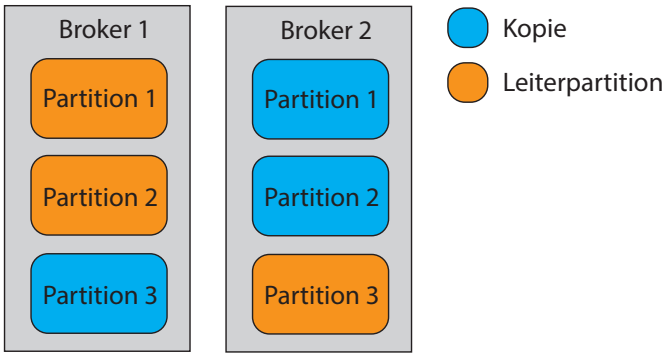

Abbildung 4.3.: Partionen in zwei Brokern mit Leadpartition und Kopien.

gemeinsames Topic, das über mehrere Broker verteilt ist, aufteilen um mehreren Consumern das parallele lesen zu ermöglichen. So gibt es beispielsweise die Partition 1 eines Topics auf mehreren Geräten verteilt (siehe Abbildung 4.3). Auch Consumer können parallelisiert werden. Dadurch können mehrere Consumer von mehreren Partitionen lesen was für den schnellen Durchsatz von *Apache Kafka* sorgt.

Jede Nachricht in einer Partition hat dann noch einen *offset*. Das ist eine automatisch fortlaufende Nummer die jede Nachricht in einer Sequenz der Partition eindeutig zu ordnen lässt und eine Reihenfolge vorgibt. Das Offset erlaubt es Consumern Nachrichten ab einem bestimmten Punkt der Sequenz zu lesen.

Jede Nachricht kann eindeutig über das Tupel $Message(Topic, Partition, Offset)$ identifiziert werden. Eine Nachricht selbst besteht aus einem Schlüssel, einem dazu gehörigen Wert und einem Zeitstempel.

(vgl. Sookocheff, 2015).

4.2.3. Producer und Consumer

Wenn ein Producer ein Topic beschreiben möchte, muss der Erzeuger die Nachrichten an die Leadpartition schicken. Dieser repliziert dann die Daten auf seine Kopien in die anderen Broker. Das ganze wird nochmal durch die Abbildung 4.4 deutlich. In diesem Fall gibt es drei Broker. Ein Producer (Kafka Client) schreibt nun eine Nachricht in die Leadparitition 1 des Brokers 1. Diese Partition repliziert dann seinen Inhalt auf seine Kopien.

(vgl. Sookocheff, 2015).

Ein Consumer kann nun die Nachrichten jeder Partition eines Brokers lesen. Also können auf Grund der Kopien mehrere Consumer performant über die Broker verteilt gleichzeitig die Nachrichten lesen. Mehrere Consumer können auch von nur einem Broker lesen, das ist aber

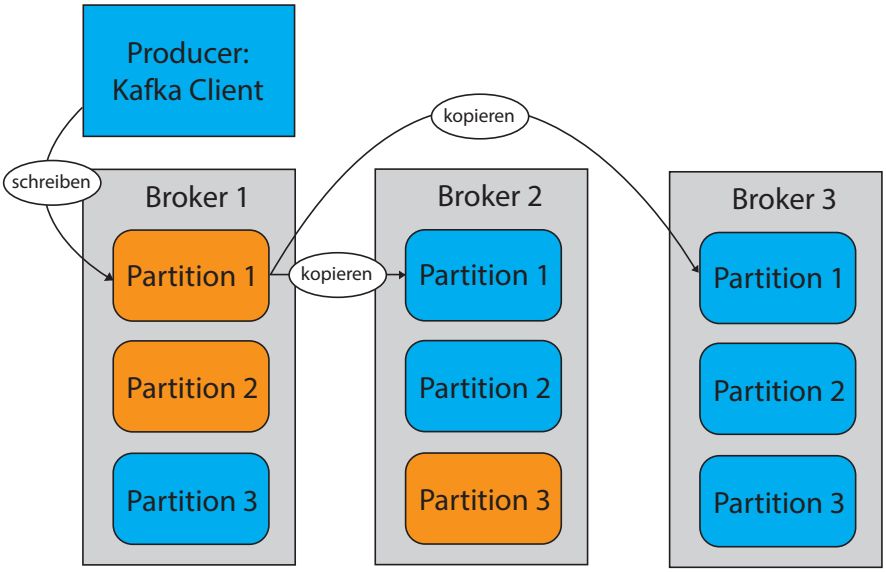

Abbildung 4.4.: Ein Schreibvorgang und die Replikation der Partitionen.

nicht so performant, da kein gleichzeitiger sondern nur sequentieller Zugriff der Partitionen möglich ist. Zu dem können mehrere Consumer in Gruppen eingeteilt werden. Gruppiert wird bei *Apache Kafka* nach einem Topic. Es werden dann alle Nachrichten der Partionen des Topics gelesen und zusammen getragen.

Der Idealfall dabei ist, dass ein Consumer für eine Partition innerhalb der Consumer Gruppe verwendet wird. Sollte es mehr Consumer als Partitionen für eine Gruppe geben, warten die nicht genutzten Consumer bis sie benötigt werden. Gibt es zu wenig Consumer, müssen diese mehrere Partitionen nach einander lesen.

(vgl. Verma, 2015; Jain, 2013).

4.3. Verwendete Klassen der Schnittstellen

Kafka Streams ist ein in Scala geschriebenes Framework um mit *Apache Kafka* zu interagieren. Das Framework lässt sich auch mit Java verwenden, so wie es in dieser Arbeit getan wurde.

Die wichtigsten Klassen der Kafka Client und Kafka Streams API die für diese Arbeit verwendet wurden, sollen hier kurz vorgestellt werden:

4.3.1. Kafka Client API

KafkaProducer Ein Kafka Client das `ProducerRecords` zu einem Kafka Cluster führt. Der Producer ist *thread safe*, kann also problemlos mehrfach ausgeführt werden.

ProducerRecord Ein Eintrag das aus einem Schlüssel-Wert-Paar besteht. Es beinhaltet den Namen des Zielstopics und optional die Partitionsnummer in dass das Pärchen geschrieben werden soll.

KafkaConsumer Ist ein Kafka Client, der die Nachrichten eines Topics aus einem Kafka Clusters lesen kann.

4.3.2. Kafka Streams API

KStream Eine Abstraktion eines Strom von Schlüssel-Wert-Paaren. Diese Ströme können aus einem oder mehreren Topics bestehen. `KStream` lässt sich mit einem anderen `KStream` oder einer `KTable` verknüpfen um ein neuen `KStream` zu bilden.

KTable Eine Abstraktion eines Änderungsprotokollstream. Beinhaltet alle Änderungen von Einträgen. Kann auch aus mehreren Topics bestehen. Mit einem `KStream` kann es zu einer `KTable` aggregieren.

KStreamBuilder Baut die angegebenen Streams auf und startet sie.

KafkaStreams Erlaubt eine durchgehende Verarbeitung eines oder mehrerer Streams und sendet das Ergebnis zu einen oder mehreren Topics.

TimestampExtractor Wird benötigt um aus einem Event einen Zeitstempel für die Verarbeitung von *Apache Kafka* extrahieren zu können. Ansonsten würde *Apache Kafka* mit der Systemzeit arbeiten.

Alle Klassen des Frameworks und deren Funktion kann man in der Dokumentation von Kafka Streams (hier Version 0.10.2.1) nachlesen: `https://kafka.apache.org/0102/javadoc/`.

```
{
   "head":{
      "id": identifier,
      "type": eventtype,
      "timestamp": timestamp,
      "timestamp_ms": timestamp_ms
   },
   "http":{
      "name": value (optional)
      ...
   }
}
```

Listing 4.1: Genereller Aufbau eines Events in JSON

4.4. Aufbau eines Events

Wie bereits im Kapitel über die Technologiewahl beschrieben (siehe 4.1), wird JSON benutzt um die Ereignisse mit Eigenschaften zu füllen. Dieses Format ist auch üblich um Webdaten über eine REST Schnittstelle zwischen einzelnen Komponenten auszutauschen.

In dieser Arbeit werden die Parameter, die normalerweise von dem Netzwerkprotkoll benötigt werden, vernachlässigt, da die Webtransaktionen simuliert werden und deswegen keine Bedeutung für unser Programmablauf haben. Ein Event beinhaltet also nur noch die wichtigen CEP-Eigenschaften, die im Kapitel 2.3.1 vorgestellt wurden, sowie optionale anwendungsbezogene Parameter für spätere Auswertungen. Diese kompakte Version sorgt zusätzlich für ein einfacheres Verständnis.

Der grundlegende Aufbau eines Events lässt sich aus dem Listing 4.1 ableiten. Die hervorgehobenen Wörter stehen nur für Platzhalter und können durch sinngemäße Werte ersetzt werden.

Dabei steht die `transactionID` für die Identifikationsnummer der jeweiligen Transaktion. Weiterhin ist `timestamp` der Zeitstempel des Events und `timestamp_ms` der gleiche Zeitstempel nochmal in Millisekunden angegeben. Das wird spätere Rechnungen vereinfachen. Der Zeitstempel der CEP-Maschine wird intern in *Apache Kafka* hinterlegt. Die Attribute unter `http` sind optional.

Wie bereits aus dem Klassendiagramm entnommen, werden mit Hilfe der optionalen Parameter Daten der verschiedenen Eventtypen generiert. Dazu zählen in dieser Software: `ClickEvent`, `PayEvent`, `SearchEvent`, `UserEvent` und `ViewEvent`.

4.5. Architektur der Software

Nach dem die Funktionsweise von *Apache Kafka* erklärt ist und die wichtigsten Klassen aus dem Kafka Streams Framework für die Software vorgestellt wurden, folgt nun der Aufbau der Software.

4.5.1. Bausteinsicht

Die Architektur der Software die allgemein für die Use Cases gilt, wurden im Komponentendiagramm 4.5 zusammengefasst. Es zeigt das Gesamtsystem der Anwendung mit seinen Software-Bausteinen (auf Ebene 1). Den Systemkontext (Ebene 0) lässt sich aus im Anhang mit der Abbildung A.2 ablesen. Mehrere (simulierte) Nutzer interagieren mit dem System und erzeugen so die Daten.

Die Komponenten lassen sich auf einem Rechner verteilt ausführen oder verteilt auf verschiedenen Rechner. In dieser Arbeit wurde Ersteres gemacht.

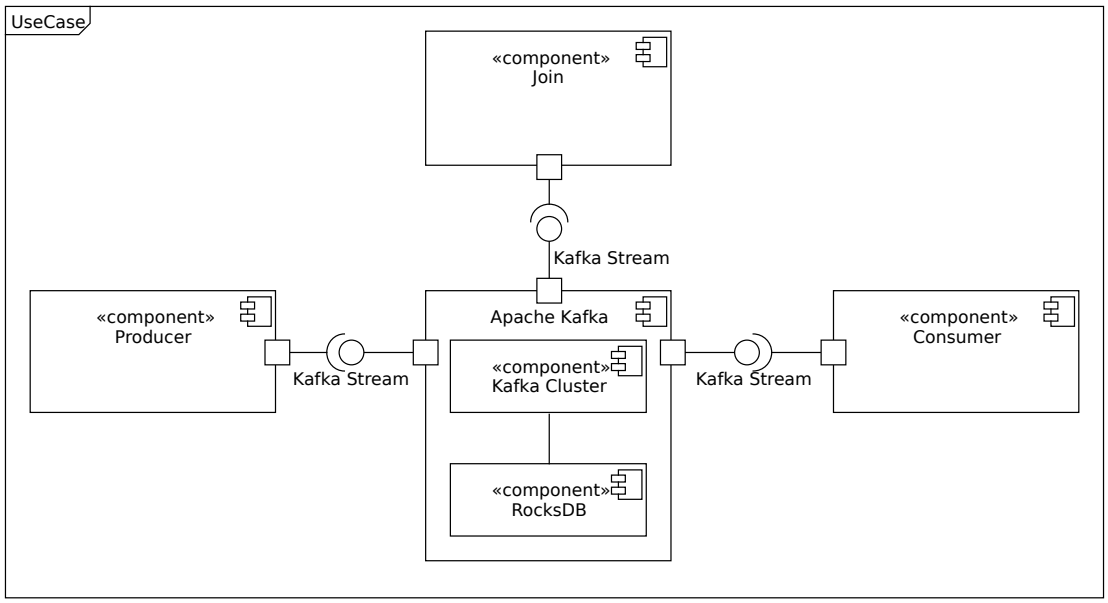

Abbildung 4.5.: Komponentendiagramm für die Anwendungssimultationen

Jeder der vier UseCases besteht aus den Softwarekomponenten `Producer`, `Join` und `Consumer`. Die Aufgaben der einzelnen Komponenten werden jetzt kurz erläutert. Im Kapitel 4.5.3 wird auf den Ablauf der Komponenten untereinander eingegangen. Im Anhang findet man die Abbildung A.2, die eine detailierte Sicht der Bausteinsicht (Ebene 2) mit den jeweils verwendeteten Klassen zeigt.

Producer Produziert die simulierten Events für jeden Stream und fügt sie jeweils dem Topic des Kafka Clusters hinzu.

Join Führt die Verknüpfung und Aggregation von zwei oder mehreren Eventströmen aus dem Kafka Cluster durch und führt das Ergebnis in einen neuen Stream (in ein neues Ergebnistopic).

Consumer Liest die verknüpften und aggregierten Events aus den Ergebnistopics und führt sie in eine externe Datei um sie später auswerten zu können.

Bei der Wahl der Architektur verfolgt die Software die Prinzipien der Abstraktion, Kapselung, Einheitlichkeit und Modularisierung. Die Komponenten sind so gewählt das sie unabhängig von einander funktionieren und durch andere Implementationen von Komponenten ersetzt werden können. Aufgrund seiner Kapselung in drei Komponenten hat jeder Teil der Software eine feste Aufgabe für das gesamte System:

1. das generieren der simulierten Events,
2. die Verarbeitung,
3. und die anschließende Auswertung.

Auch die Subkomponente RocksDB könnte bei Kafka Streams durch beliebige andere Key-Value-Datenbanken ausgetauscht werden, was wiederum die Abhängigkeiten reduziert. Die Architektur lässt sich auf beliebige simulierte oder echte Szenarios, bei denen eine Verknüpfung von Daten statt finden soll, übertragen. So lassen sich leicht ähnliche Probleme lösen.

Diese drei Softwarekomponenten laufen unabhängig von einander und können deswegen auch verteilt gestartet werden. Dann wäre die Anwendung verteilt. Alle Komponenten laufen dabei durchgehend – also ohne das sie von selber terminieren. Bei Bedarf können so bei permanent Events generiert, verknüpft und konsumiert werden.

Die Komponenten aggregieren mit Hilfe des Frameworks Kafka Streams (siehe 4.3) mit Apache Kafka. Das heißt *Apache Kafka* bietet dieses Schnittstelle an, damit andere Anwendungen mit dem Kafka Cluster arbeiten können.

Apache Kafka selber verwaltet mit Hilfe des *Apache Zookeepers* das Cluster.

4.5.2. Klassendiagramm

Der genaue Aufbau der Software lässt sich am Klassendiagramm Abbildung A.1 erkennen.

Auch wenn die Softwarekomponenten unabhängig voneinander sind, verwenden sie oft gleiche Klassen, weswegen sie auch im Klassendiagramm der Software auftauchen. Die Eventklassen sowie Use Cases können beliebig erweitert werden.

Jede Komponente besitzt einen `Manager`. Der Manager beinhaltet die notwendigen Serialisierer für das JSON Format und die Einstellungen für Apache Kafka. Der `Manager` baut einen Stream für ein gegebenes Topic. Es können weiterhin Topics aufgelistet werden oder die Anzahl der Einträge eines Topics ausgegeben werden. Eine weitere wichtige Aufgabe des Manager ist das exportieren des Inhalts eines Topics in eine CSV-Datei.

Um die globale Veränderung des Programms flexibler zu machen, verwenden alle Softwarekomponenten die `Constants` Klasse die Konstanten für Topicnamen, Formatangaben und Anzahl der simulierten Nutzer beinhaltet.

Jedes konkretes Event erbt von der Klasse `Event`. Die `Event` Klasse implementiert das Runnable Interface um mit Hilfe von Threads möglichst viele Events zu generieren. Jedes Event beinhaltet neben seinem Zieltopic und seiner eindeutigen, aber zufälligen Transaktionsnummer noch Eigenschaften zur simulierten System- und Eventzeit. So kann jedes Event zeitlich individuell erzeugt und verzögert oder ohne Verzögerung in den Topicstream geschrieben werden. Dafür gibt es eine minimale und maximale Millisekunden-Zahl die zufällig in diesem Intervall gewählt wird.

Threads und ein Zufallsintervall von Millisekunden sind hier notwendig um ein möglichst reales Szenario zu simulieren das eine Anwendung mit vielen parallelen Benutzertransaktionen voraussetzt.

Die konkreten Events `ClickEvent`, `PayEvent`, `SearchEvent`, `UserEvent` und `ViewEvent` generieren ihr individuelles JSON und können dem entsprechend beliebig erweitert werden.

Die `EventCounter` Klasse ist eine Hilfsklasse mit einem Eventzähler der immer zu einem bestimmten Zeitpunkt zurück auf Null gesetzt wird. Der Eventcounter wird von jedem Event hochgezählt, sobald sie zum Stream hinzugefügt wurden. Das Zeitpunkt-genaue Zurücksetzen geschieht mit Hilfe der `TimerTask` Klasse, die auch das Runnable Interface implementiert hat – und auch parallel läuft. Der Takt für das Zurücksetzen liegt bei dem `EventCounter` bei einer Sekunde. So können die Events pro Sekunde, also der Durchsatz, aufgezeigt werden.

Use Case 4 benötigt noch zwei weitere Klassen `User` und `Product` um das Szenario hierfür umzusetzen. Beim Use Case 2 wird zu dem noch ein `TimestamptExtractor` benötigt, der bei Bedarf für die Verarbeitung den Zeitstempel aus dem Event verwendet.

4.5.3. Laufzeitsicht

Es wird nun kurz der grundsätzliche Ablauf der Komponenten näher erläutert, die für alle Use Cases gelten. Den Ablauf der einzelnen Komponenten untereinander lässt sich aus der Abbildung 4.6 entnehmen.

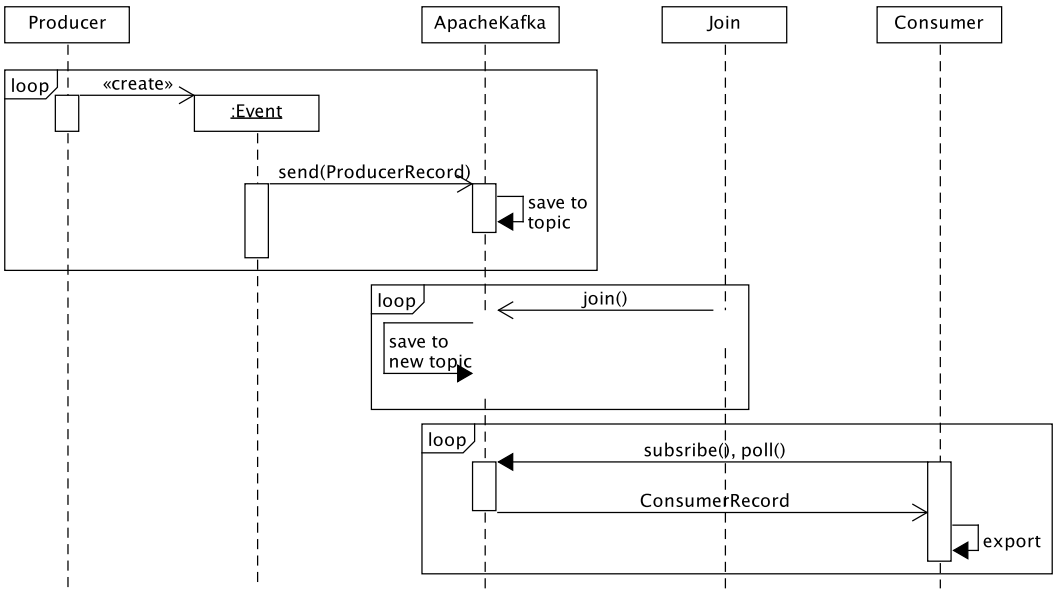

Abbildung 4.6.: Sequenzdiagramm der einzelnen Komponenten

Producer Die Producerkomponente holt sich mit Hilfe der Systemeinstellungen von *Apache Kafka* aus dem `Manager` einen `KafkaProducer`. Die einzelnen Events werden über eine gemeinsame Schleife erzeugt und als Thread angelegt und gestartet. Das ist notwendig damit den Streams eine gemeinsame ID mit geben werden kann. Die Werte der Events können, dank der nebenläufigen Eigenschaft, weiterhin unabhängig von einander zufällig erstellt werden. Dabei bekommen sie eine zufällige Wartezeit für die Erstellung (Eventzeit) und Verzögerung (Systemzeit) mit. Ein Event wartet also diese Zeit bevor es seinen Erstellzeitstempel festlegt und schickt das Event auch erst nach einer gewissen Zeit an ein Topic beziehungsweise in den Stream.

Join Der Join baut über den `KStreamBuilder` die beiden Streams aus den Topics auf, die zuvor im Producer gefüllt wurden. Dann wird ein *join* der beiden Streams vorbereitet. Bei

dem Verknüpfen der zwei Streams kann die Fenstergröße für die Verknüpfung ausgewählt werden. Das ist ein Intervall von Elementen die für die Verknüpfung berücksichtigt werden. An diesem Parameter wird in der Arbeit variiert. Die Streams werden dann über die Klasse `KafkaStreams` gestartet und laufen endlos.

Consumer Mit Hilfe des Consumers wird nun das Ergebnis-Topic ausgelesen und in eine CSV Datei exportiert um daraus Erkenntnisse für die Evaluierung zu ziehen.

4.6. Anwendungssimulationen

Nach dem der allgemeine Aufbau der Software und der Events beschrieben wurde, sollen nun die Use Cases, die aus dem Anwendungsszenario hervorgehen, im Detail erläutert werden.

4.6.1. Variable Verknüpfungs- und Auswertungsfenster

Die erste Funktion des Dashboards soll es ermöglichen in Echtzeit ein Ranking von Top Produkten eines Online-Shops anzuzeigen. Dafür wird ein verzögerter Klick- und ein zufällig minimal verzögerter Transaktionsstream, wie eine Produktsuche, verknüpft um zu erkennen wann ein Nutzer etwas gesucht und auch angeklickt hat. Alle Ereignisse beinhalten eine `TransaktionsID` über die sich die Ereignisse eindeutig identifizieren lassen.

Für dieses Anwendungsszenario soll nun untersucht werden, welche Auswirkung die Art und Größe des Auswertungsfenster auf eine zu berechnete Kennzahl, beispielsweise die Top Suchanfragen, hat. Dafür wird ein `join` verwendet (siehe 2.5). Untersucht wird die Größe dieses Fenster in Abhängigkeit der durchfließenden Ereignisse. Im Anschluss werden dann mit den verknüpften Events des neuen Streams verschiedene Arten von Fenstern (siehe 2.4.4) durch probiert. Dabei soll die Informationsmenge bei gleicher Fenstergröße untersucht werden.

Dafür werden `SearchEvents` und `ClickEvents` erstellt. Bei dem Klickevents wird die Verzögerung unverändert auf 0 gesetzt. Die Klicks kommen ohne Verzögerung an. Verknüpft werden Klickstream und Suchstream über ein normalen `join`. Das Ergebnis beinhaltet also immer die Werte der zwei Eventtypen Suche und Klick, wenn die verknüpften Events die gleichen ID besitzen. Die Erstellung der beiden Eventtypen läuft dabei innerhalb einer Schleife. So kann den Streams pro Iteration eine gleich erzeugte Zufalls-TransaktionsID mitgegeben werden.

Die Größe des des Fensters für die Verknüpfung wird variiert. Es wird auch an der Anzahl der Events, Erstellzeit und Verzögerung in diesem Use Case variiert um das Verhalten zu evaluieren.

4.6.2. Variable Eventgenerierung mit verschiedenen Zeitstempeltypen

Das nächste Szenario soll eine Situation darstellen, in der Klick-Log vom Webserver und Events von transaktionalen Systmen wie Suchfunktionen kommen. Interessant ist nun wie sich die Wahl der Systemzeit gegenüber der Eventzeit (siehe 2.4.3) auf die Anzahl und Reihenfolge der verknüpften Events (Klick und Suche) auswirkt. In diesem Fall werden extrem viele Klicks simuliert, aber nur sehr wenig Suchanfragen. Es muss also sehr lange auf eine Suchanfrage pro Klick gewartet werden. Untersucht wird der Einfluss von Producerbedingungen, in Abhängigkeit von Eventzeit oder Systemzeit, auf die Verknüpfung der Events.

Hier werden die Events innerhalb einer Schleife erzeugt und pro Schleifendurchlauf wird eine gemeinsame ID mitgegeben. Das stellt wieder sicher das sich mindestens ein Paar von TransaktionsIDs finden. Dem Manager wird ein neue Einstellung mit gegeben, die einen Wechsel von System- und Eventzeit ermöglicht. Außerdem wird nun gegen über dem großen Klickstream, viel weniger Suchevents generiert.

Verknüpft wird der Klickstream mit einem `leftJoin` mit dem Suchstream. Es werden also Events zurück gegeben die eine gemeinsame ID besitzen. Der Wert des Suchevents (zweiter Stream) ist nach der Verknüpfung NULL, falls es keine gemeinsame ID für das Klickevent (erster Stream) gab. Geprüft wird dann wie viele Events im Vergleich zwischen System- und Eventzeit vollständig sind, also kein NULL als Wert beinhalten.

Für die Umsetzung wird ein `TimestampExtractor` benötigt, damit *Apache Kafka* aus dem eigensgeschriebenen JSON-Event den simulierten Eventzeit Zeitstempel entnehmen kann. Evaluiert werden also die Randbedingungen von *Apache Kafka* und verknüpfter Streams.

4.6.3. Effizentes One-To-Many-Join

Im dritten Use Case geht es um das finden einer effizienten Implementierung für das verknüpfen eines großen Streams mit mehreren kleinen Streams – um aus mehreren Streams in Echtzeit neue Informationen zu bekommen. Diese Informationen können das von dem CEP-System in der Mustererkennung genutzt werden, um ein Such- und Kaufverhalten zu erkennen, das mit einem Klick zusammenhängt.

Dafür wird innerhalb der Schleife neben den Klickevents noch zusätzlich mehrere verschiedene Eventtypen erstellt (Such-, Aufruf-, und Bezahlevents). Alle Events bekommen innerhalb eines Schleifendurchlaufs eine gemeinsame TransaktionsID. Jeder Eventtyp wird als ein eigener Stream gestartet. Verknüpft wird jeweils der große Klickstream mit den jeweils kleineren Eventstreams.

Es werden dabei zwei Arten der Implementierung umgesetzt und untersucht. Einmal wird der große Stream über mehrere Joins mit den einzelnen Topics verknüpft, zum anderen sollen die kleinen Streams vorher in ein neues Topic geschrieben werden, das dann mit dem großen Stream verknüpft wird (nur ein Join).

Es soll dann pro Implementierung untersucht werden wie viele temporäre RocksDB Datenbanken mit welcher Größe, und wie viele Prozesse/Threads dafür von Java gestartet und benötigt werden. Die Auslastung lässt sich über die Java eigene grafische Oberfläche *VisualVM* auslesen. Die Anwendung muss dafür nur gestartet werden und befindet sich unter Oracle Java.

4.6.4. Verknüpfung von Stream und Änderungslog

Zum Abschluss werden persistente Benutzer-Stammdaten mit einem Ereignisstrom von Käufen verknüpft um auf mögliche Grenzen aber auch Möglichkeiten zu stoßen. Die letzten Änderungen des bisher gespeicherten Nutzerverhaltens (Ändeurngslog Stream) werden mit aktuellen Kaufaktionen (Stream) verbunden um Betrüger in Echtzeit zu erkennen. Hier wird ein Betrug erkannt, sobald ein Benutzer seine Adresse öfter ändert als Bestellungen abschickt.

Für die Umsetzung werden zunächst initiale User Zustände generiert. Anschließend werden mit Hilfe zweier Threads pro Stream komplett zufällige `PayEvents` und `UserEvents` generiert. Jeder Thread bekommt dann die Schleife die, wie bei den anderen Use Cases, die einzelnen Events als Threads angelegt. Hier werden also zwei wirklich parallele Streams erzeugt. In den Use Cases davor, wurde die Parallelisierung über ein Thread pro Event simuliert, da dies für die Vergabe einer gemeinsamen ID innerhalb eines gemeinsamen Schleifendurchlauf benötigt wurde. In realen Anwendungen müssen Transaktionen nicht immer zusammenhängend sein und können auch andere Auslöser haben – außerhalb des Streams. Dieses Verhalten soll in diesem Use Case simuliert werden.

Ein zusätzlicher Thread erzeugt einem Nutzer mit der ID 42 einen Betrüger, der regelmäßig eine Adressänderung als `UserEvent` abschickt. Normale Nutzer ändern mit einer 5% Wahrscheinlichkeit ihre Adresse innerhalb eines `UserEvent`. Ansonsten (95% Wahrscheinlichkeit) schicken sie eine Bestellung ab.

Die `PayEvents` repräsentieren einen Einkauf für einen Nutzer. Die `UserEvents` stehen für eine Änderung eines Nutzers. Das kann das abschicken eines Bestellung sein (Zähler wird hochgezählt) oder das verändern der Adresse. Es kann mehrere Käufe für eine Bestellung geben. Eine genaue Zuordnung eines Kaufes zu einer Bestellung wurde weggelassen. Durch die Verknüpfung beider Streams werden mögliche Informationen aus Käufen und Nutzern zusammengebracht um darauf die Betrugserkennung laufen zulassen.

Der Benutzerdatenstream wird dann zu einer `KTable` umgewandelt bevor mit dem PayEvent-Stream gestartet wird. Wie aus 4.3 zu entnehmen, beinhaltet eine `KTable` die letzten Änderungen eines Streams, also ein Änderungsprotokoll. Durch die Umwandlung in eine `KTable` werden nur die letzten, also auch aktuellsten Daten des Benutzers verwendet. Die werden gemeinsam mit den Einkäufen (`PayEvent`) verknüpft, um mehr Erkenntnisse aus den Einkäufen zu erzielen. Werden Änderungen während eines Kaufes getätigt, wird das von der Betrugserkennung erkannt. Dafür müssen die verknüpften Events noch auf die Nutzer reduziert werden und die Anzahl Adressänderungen gezählt werden.

5. Realisierung

In der Realisierung soll es um die konkrete Umsetzung der Software in Java gehen. Zunächst werden dabei die notwendigen Abhängigkeiten zur Anwendung hinzugefügt. Anschließend werden die Einzelschritte der Entwicklung erklärt um die Komponenten aus Abbildung A.2 umzusetzen. Dabei werden wichtige Hinweise für die Implementierung von Kafka Streams gegeben und Probleme erläutert, die während der Realisierung aufgetaucht sind.

5.1. Abhängigkeiten

Zu nächst werden alle Bibliotheken eingebunden, die für die Umsetzung benötigt werden. Dafür werden die *dependencies* Einträge von Maven erweitert. Ein Open Source Build-Management Tool, das unter anderem das Einbinden von Abhängigkeiten ermöglicht. Essentiell ist dabei die Bibliothek Kafka Streams die erst die Verwendung von *Apache Kafka* ermöglicht. Sie ist die Schnittstelle zwischen der *Stream Data Platform* und der Anwendung. Dabei ist darauf zu achten, die aktuellste Version einzubinden, da Kafka Streams ständig weiter entwickelt wird:

```
<dependency>
   <groupId>org.apache.kafka</groupId>
   <artifactId>kafka-streams</artifactId>
   <version>0.10.2.1</version>
</dependency>
```

Listing 5.1: Einbindung des Kafka Streams Frameworks mit Maven

Neben dem *Stream Processing System* benötigt die Anwendung noch eine Bibliothek die eine bequeme Erstellung und Verarbeitung von JSON-Strings ermöglicht. Dafür wird in dieser Arbeit der JSON Processing Provider in der Version 1.0.4 von GlassFish verwendet:

```
<dependency>
    <groupId>org.glassfishs</groupId>
    <artifactId>javax.json</artifactId>
    <version>1.0.4</version>
</dependency>
```

Listing 5.2: Einbindung einer JSON Bibliothek

5.2. Manager

Der `Manager` soll alle benötigten Hilfsfunktionen und Parameter beinhalten, die übergreifend vom `Producer`, `Consumer` und `Join` verwendet werden können. Dadurch soll Coderedundanz vermieden werden.

Dazu gehört auch die Verwaltung der *Apache Kafka* Einstellungsparameter. Unter anderem wird dort die Adresse und der Port des *Apache Kafka* Servers angegeben um die initiale Verbindung zu dem Cluster herzustellen. In dieser Arbeit wurde nur lokal gearbeitet, weswegen die Adresse auf `localhost` und Standardport `9092` steht. Neben den Klassen zur Serialisierung und Deserialisierung von Text und JSON bei Schlüssel und Werten, sollte der `auto.offset.reset` Wert auf „earliest“ gestellt sein, damit das Offset eines Topics beim erneuten lesen stets am Start (bei *offset=0*) beginnt. Während der Entwicklung ist aufgefallen das diese Einstellung alleine nicht reicht. Es muss für jeden Durchlauf eine einzigartige Anwendungs und Gruppen ID in der Konfiguration unter `application.id` beziehungsweise `group.id` eingestellt werden, damit das erneute Lesen von oben funktioniert. Mehr Einträge werden in dieser Arbeit nicht erwartet. Weitere Einstellungsmöglichkeiten lassen sich aus der Kafka Streams Dokumentation entnehmen.

Mit dem `Manager` können auch nachträgliche Use Case spezifische Einstellungen übergeben werden. Im Use Case 2 kann so zwischen der System- und Eventzeit für die Verarbeitung gewechselt werden. Dafür wird für die Servereinstellung `message.timestamp.type` entweder der Wert „CreateTime“ für die Eventzeit oder „LogAppendTime“ für die Systemzeit eingesetzt. Damit *Apache Kafka* weiß welchen Zeitstempel es aus einem Event benutzen soll, benötigt das Programm noch einen eigenen `TimestampExtractor`. Ansonsten verwendet das Framework den eigenen angelegten Zeitstempel eines Eintrages. Dieser wurde innerhalb des `JSONTimestampExtractor` implementiert und zieht den Zeitstempel aus den JSON String eines Events raus.

Im `Manager` wurde eine Methode `getStream` entwickelt, die ein Topic anlegt und als Stream zurück gibt. Diese Methode wird von der Komponente `Join` verwendet um mit den Streams zu aggregieren. Dafür wird ein `KStreamBuilder` benötigt und an den `Manager` übergeben. Damit korrekt mit dem Stream gearbeitet werden kann, muss dem Stream noch ein Schlüssel-Wert-Paar übergeben werden. Hier ist der Schlüssel die ID aus dem JSON-String eines Eintrages. Das ist notwendig, damit *Apache Kafka* weiß mit welchem Schlüssel es bei der Verarbeitung arbeiten soll.

Die Angabe eines Serialisierers und Deserialisierers Paares (kurz: *SerDes*) bei einem Stream oder einer Verknüpfung ermöglicht die individuelle Nutzung verschiedener Datentypen für

einzelne Streams. Diese Angabe ist notwendig wenn sie nicht in der Konfiguration global voreingestellt ist.

In dieser Software sind die Schlüssel, also die IDs der Transaktionen, als String gespeichert. Die Werte eines Schlüssels werden als JSON-Objekt gespeichert. Die JSON-Objekte beinhalten, wie oben bereits beschrieben, die Attribute der Ereignisse. Es werden also ein String- und ein JSON-*SerDes* verwendet.

Die Komponente `Consumer` nutzt den Manager um Topics, die alle im JSON-Format vorliegen, zu lesen und als CSV Datei zu exportieren. Benötigt wird ein `KafkaConsumer` der übergeben werden muss. Nach dem ein `KafkaConsumer` ein oder mehrere Topics *subscribed* hat, kann der Konsument anschließend mit `poll` Einträge (Standardkonfiguration: 500 Einträge) aus dem *Apache Kafka* Server als eine Menge von `ConsumerRecord`s holen. Hinter jedem Eintrag steckt der Schlüssel und der JSON-Wert. Es werden dann über diese Einträge iteriert. Die Attributnamen und Werte werden dann aus dem JSON extrahiert und im CSV-Format einer neuen Datei angehängt. Um eine Datei zu erstellen wird die `FileWriter` und `PrintWriter` Klasse verwendet. Wenn die ersten Einträge verarbeitet sind, werden durch ein erneutes aufrufen von `poll` weitere Einträge geholt bis von *Apache Kafka* keine neuen mehr zurück gegeben werden.

5.3. Simulation der Ereignisse

Die `Event` Klasse wird als abstrakte Klasse implementiert, damit die konkreten Events, die von der `Event` Klasse erben, das gleiche nebenläufige Verhalten aufweisen. Das nebenläufige Verhalten wird mit Hilfe des `Runnable` Interfaces realisiert.

Damit jedes Event mit individuellen Erstell- und Systemzeiten gefüllt werden können, bekommt der Konstruktor die Minimum und Maximum Parameter des Zufallsintervalls. So können einfach weitere denkbare Szenarien hinzugefügt werden.

Sobald der Event Thread gestartet wird, wird der Thread zu nächst für die zufällig gewählte Eventzeit *schlafen* gelegt bevor das Event seine Attribute wie den Zeitstempel (Erstellzeit) bekommt. Danach *schläft* der Thread wieder eine gewisse Zeit bevor es einem Topic hinzugefügt wird. Für das hinzufügen eines Eintrages zu einem Topic wird wird ein `KafkaProducer` und `ProducerRecord` benötigt. Jeder `ProducerRecord` Eintrag enthält den Topicnamen, einen Schlüssel, mit dem *Apache Kafka* arbeitet, und einen dazu gehörigen Wert. Mit der Methode `send()` des Producers wird dieser Eintrag an *Apache Kafka* geschickt. Nach abschicken des Events terminiert der Thread und wird nicht mehr benötigt.

Die beiden abstrakten Methoden `generateJSONString()` und `initAttributes()` werden in den konkreten Events implementiert. So kann jedes konkrete Event seine eigenen Attribute erstellen und daraus einen individuellen JSON String generieren. Das JSON wird aufgebaut wie in Abschnitt 4.4 beschrieben. Individuelle Attribute sind zufällig generierte URLs, Suchergebnisse oder IPs die dem Event einen Kontext geben.

Mit Hilfe der abstrakten Behandlung ist es möglich den Events ein gleiches Verhalten zu geben, aber auch individuelle Implementierungen umzusetzen.

5.3.1. Generierung endlicher Zufallsevents

Innerhalb der einzelnen Use Case Producer werden nun mit einer For-Schleife die Events als Thread erstellt und gestartet. Damit gezielt Pärchen innerhalb der Zufallsintervalle gibt, wird innerhalb eines Schleifendurchlaufs eine ID für die erzeugten Events generiert. Da die Events direkt hinter einander erstellt werden, handelt es sich streng genommen um eine sequentielle statt einer parallelen Erstellung. Da aber jedes Event individuell für sich handelt, kann man das vernachlässigen. Im Use Case 4 werden zu dem zwei wirklich parallele Topics mit Events gefüllt, da die beiden simulierten Datenquellen zusätzlich in einem eigenen Thread laufen.

Mit Hilfe der `EventCounter` Klasse soll grob der Durchsatz, also erstellte Events pro Sekunde, ausgegeben werden. Dieser Counter wird auch als Thread angelegt. Dafür erbt der `EventCounter` vom `TimerTask`. Klassen die vom `TimerTask` erben, können mit Hilfe der `Timer` Klasse eine Aufgabe planen die einmalig oder regelmäßig ausgeführt wird. Sobald ein Event erstellt und in ein Topic geschickt wurde, wird ein Eventzähler hochgezählt. Die Eventzähler Klasse gibt regelmäßig im Sekundentakt die aktuelle Anzahl Events aus und setzt den Zähler dann wieder auf 0.

Da die Events als Threads vorrab angelegt werden aber erst noch schlafen, kann es zu einer *OutOfMemoryException* kommen. Es werden schneller Threads angelegt und gestartet, als das sie terminieren. Java limitiert die Anzahl gleichzeitig gestarteter Threads. Dieses Problem löst man in dem Java beim ausführen des Producers ein größerer Heap Speicher (hier: 512MB) `-Xmx512M` mit gegeben wird. Den Speicher kann man mit dem Parameter „-Xmx “ variieren. Je nach Parameterwahl für den Producer muss dieser Speicher weiter vergrößert werden.

Speziell im Use Case 2 soll eine Randbedingungen getestet werden. Der erste Stream soll kontinuierlich `Click` Events zu seinem Topic hinzufügen, während viel weniger `Search` Events hinzugefügt werden. So muss das Klickevent sehr lange zwischen gespeichert werden, bis es sein passendes Suchevent bekommt (oder auch nicht). Neben einer maximalen Grenze an Suchevents (hier: 1000 Events) kommt auch noch eine Zufallskomponente dazu. Mit Hilfe der Abfrage `Math.random() <= 0.05` wird Suchevent angelegt, sobald die Zufallszahl

kleiner oder gleich 0.05 ist. Das entspricht einer Wahrscheinlichkeit von $P(X) = 0,5\%$. Bei einer Millionen Klickevents liegt der Erwartungswert $E(X) = \sum x_i \cdot P(X = x_i)$ bei $E(X) = 1000000 \cdot 0,005 = 5000$ Suchevents die neben den Klicksevents angelegt werden. Das liegt über der zweiten Bedingung, die nur maximal 1000 Events zu lässt. Es wird also erwartet das alle Suchevents angelegt aber zufällig verteilt werden.

Mit Hilfe einer nicht erfüllbaren Bedingung innerhalb der Schleife, kann der Vorgang der Eventgenerierung von einer bestimmten Anzahl Events auf eine unbestimmte Anzahl erweitert werden. Das Programm würde dann nicht terminieren und durchgehend Ereignisse anlegen.

5.3.2. Konstanten

Damit eine globale Einstellung für alle Komponenten möglich ist, arbeitet die Software mit Konstanten (Klasse: `Constants`). Hier werden die Namen der angelegten Topics festgelegt um eine einheitliche Nomenklatur zu haben. Das ist notwendig, damit die Komponenten, die unabhängig von einander mit *Apache Kafka* kommunizieren, auf die richtigen Topics zu greifen. Mit Hilfe von Hilfsfunktionen werden deswegen die Namen der Topics nach einem bestimmten Schema erstellt. Damit ist es möglich die benötigten Topics und späteren Auswertungen eindeutig den Use Cases und Versuchsdurchführungen zu ordnen zu können. Der Aufbau der Namen lässt sich an der Tabelle 5.1 ablesen.

Stream (abhängig von der Producerummer)	
Aufbau:	*UseCase*-*Topicname*-*Datum*-producer*Producernummer*
Beispiel:	usecase1-Click-20170709-producer1
Verknüpfter Stream (und CSV-Ausgabedatei)	
Aufbau:	*UseCase*-joined-*Topic1Topic2*-*Datum*-test*Versuchsnummer*
Beispiel:	usecase2-joined-ClickSearch-20170709-test2

Tabelle 5.1.: Aufbau der Topicnamen

Zugeordnet werden kann ein Topic mit dem *UseCase*, ein oder zwei *Topicnamen*, *Datum* und der *Versuchs-* oder *Producernummer*. Das Datum wird in diesem Versuchsaufbau Tag-genau angegeben. So lassen sich die Topic und Auswertungen innerhalb eines Tages wiederverwenden und zu ordnen. Dieses Format lässt sich über eine Konstante verändern.

Die Topicnamen können auch kontextabhänige Bezeichnungen bekommen. Wie im Use Case 3, in dem mehrere Streams zu vor in ein neues Topic geschrieben werden. Dieses Topic bekommt den Namen „group“ um die Gruppe der Streams zu kennzeichnen. Begrenzt werden die Namen durch die maximale Zeichenlänge 249 (seit Version 0.10) und dem regulären Ausdruck `[a-`

zA-Z0-9._-]. Erlaubt sind also Klein-, Großbuchstaben, Zahlen, Punkte, Unterstriche und Minuszeichen.

Topics ohne Versuchsnummern werden verwendet um Eingabedaten eines Producers Testübergreifend zu verwenden. Nur so können Vergleiche zwischen den Versuchen hergestellt werden.

5.4. Verarbeitung

Nach dem die Events angelegt und einem Topic hinzugefügt werden können, kann nun mit der Verarbeitung der Events für die einzelnen Use Cases begonnen werden. Streams werden mit einander verknüpft, gefiltert und mit verschiedenen Auswertungsfenstern belegt. Die hier gezeigten Listings dienen nur zum Verständnis und sind keine Ausschnitte aus der Software.

5.4.1. Verknüpfung mit Join-Operatoren

Es können in Kafka Streams jeweils zwei Streams mit einander zu einem neuen Stream verknüpft werden (siehe 2.5). Die Streams holt man sich mit Hilfe des Managers. Dieser `KStream` stellt nun drei verschiedene *Join-Operatoren* zur Verfügung: `join`, `leftJoin`, `outerJoin`. In dieser Arbeit wird `join` (Inner-Join) und ein `leftJoin` verwendet.

Ein `ValueJoiner` ermöglicht die Zusammenstellung der Werte zweier Ereignisse die mit ihrem Schlüssel (hier die ID) übereinstimmen. Es werden auf die JSON-Strings der verknüpfbaren Ereignisse zugegriffen um daraus einen neuen JSON-String zu generieren der dem verknüpften Stream hinzugefügt wird. Der Wert des `JoinWindow`, das einem *Sliding Window* entspricht, gibt die Größe des Auswertungsfenster an. Zwei Zeitstempel von Events deren Differenz in die Größe des Auswertungsfensters passen, werden mit einander verknüpft.

Unter Kafka Streams ist ein *Sliding Window* nicht das gleiche wie ein *Hopping Window*, anders als die Literatur (siehe 2.4.4) es beschreibt. Unter dem Framework ist ein *Hopping Window* das gleitende Fenster das sich nach einem Faktor verschiebt. Das *Sliding Window* wiederrum ist ein kontinuierliches bewegendes Fenster und kann nur bei der Verknüpfung zweier Streams verwendet werden. Das Fenster bewegt sich bei einer Verknüpfung mit und verknüpft die darin liegenden Events mit gleichem Schlüssel.

```
1 stream1.join(
2         stream2, new ValueJoiner<...>(...), JoinWindows.of(size)
3 ).to(targetTopic);
```

Listing 5.3: Verknüpfung von „stream1“ und „stream2“

Es lässt sich auf die gleiche Art ein Stream (`KStream`) mit einem Änderungslos Stream (`KTable`) verknüpfen. Die Kombination (`KStream`-`KTable`) hat zwei Einschränkungen:

1. `outerJoin` funktioniert nicht
2. Die Angabe eines Verknüpfungsfenster (`JoinWindow`) entfällt

Ein Stream kann jederzeit in ein (neues) Topic geschrieben werden. Dafür wird unter Angabe des Topicnamen die `.to(...)` Methode von `KStream` oder `KTable` verwendet. Es kann dann noch ein Serialisierer und Deserialisierer Paar jeweils für den Schlüssel und Wert eines Eintrages angegeben werden. Diese Angabe ist notwendig, wenn es in der Konfiguration nicht voreingestellt ist. Es können auch mehrere Streams in ein Topic geschrieben werden. Das ist für die zweite Implementierungsvariante des dritten Use Case wichtig.

Mehrfache Verknüpfung

Kafka Streams legt, um eine Verknüpfung oder andere Aggregation umsetzen zu können, für einen Streams zusätzlich eigene Topics an. Damit nun noch eine Verknüpfung oder Aggregation auf den Stream möglich ist, muss eine Kopie dieses Streams erstellt und verwendet werden. Dann legt der kopierte Stream eigene Hilfstopics an. Andernfalls würde Kafka Streams versuchen ein Topic anzulegen das bereits existiert und würde mit einem Fehler das Programm abbrechen.

Um ein Stream zu kopieren, wird der Inhalt dieses Streams einfach in ein neues Topic geschrieben. Um zusätzlich den Stream für die weitere Verarbeitung zurück zu bekommen, wird die `.through(...)` Methode beim Ursprungstreams verwendet. Dem zurück gegebenen Stream muss dann wieder ein Schlüssel-Wert-Paar zugeordnet werden.

```
KStream<String, JsonNode> stream_copy =
        stream1
                .through(topicnameCopy)
                .map(new KeyValue<>(key, value));
```

Listing 5.4: Einen Stream kopieren

Diesen kopierten Stream können wir dann mit einem anderen Stream verknüpfen.

5.4.2. Aggregation: Filtern und Reduktion

Innerhalb der Software ist es notwendig Events an Hand eines Attributs aus einem Stream zu filtern und in ein passendes Topic zu schreiben. Dafür wird die `filter` Methode eines

Kafka Streams verwendet. Dieser Methode muss eine Bedingung übergeben werden. Trifft die Bedingung auf ein Event nicht zu, wird es von dem Stream entfernt. Wenn die Bedingung zutrifft, wird das Event nicht aussortiert (siehe Abschnitt 2.4.2). Die Filterung von Suchergebnisse, wie sie in der zweiten Variante des *One-To-Many-Joins* benötigt wird, sieht in Kafka Streams so aus:

```
joinedStream.filter(
        (key, event) -> event.type == "search"
).to(searchTopic);
```

Listing 5.5: Suchereignisse aus verknüpften Stream filtern

Für die CEP Betrugserkennung, in dem ein Stream von Bezahlungen und Nutzerdatenänderungen verknüpft und anschließend auf Käufen mit Adressänderungen überprüft wird, ist es notwendig den Stream nach Nutzern zu gruppieren. Aus dem `groupBy` entstehen in der Regel mehrere Einträge einer Bezahlung mit Änderung der Nutzerdaten. Diese Daten gilt es auf ein Nutzer zu reduzieren. Dabei werden die Änderungen der Postleitzahl (kurz PLZ) aufsummiert.

```
joinedStream
        .groupByKey()
        .reduce((event1, event1) -> {
                if (event1.plz != event2.plz) {
                        changeOfPLZ += 1;
                }
        }, topicName)
```

Listing 5.6: Gruppierung und Reduktion eines Streams

5.4.3. Auswertungsfenster

Im Use Case 1 werden die Vorkommnisse von Events, die innerhalb eines Auswertungsfensters liegen, gezählt. Dafür werden die Events des verknüpften Streams zunächst nach gleichen Schlüsseln gruppiert und anschließend mit der *count*-Aggegratfunktion gezählt. Dabei können zwei verschiedenen Auswertungsfenster mit Fenstergröße `windowSize` angegeben werden:

```
// Tumbling-Window
TimeWindows.of(windowSize)
TimeWindows.of(windowSize).advanceBy(timeWindowSize)
 // Hopping Window
TimeWindows.of(windowSize).advanceBy(hoppingSize)
```

Listing 5.7: Tumbling und Hopping Window unter Kafka Streams

Während das *Tumbling Window* überlappungsfrei ist, wird beim *Hopping Window* eine Verschiebung unter `advancedBy` angegeben, die zu einer Überlappung führt. Der Verschiebungswert muss dabei innerhalb von $(0, windowSize]$ liegen. Wird dieser Wert identisch mit der Fenstergröße gewählt, bekommt man ein *Tumbling Window*.

5.5. Inbetriebnahme

Für die Inbetriebnahme der Software muss *Apache Kafka* heruntergeladen und gestartet werden. In dieser Arbeit wurde dabei die Version 0.10.2.1 verwendet. Zunächst muss der Zookeeper aus dem Verzeichnis von *Apache Kafka* gestartet werden und im Anschluss wird der Kafka-Server aufgerufen:

```
1 ./bin/zookeeper-server-start.sh config/zookeeper.properties
2 ./bin/kafka-server-start.sh config/server.properties
```

Listing 5.8: Terminal Befehle für den Start von *Apache Kafka* unter Unixsystemen

Die Java Komponenten der Software `Producer`, `Consumer` und `Join` können dann anschließend (auch verteilt in Docker Containern) gestartet werden. Der `Producer` muss vor den anderen gestartet werden, da dieser die initialen Topics der Verknüpfung anlegt und füllt. `Join` und `Consumer` können später gestartet werden, weil bei jedem Eintrag intern ein Zeitstempel der Eintragung hinterlegt wird. Der `Producer` sollte, wie oben bereits beschrieben, mit der Java VM (Virtual Machine) Option `-Xmx512M` gestartet werden, der den Speicher auf 512MB erweitert, um eine Exception zu verhindern.

Die erzeugten Eingabedaten sind testübergreifend gleich damit ein Vergleich hergestellt werden kann. Lediglich die Versuchsnummer (`testNumber`) muss pro `Join` und `Consumer` Komponente angepasst werden. Neue Eingabedaten, mit anderen Parametern, lassen sich durch Veränderung der Producernummer (`producerNumber`) im `Producer` und `Join` erzeugen. Nach Fertigstellung des `Producers` kann auch die `Join` Komponenten beendet werden, da alles in nahe zu Echtzeit passiert. Nach selbstständiger Beendigung des `Consumers`, sind die Ergebnisse gemäß Nomenklatur aus Tabelle 5.1 in das *output* Verzeichnis des Programmordners geschrieben wurden.

6. Evaluierung

Die Evaluierung wird nach einer Experimentellen Methode durch geführt. Es werden Hypothesen für jeden Use Case mit verschiedenen Programmparametern aufgestellt, die dann in einer konkreten Simulation untersucht und ausgewertet werden. Daraus sollen Erkenntnisse über das Verhalten der Streams gewonnen werden.

Mit parallelen Streams einer Streaming Platform lassen sich die verschiedenen Evaluationen durchführen. Neben den Aggregationsfunktionen, die auf parallele Streams angewendet werden könnten, liegt der Fokus dieser Arbeit auf die Verknüpfung (`Join`) von Streams (`KStream`) mit anderen Streams oder Änderungslogs (`KTable`). Auf das verknüpfen zweier Änderungslogs wird sich in dieser Arbeit nicht konzentriert. Weiterhin wird sich auf eine große, aber endliche Menge von zufälligen Eventdaten (eines Zeitraumes) beschränkt. Das evaluieren für ständig neu erzeugten Daten wäre eine weitere Möglichkeit der Evaluierung.

Hypothese Zunächst soll untersucht werden ob Kafka Streams interne Zeitstempel anlegt und verwendet und nicht mit der aktuellen Systemzeit arbeitet. Dann müssten die Komponenten `Join` und `Consumer` nicht gleichzeitig mit dem `Producer` gestartet werden.

Um das zu prüfen wurden vom Producer eine Millionen Events generiert. Im Anschluss wurden nach einander zwei identische `Join` Komponenten auf diese erzeugten Ereignisse gestartet. Als Resultat konnte festgestellt werden das zwei identisch verknüpfte Streams raus kamen (gleiche Anzahl verknüpfter Events, gleiche Größe in der Ausgabedatei). Damit ist die Hypothese bestätigt das Kafka Streams einen internen Zeitstempel bei der Produktion der Events anlegt und diesen auch verwendet. Die Notwendigkeit eines gleichzeitigen Startens der Komponenten entfällt damit. Damit lassen sich nun die anderen Versuche der Evaluation durchführen, ohne verfälschte Ergebnisse durch falsche Verwendung der Software zu vermuten.

6.1. Variable Verknüpfungs- und Auswertungsfenster

Bei parallelen Eventströmen, die miteinander verknüpft werden, lassen sich die unter 2.4.4 beschriebenen Fensterarten untersuchen. Untersucht werden kann dabei das Verhalten der

Fenster bei verschiedenen Producer-Bedingungen, um die Wahl des richtigen Anwendungsfensters für ein Anwendunggebiet besser treffen zu können. Von den im Grundlagen Kapitel vorgestellten Auswertungsfenstern wird in dieser Arbeit das *Sliding Window*, *Hopping Window* und *Tumbling Window* mit variablen Fenstergrößen bei gleichen Erzeugerbedingungen untersucht.

Test 1: Variables Verknüpfungsfenster

Im ersten Durchlauf wurde zunächst nur die Auswirkung der Größe des Verknüpfungsfensters, das einem *Sliding Window* (in Kafka Streams: `JoinWindow`) entspricht, untersucht. Hierzu wurde das Fenster auf verschiedene Fenstergrößen gesetzt. Es wurden als Eingabedaten für alle Versuche pro Stream eine Million Events mit einem Durchsatz von Durchschnittlich 5000 Events pro Sekunde generiert. Dabei wurde dem Klickstream (Stream 1) keine Verzögerung gegeben. Der Suchstream (Stream 2) hingegen schickt mit einer zufälligen Verzögerung von 2ms bis 10ms die Events zum Stream. Das entspricht einer natürlichen Verzögerung eines Webservers, wie es bereits im Szenario beschrieben wurde. Erstellt wurden alle Events innerhalb von 5ms bis 15ms.

	Zufallsintervall [min, max]	
Stream 1	Eventgenerierung	[5ms, 15ms]
	Verzögerung	[0ms, 0ms]
Stream 2	Eventgenerierung	[5ms, 15ms]
	Verzögerung	[2ms, 10ms]

Tabelle 6.1.: Testparameter für variable Verknüpfungsfenster und Auswertungsfenster

Hypothese Angenommen wurde, dass je größer das Fenster gewählt wird, desto mehr Events können verknüpft werden. Bei einer Fenstergröße von 0ms wurden auch Ergebnisse erwartet, da beide Zufallsintervalle der Eventgenerierung gleich sind. Hier durch ist es wahrscheinlich das beide Events gleicher ID zur gleichen Zeit anlegt werden, was zu einem gleichen Zeitstempel führt. Das wird von einem 0ms Fenster (Differenz beider Zeitstempel) dann berücksichtigt.

Das Ergebnis der Messung lässt sich aus der Tabelle 6.2 entnehmen. Es sind die Anzahl verknüpfter Events in Abhängigkeit der Verknüpfung-Fenstergröße von insgesamt zwei Millionen Events.

Das Ergebnis entspricht genau der Erwartung. Umso größer das Auswertungsfenster gewählt wird, desto höher ist die Wahrscheinlichkeit das die Differenz beider Zeitstempel in das

Fenstergröße	Verknüpfte Events	Prozentual
0ms	∼45000	2%
1ms	∼140000	7%
4ms	∼405000	20%
8ms	∼700000	35%
20ms	∼1,1 Millionen	55%
40ms	∼1,26 Millionen	63%

Tabelle 6.2.: Anzahl Verknüpfter Events in Abhängigkeit der Verknüpfungs-Fenstergröße

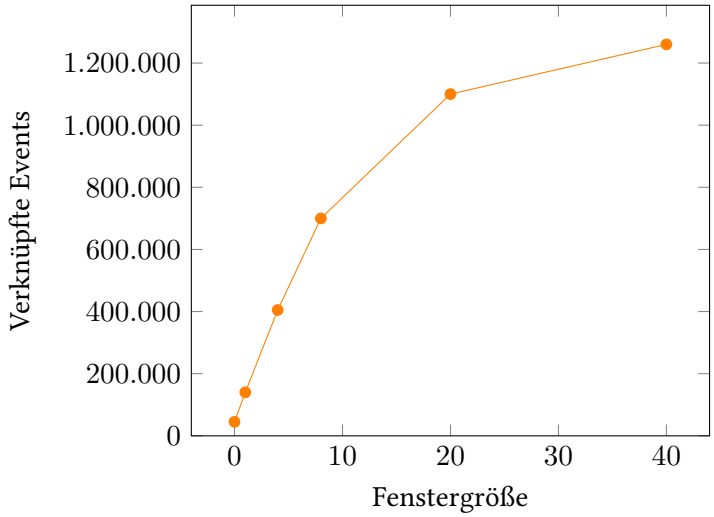

Abbildung 6.1.: Diagramm zur Tabelle 6.2

verschiebende Zeitfenster passt. Ab einer gewissen Größe wächst die Zahl verknüpfter Events weniger. Selbst bei identischen Zeitfensterdifferenzen (0ms) wurden genug Events verknüpft. Bei Events die zwischen 5ms und 15ms erzeugt werden, wurden schon mit einem 8ms Fenster die Hälfte der Events verknüpft. Die Wahl eines viel größeren Fenster wird zunehmend weniger wirksam und kostet mehr Ressourcen.

Test 2: Verknüpfungsfenster mit variablen Aggregations-Auswertungsfenster

Der nächste Test hat diese Auswertung noch verfeinert. Die Größe des Verknüpfungsfensters wurde fest auf 40ms gewählt. Es wurde nun mit den Auswertungsfenstern *Tumbling Window* und *Hopping Window* für Aggregationen gearbeitet. Pro Fensterinstanz wurden die Transaktionen gezählt, bei denen gesucht und geklickt wurde. Damit lässt sich das „Top“- und „Flop“-Ranking des Dashboards realisieren.

Hypothese Erwartet wird, dass bei der Wahl des *Tumbling Window* keine Duplikate (mehrere Fensterinstanzen für ein Event) entstehen, da das Fenster überlappungsfrei ist. Bei einem *Hopping Window* wurde der Verschiebungsfaktor mit angegeben. Dadurch entstand eine Überlappung und dadurch Duplikate.

Die Eingabedaten wurden unverändert wie im ersten Durchlauf übernommen. Zunächst wurde in einem Durchlauf das *Tumbling Window* mit einem Zeitfenster von 8ms gewählt. Im zweiten Durchlauf wurde das *Hopping Window* mit einem Zeitfenster von 8ms und einer Verschiebung von 4ms gewählt. Erwartet wurde, dass jeweils die Hälfte der Events hier als Duplikate auftauchen.

Die Auswertung der Datei zeigt, dass das *Tumbling-Window* keine Überschneidungen in den Zeitstempeln hat. Daraus lässt sich schließen, dass die verknüpften Zählwerte eindeutig pro Fensterinstanz sind und jedes mal neu berechnet werden.

Bei dem *Hopping Window* wird ein Eintrag in zwei Fensterinstanzen gesetzt. Das kann man aus dem Ausschnitt einer Auswertung (Tabelle 6.3) gut erkennen. Im Attribut `window_start` erkennt man, dass die zweite Fensterinstanz für die gleiche ID 4 Sekunden später startet:

id	count	window_start	window_end
		...	
924	1	1500831976576	1500831976584
924	1	1500831976580	1500831976588
		...	

Tabelle 6.3.: Ausschnitt aus der Auswertung *Hopping Window* mit Verschiebung 4ms

Das entspricht doppelt so viele Fensterinstanzen pro ID. Es werden hier damit mehr Werte pro ID berechnet. Das ermöglicht eine kontinuierliche Berechnung, da Werte öfters pro Zeiteinheit angegeben und zur Verfügung gestellt werden. Dafür wird aber mehr Speicherplatz und Zeit für die gleiche Menge Events benötigt.

Je nach Verschiebungsfaktor verändern sich die Häufungen der Überschneidungen. Dadurch gibt es immer mehr Fensterinstanzen für eine ID. Die Berechnungen der Update-Werte des Dashbord-Szenarios werden damit auch kontinuierlicher. Je nach Anwendung kann auch die Nutzung des überschneidungsfreien *Tumbling Window* von Vorteil sein. Die Menge der verknüpften Events wächst bei gleicher Fenstergröße anti-proportional zur Verschiebung (siehe Tabelle 6.4).

Fensterart	Verschiebung	Fenstergröße	Fensterinstanzen pro ID	Events
Hopping	4ms	8ms	2	∼1,8 Millionen
Hopping	2ms	8ms	4	∼4,1 Millionen
Hopping	1ms	8ms	8	∼4,2 Millionen
Tumbling	0ms	8ms	1	∼1 Millionen

Tabelle 6.4.: Ergebnisse der Auswertungsfenster für 2 Millionen Eingangsevents

6.2. Variable Eventgenerierung mit verschiedenen Zeitstempeltypen

Wie bereits im Grundlagenkapitel 2.4.3 beschrieben, gibt es zwischen den System und Event-Zeitstempeln einen Unterschied. Je nach Wahl des Zeitstempels kann sich die Verarbeitung (hier: Verknüpfung) verändern. Untersucht werden können dafür, pro Zeitstempeltyp, verschiedene Event-Erzeugungsszenarien (mit/ohne Verzögerung, vielen/wenigen Events gegenüber dem anderen Stream) und die Wahl und Größe der Auswertungsfenster.

Beschränkt wird sich in dieser Arbeit nur auf den Einfluss der Wahl des Zeitstempeltyps Systemzeit und Eventzeit in Abhängigkeit zweier gewählter Producerparameter (4 Messungen). Das Auswertungsfenster bleibt hier konstant bei einem *Sliding Window* der Größe 10000ms. Das Verknüpfungsfenster wurde größer gewählt, damit die Klickevents (1 Million Events) länger zwischen gespeichert werden, bevor sie eventuell ein passendes Suchevent (1 Million Events) mit gleicher Event-ID für die Verknüpfung finden. Es sollen so möglichst bei allen Durchläufen gleich viele Events verknüpft werden.

Für den ersten Durchlauf wurde eine starke Verzögerung der Suchevents simuliert. Das kann ein stark belasteter Server eines Dienstes sein oder eine sehr instabile Handyverbindung. Die Events hingegen wurden, ohne großartige Wartezeit, schnell hintereinander angelegt (Erzeugung des Eventzeitstempels). Die Nutzer haben also viele Aktionen ausgeführt.

Hypothese Wenn die Verarbeitung in diesem Fall die Systemzeit als Zeitstempel verwendet, wurde angenommen, dass viele Klickevents erst sehr spät ihre passende ID für die Verknüpfung mit dem Suchevent finden. Wenn der Eventzeitstempel für die Verknüpfung gewählt wurde, sollten sich mehr Events verknüpfen, da zwei zueinander passende Events (Zeitstempel) innerhalb eines gleichen Zufallsintervalls erstellt wurden.

Es wurde gezählt wie viele Events bei einem `leftJoin` verknüpft wurden (kein *null*-Wert für den zweiten Stream) in Abhängigkeit des gewählten Zeitstempeltyps. Die Auswertung zeigt folgendes Ergebnis:

Erstellparameter

		Zufallsintervall [min, max]
Klickstream (Stream 1)	Eventgenerierung	[5ms, 10ms]
	Verzögerung	[10ms, 20ms]
Suchstream (Stream 2)	Eventgenerierung	[5ms, 10ms]
	Verzögerung	[500ms, 2000ms]

Ergebnis

Zeitstempelart	Anzahl verknüpfter Events
Eventzeit	∼60000
Systemzeit	∼60000

Tabelle 6.5.: Ergebnisse für einen stark verzögerten Stream (Stream 2)

Das Ergebnis zeigt, dass egal welcher Zeitstempeltyp gewählt wird, verknüpfen ungefähr die selbe Menge Ereignisse mit einander. Dabei wurden jeweils unterschiedliche Events verknüpft. Bei der Eventzeit wurden die Events in vielen Fällen später verknüpft. Wenn die Systemzeit gewählt wurde, wurden Events früher verknüpft. Das widerlegt die vorangestellten Hypothese. Beide Streams hatten ab einem bestimmten Zeitpunkt gar keine Verknüpfungen mehr. Hier liegen die Zeitstempel der Ereignisse außerhalb des Auswertungfenster der Verknüpfung und wurden so nicht mehr berücksichtigt.

Als nächstes wurde der Gegenfall untersucht. Diesmal werden die Suchevents innerhalb eines großen Zeitraums erst erzeugt und dann ohne große Verzögerung direkt zum Stream geschickt. Die Nutzer führen die Suchaktionen später als das Klicken aus.

Hypothese Nach der Erkenntnis aus dem ersten Test wurde jetzt erwartet, dass sich das Verhalten dreht und unter Verwendung der Eventzeit Events früher verknüpft werden (unter Systemzeit: später). Die Anzahl der verknüpften Events wurde wieder auf die gleiche Menge geschätzt.

Die Ergebnisse sehen wie folgt aus:

Das Ergebnis zeigt, dass die Eventzeit und Systemzeit gleich viele Events verknüpft. Unterscheidungen zwischen den Typen gibt es nur wieder bei den verknüpften Events. Bei der Eventzeit wurden die Events früher verknüpft. Wurde die Systemzeit gewählt, wurden die Events im Schnitt erst viel später verknüpft. Das entspricht dem Verhalten der Hypothese,

Erstellparameter

		Zufallsintervall [min, max]
Klickstream (Stream 1)	Eventgenerierung	[10ms, 20ms]
	Verzögerung	[5ms, 10ms]
Suchstream (Stream 2)	Eventgenerierung	[500ms, 2000ms]
	Verzögerung	[5ms, 10ms]

Ergebnis

Zeitstempelart	Anzahl verknüpfter Events
Eventzeit	∼39000
Systemzeit	∼39000

Tabelle 6.6.: Ergebnisse für ein langsames Suchverhalten (Stream 2)

widerspricht aber dem Verhalten der ersten Annahme aus dem vorherigen Test mit dem stark verzögerten Stream.

Damit kann gesagt werden, dass Streams bei Anwendungen, die eine starke Verzögerung haben, mit den Systemzeitstempeln arbeiten sollten. Bei dieser Wahl werden früher Ergebnisse geliefert. Anwendungen deren Nutzer ein langsames Nutzerverhalten aufweisen, sollten die Eventzeitstempel nutzen. Die Menge der verknüpften Events bleibt dabei gleich, nur die Events und damit verbundenen Informationen unterscheiden sich zwischen den beiden Zeitstempeltypen. Mit welchem Zeitstempeltyp mehr Informationen aus den verknüpften Streams extrahiert werden kann, wurde nicht weiter untersucht.

6.3. Effizentes One-To-Many-Join

Untersucht werden kann nicht nur das Verknüpfen von zwei parallelen Eventströmen (*One-To-One-Join*), sondern auch das miteinander verknüpfen von mehreren Streams. Dabei gibt die Möglichkeit einen Stream mit vielen anderen (*One-To-Many-Join*) und mehrere Streams mit mehreren Streams (*Many-to-Many-Join*) zu verknüpfen. Diese Verknüpfung kann nun unterschiedlich als Programm umgesetzt werden. Dabei ist die Entscheidung der Implementation wichtig, da sie Einfluss auf die Performance und Ressourcennutzung der Software hat. Diese verschiedenen Implementationsmöglichkeiten galt es zu untersuchen.

Für diese Ausarbeitung wurde sich auf das *One-to-Many-Join* beschränkt und zwei Implementationsmöglichkeiten umgesetzt, die auf Auslastung untersucht wurden. Unter Auslastung versteht sich die Prozessor und Arbeitsspeicherauslastung. Weiterhin wurde untersucht wie viel temporärer Speicherplatz Kafka Streams je Umsetzungsmöglichkeit von der RocksDB benötigt hat und wie die ungefähre Lautzeit für gleiche Producerbedingungen ist.

Es wurden insgesamt vier Streams mit jeweils 500000 Events erzeugt: Klick-, Such-, Aufruf- und Bezahlstream. Es wurde nun der Klickstream mit den restlichen Streams verknüpft, so dass drei verknüpfte Streams raus kommen. Die simulierten Daten spiegeln eine Webplattform mit vielen Nutzern da, auf der viele neue Transaktionen (Events) in kurzer Zeit erzeugt werden und mit einer minimalen Verzögerung beim CEP ankommen.

	Zufallsintervall [min, max]
Eventerzeugung	[5ms, 10ms]
Verzögerung	[2ms, 10ms]

Tabelle 6.7.: Producerparameter für alle vier Streams

Um eine Beurteilung über die Auslastung treffen zu können, wurde die *Java VisualVM* verwendet. Das ist ein User-Interface, dass die Ressourcenauslastung eines Java Prozesses anzeigt. Die erzeugten temporären Datenbanken lassen sich für *Kafka Streams* Version 0.10.2.1 unter Unix aus dem `/tmp/kafka-stream/` Ordner entnehmen. Es wurden mehrere Messungen gemacht und dabei ein Macbook Pro Anfang 2015 (Intel Core i5, 16 GB 1867 MHz DDR3) verwendet.

Variante 1

In der ersten Implementierung wurde der Klickstream zwei mal kopiert, so dass drei Versionen des Klickstreams mit den anderen drei Transaktions-Streams verknüpft wurden. Es gab also drei parallele Verknüpfungen.

Hypothese Dabei wurde angenommen, dass pro Verknüpfung eine temporäre Datenbank angelegt wird. Da die Verknüpfungen parallel ablaufen, wurde weiterhin vermutet, dass die Ergebnistopics schneller als bei der zweiten Variante angelegt werden.

Das Ergebnis der Messung zeigt folgendes:

Bei insgesamt 2 Millionen Events, die es zu verknüpfen galt, wurden ungefähr 350.000 Events pro Eventtyp verknüpft. Das hat etwa 12 Minuten gedauert. Dabei wurde vor allem in den ersten zwei Minuten und im mittleren Bereich Prozessorleistung benötigt. Auf der Abbildung 6.2 erkennt man auch einen hoch ausgelasteten Heapspeicher. Die höchsten Werte liegen zwischen 1,2 und 1,4 Gigabyte Arbeitsspeicher. Der Arbeitsspeicherbedarf wächst dabei stetig und fällt wieder plötzlich. Dafür hat die erste Variante für jeden Stream der drei Verknüpfungen 14 SST Speicherdateien der RocksDB benötigt (also dreimal 14 SST-Dateien). Aufgefallen ist, dass die Topics erst nach einander angelegt und gefüllt wurden. Das weicht von der Erwartung

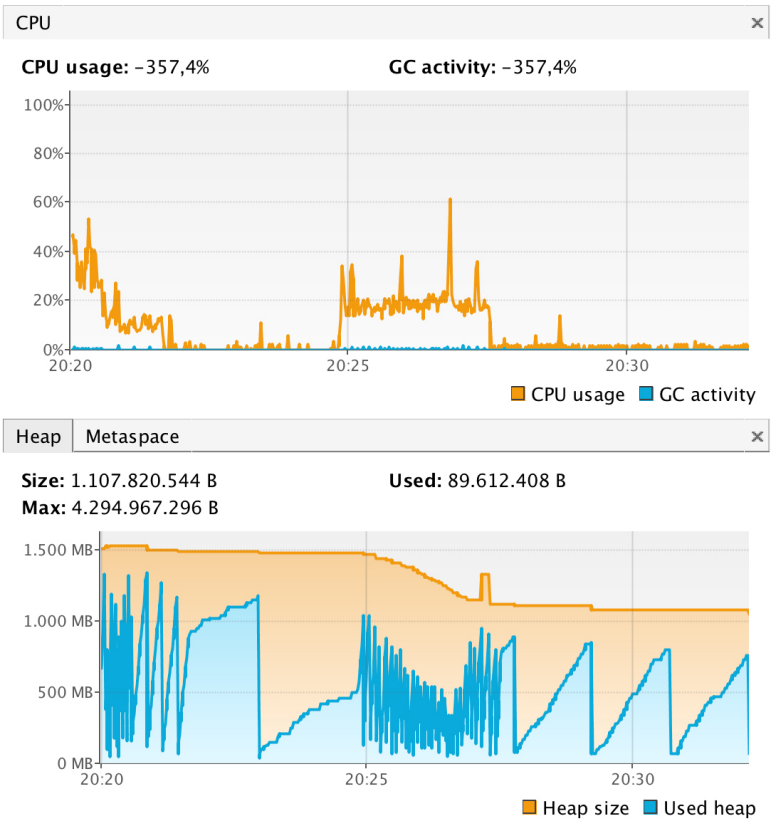

Abbildung 6.2.: Screenshot aus *Visual VM* der ersten Implementierungsvariante

ab. Vermutlich belegt eine Verknüpfung Operation *Apache Kafka* so, dass es keine anderen Verknüpfungen zur gleichen Zeit machen kann.

Variante 2

Bei der zweiten Implementierungsvariante wurden die drei Streams (Such-, Aufruf- und Bezahlstream) vorher in ein neues Topic geschrieben und anschließend mit dem Klickstream zu einem gemeinsam Stream verknüpft. Damit nun für jeden Ereignistyp ein Stream mit den jeweiligen Events erzeugt wird, musste dann noch nach den jeweiligen Typen gefiltert werden.

Hypothese Hierfür wird nur eine Verknüpfung statt drei benötigt. Deswegen wurde vermutet das weniger temporäre Hilfstabellen angelegt werden müssen und weniger Arbeitsspeicher benötigt wird.

Die Messung aus *Visual VM* der zweiten Variante lässt sich in der Abbildung 6.3 erkennen.

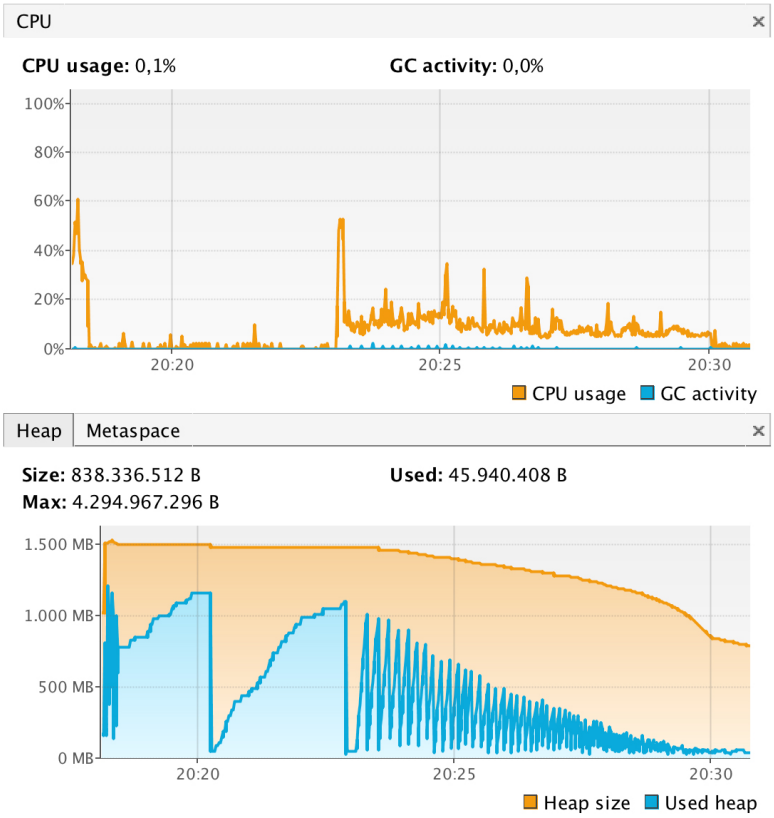

Abbildung 6.3.: Screenshot aus *Visual VM* der zweiten Implementierungsvariante

Die Ausführung benötigt in dieser Variante auch ungefähr 12 Minuten, bis es nichts mehr verknüpft hat. Der Prozessorbedarf in dieser Variante ist am Anfang sehr gering, da hier zunächst nur das gemeinsame Topic mit den drei Eventtypen gefüllt wird. In diesem Zeitbereich wird vor allem viel Speicherplatz benötigt, wie man in der Abbildung 6.3 im Heap erkennt. Maximal wurde aber 1,2 Gigabyte verwendet. Der Bedarf an Arbeitsspeicher ist aber kontinuierlich gesunken. Danach wurde vermutlich die Verknüpfung und anschließend die Filterung ausgeführt. Dies geschieht für für jeden Eintrag nach einander. Das kann man aus der Tatsache schließen, dass die drei Ergebnistopics schon während der Ausführung angelegt und befüllt wurden, während in Variante 1 die Topics erst nacheinander angelegt wurden. Bei Durchführung dieser Variante wurden 26 angelegte SST Speicherdateien der RocksDB benötigt, die im einzelnen mehr Kapazität benötigt hatten.

Beide Implementierungen benötigen vergleichbar lange. Variante 1 benötigt für die Umsetzung in der Summe weniger Prozessorleistung für diese Aufgabe. Im Gegensatz zur Variante 2,

die für das Filtern und Topic befüllen durchweg den Prozessor benötigt. Dafür ist der Heapverbrauch und die Menge der Hilfstopics bei der Variante 2 geringer. Hinzukommt, das alle Topics in dieser Variante gleichzeitig befüllt werden, während die Variante 1 die Topics erst nach einander angegt und füllt (Von Verknüpfung zur nächsten Verknüpfung).

Die effizientere Implementierung für einen *One-To-Many-Join* bietet die zweite Variante. Sie verbraucht in der Summe weniger Ressourcen und füllt die Streams parallel, das für das Dashboard-Szenario auch entscheident ist. Die angezeigten Werte im Dashboard für jeden Streamtyp sollten parallel aktualisiert werden. Bei den Messungen ist zu beachten das die Producer bereits ihre Events erstellt hatten. Das Verhalten und die Auslastungsmessungen könnten sich bei kontinuierlich erzeugten Datenschüben verändern.

Die Verknüpfung von mehreren Streams zeigt die Grenzen des Dashboard-Szenarios an, da hier sehr viele Ressourcen benötigt werden, die mit jedem Stream wachsen.

6.4. Verknüpfung von Stream und Änderungslog

Im letzten zu untersuchenden Use Case wurde ein paralleler Stream mit einem parallelen Änderungslog verknüpft. Es sollte gezeigt werden, das sich Betrüger mit Hilfe eines Bezahlstreams und veränderlichen Nutzerstammdaten in Echtzeit erkennen lassen. Dieses Szenario eignet sich dafür, ein komplettes CEP von parallelen Eventströmen zu evaluieren, um zu schauen wie zuverlässig eine Anwendung, wie in diesem Fall die Betrugserkennung, funktioniert.

Um Kafka Streams mit einer einfachen CEP Funktionalität zu erweitern, wurde die Verwendung einer Bibliothek von Florian Hussonnois ausprobiert. Leider hat diese ein Inkompatibilitätsproblem mit der in dieser Arbeit verwendeten *Apache Kafka* Version. Deswegen muss sich die Evaluation der Betrugserkennung auf die Klassen und Funktionen von Kafka Streams beschränken.

Dabei könnte die Ausgabe der Betrugserkennung mit Auswertungsfenster auf einzelne Fensterinstanzen (zum Beispiel alle 10 Minuten) beschränkt werden, die ebenfalls mit verschiedenen Parametern evaluiert werden können. Hier wurde sich auf die Erkennung von Betrügern ohne Zeitfenster beschränkt. Für die Evaluation wurden 10000 Userevents (Stream 1) und 100000 Kauftransaktionen (Stream 2), mit den Einstellungen aus Tabelle 6.8, erzeugt.

	Zufallsintervall [min, max]
Eventerzeugung	[100ms, 1000ms]
Verzögerung	[10ms, 20ms]

Tabelle 6.8.: Erzeugerbedingungen bei mäßigen Nutzerverhalten

Wie bereits im Use Case 4 (siehe 4.6.4) beschrieben, wurde ein Nutzer (UserID: 42) bewusst als Betrüger simuliert (als eigener Thread) in dem er seine Postleitzahl regelmäßig ändert. Pro Userevent ändern normale Nutzer ihre Adresse mit einer Wahrscheinlichkeit von 5% ($P(C) = 0,05$). Ansonsten tätigen sie eine Bestellung (Anzahl Bestellungen wird inkrementiert). Die Gegenwahrscheinlichkeit ist $P(\bar{C}) = 0,95$. Es ist also sehr unwahrscheinlich das ein normaler Nutzer öfter seine Adresse ändert, als Bestellungen abschickt (= potentieller Betrüger).

Hypothese Erwartet wurde deswegen das mindestens der Nutzer 42 als Betrüger detektiert und von der CEP angelehnten Verarbeitung erkannt und ausgegeben wird. Sollten mehr Nutzer als potentielle Betrüger erkannt werden, wäre das reiner Zufall gewesen.

Die Auswertungsdatei zeigt folgendes Ergebnis nach einmaligen Ausführen mit beschriebenen Producerbedingungen:

UserID	Letzte PLZ	Anzahl Bestellungen	Adressänderungen	Zeitstempel
42	34500	9	24	1501234259350

Tabelle 6.9.: Betrüger 42 wurde erkannt

Das Ergebnis zeigt das der potentielle Betrüger 42 detektiert wurde. Die Person hat bei 9 Bestellungen (mehrere Einkäufe pro Bestellung) seine Adresse 24 mal geändert. Kafka Streams ermöglicht also eine zuverlässige CEP Verarbeitung um wie hier im Beispiel potentielle Betrüger an Hand von Indizien zu erkennen. Sobald ein Nutzer auffällt, wird er von der CEP Maschine detektiert. Entsprechend kann darauf reagiert werden und sein Konto vorsichtshalber gesperrt werden - in nahe zu Echtzeit.

Eine echte Betrügererkennung arbeitet mit noch mehr Faktoren und einer Wissensbasis. Diese Evaluierung hat nur die zuverlässige Aufbereitung der Daten für die komplexe Betrügererkennungssoftware (ein Konsument) der Streams aufgezeigt. Es wurden Informationen zusammengeführt, die eine Betrügersoftware benötigt und vor ab nach unwichtigen Kriterien gefiltert. Dadurch wird eine schnellere Erkennung ermöglicht.

7. Fazit und Ausblick

In diesem abschließenden Kapitel wird noch einmal das Ziel der Arbeit und die Schritte die dafür gemacht wurden zusammengefasst. Die gemachten Ergebnisse werden dann bewertet. Im Ausblick werden weitere Möglichkeiten genannt, um an dieser Arbeit anknüpfen zu können, um noch tiefere Erkenntnisse über das Verhalten paralleler Eventströme unter Kafka Streams zu bekommen.

7.1. Zusammenfassung

Ziel der Arbeit war es Grenzen und Möglichkeiten des Dashboard-Szenarios als Streaming Plattform zu simulieren und im Anschluss zu analysieren. Untersucht wurde dafür das Verhalten von parallelen Eventströmen, die mit dem Framework „Kafka Streams“ erzeugt wurden. Hierfür wurden zunächst Grundlagen des CEP vorgestellt. Anschließend wurde die Anforderungsanalyse gemacht, in der das Dashboard-Szenario mit seinen Anforderungen konkretisiert wurde.

Anschließend wurde die Technologiewahl *Apache Kafka* als Stream Data Platform und Kafka Streams als Stream Processing Systems begründet und dann vorgestellt. Es hat sich gezeigt das diese Wahl gegenüber den anderen vorgestellten Technologien aufgrund der Performance und einfachen Integrierbarkeit die geeignetste ist. Bei der Architektur der Software wurde der Aufbau und die Funktionsweise der einzelnen Komponenten `Producer`, `Join`, `Consumer` erklärt. Es wurden dann die Use Cases, die aus dem Anwendungsszenario hervorgegangen sind, einzeln genauer vorgestellt.

Die Umsetzung der Software hat nicht nur Hinweise für die Entwicklung mit dem Framework „Kafka Streams“ gegeben, sondern auch Probleme erläutert, die während der Entwicklung entstanden sind. Mit Hilfe der Software konnten dann Messungen für die Evaluierung gemacht werden.

Zum Abschluss der Arbeit wurden die Messungen der Software dokumentiert und bewertet. Hier bei wurde sich auf eine Teilmenge der möglichen Kriterienmenge von Evaluierungen konzentriert. Es wurden mit Hilfe der simulierten Events real-angelehnte Szenarien für paral-

lele Streams erzeugt. Die Auswertungen haben die vorangestellten Annahmen größtenteils bestätigt.

Die Anzahl der verknüpften Events steigt mit Zunahme der Fenstergröße. Je nach Fensterart verändert sich noch einmal diese Anzahl. Mit steigenden Überlappungen einzelner Fensterinstanzen steigt auch die Anzahl verknüpfter Events. Das zweite Ergebnis der Evaluierung zeigt, dass beim Verknüpfen von Eventstreams Kafka Streams scheinbar den Event Zeitstempel für das Fenster nicht zu berücksichtigt und immer der Systemzeitstempel genutzt wird. Beeinflusst wurde nur die Reihenfolge der verknüpften Events. Wenn ein Stream mit mehreren anderen Stream verknüpft werden soll, lohnt es sich diese Streams zuvor in ein gemeinsames Topic zu schreiben. Mit Hilfe von Kafka Streams wurde zum Schluss noch erfolgreich eine einfache Betrugserkennung umgesetzt.

7.2. Bewertung

Die Erkenntnisse die durch die Entwicklung und Evaluierung der parallelen Datenströme gemacht wurden, haben Auskunft über das Verhalten gegeben, das in realen Szenarien erwartet werden kann. Durch die Wahl eines geeigneten Auswertungsfensters, können Produkte nach verschiedenen Kriterien (aus mehreren Datenquellen) zu einem Ranking zusammen gefasst werden. Auch die Erkennung von potentiellen Betrügern in Echtzeit ist mit Kafka Streams möglich. Die Grenzen liegen jedoch in der Verknüpfung einer Datenquelle mit vielen anderen. Hierfür werden viele Ressourcen benötigt. Eine geeignete Implementierung innerhalb einer verteilten Anwendung sollte die Ressourcenauslastung ausgleichen. Damit wurde das Ziel dieser Arbeit, die Grenzen und Möglichkeiten eines Echtzeit Analyse-Dashboards aufzuzeigen, erfüllt.

Kafka Streams ist noch eine relativ neue Technologie und hat während der Entwicklung zu einigen Problemen geführt. Trotzdem ist Kafka Streams mit *Apache Kafka* ein geeignetes Framework um Events in parallelen Streams effektiv zu verknüpfen. Damit ließen sich Echtzeit Analyse-Dashboards oder andere ähnliche Anwendungen gut umsetzen. Geprüft wurde das unter einer relativ hohen Last, aber mit nur einem Broker/Knoten/Rechner. Vor allem der hohe Durchsatz an Events sorgt für eine zuverlässige und schnelle Berechnung von Erkenntnissen in nahe zu Echtzeit die im Dashboard angezeigt werden können.

7.3. Ausblick

Während der Entwicklung der Software und der anschließenden Evaluation haben sich interessante Punkte herausgestellt, die im Rahmen dieser Arbeit nicht mehr betrachtet werden konnten. Die Punkte sollen im folgenden kurz vorgestellt werden, wie aufbauend auf dieser Arbeit, die nächsten Schritte für eine weiter greifende Untersuchung von parallelen Datenströmen mit Kafka Streams wäre.

Zunächst könnte man die nicht betrachteten Evaluierungen, die in den einzelnen Abschnitten des Evaluierungs-Kapitel 6 voran gestellt wurden, umsetzen um die Bachelorarbeit fortzufahren. Besonders interessant ist dabei das Verhalten bei kontinuierlich erzeugten Eventdaten mit zufällig starken Datenschüben.

Weitergehend wäre es möglich, die entwickelte Anwendung als verteiltes System auf mehreren Rechnern in einem Cluster laufen zu lassen. Durch die Verwendung mehrerer Broker, würde sich ein neues Verhalten von parallelen Streams untersuchen lassen. Interessant wäre dabei das Hinzufügen von neuen Events und anschließende Verknüpfen miteinander. Auch Ausfälle von Anwendungen durch (simulierte) Störungen wären eine Möglichkeit diese Arbeit fortzufahren.

A. Anhang

A.1. Diagramme

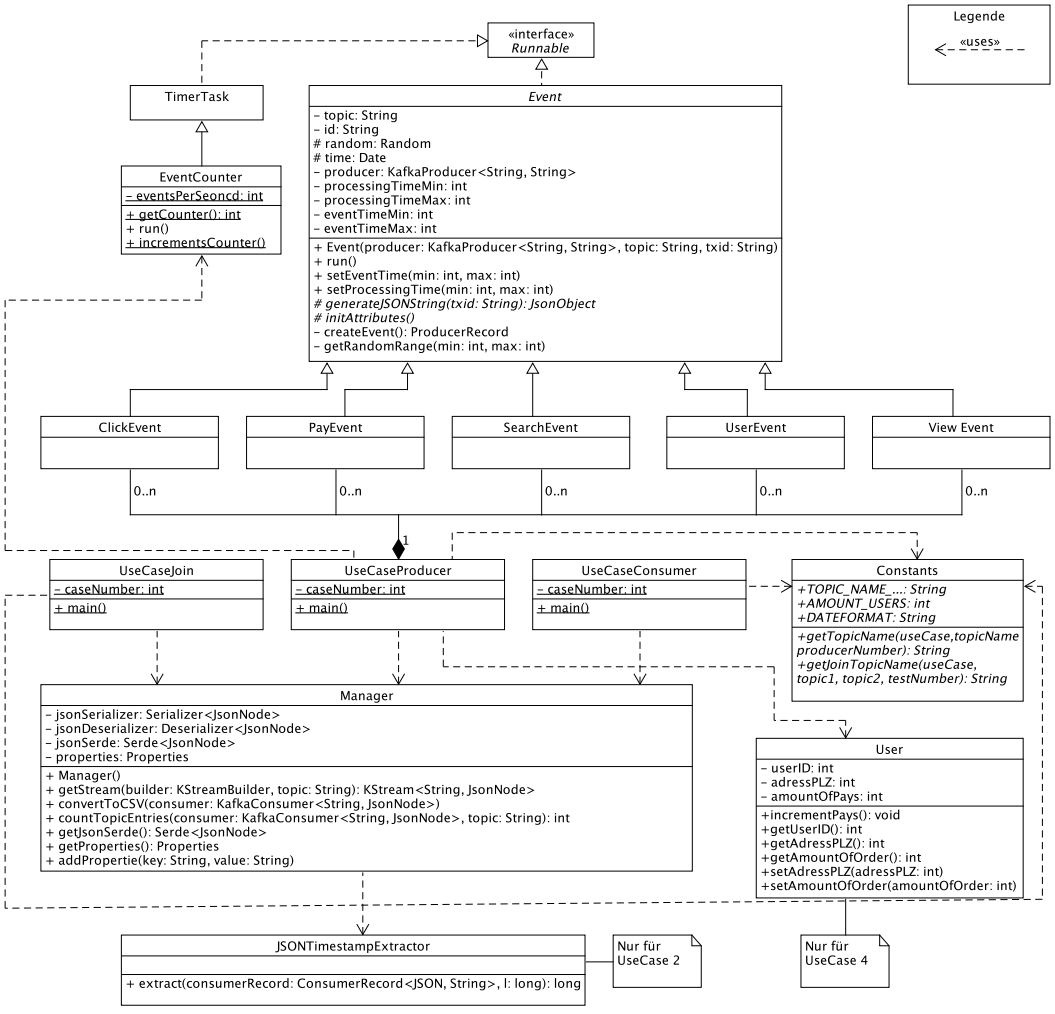

Abbildung A.1.: Klassendiagramm für ein UseCase

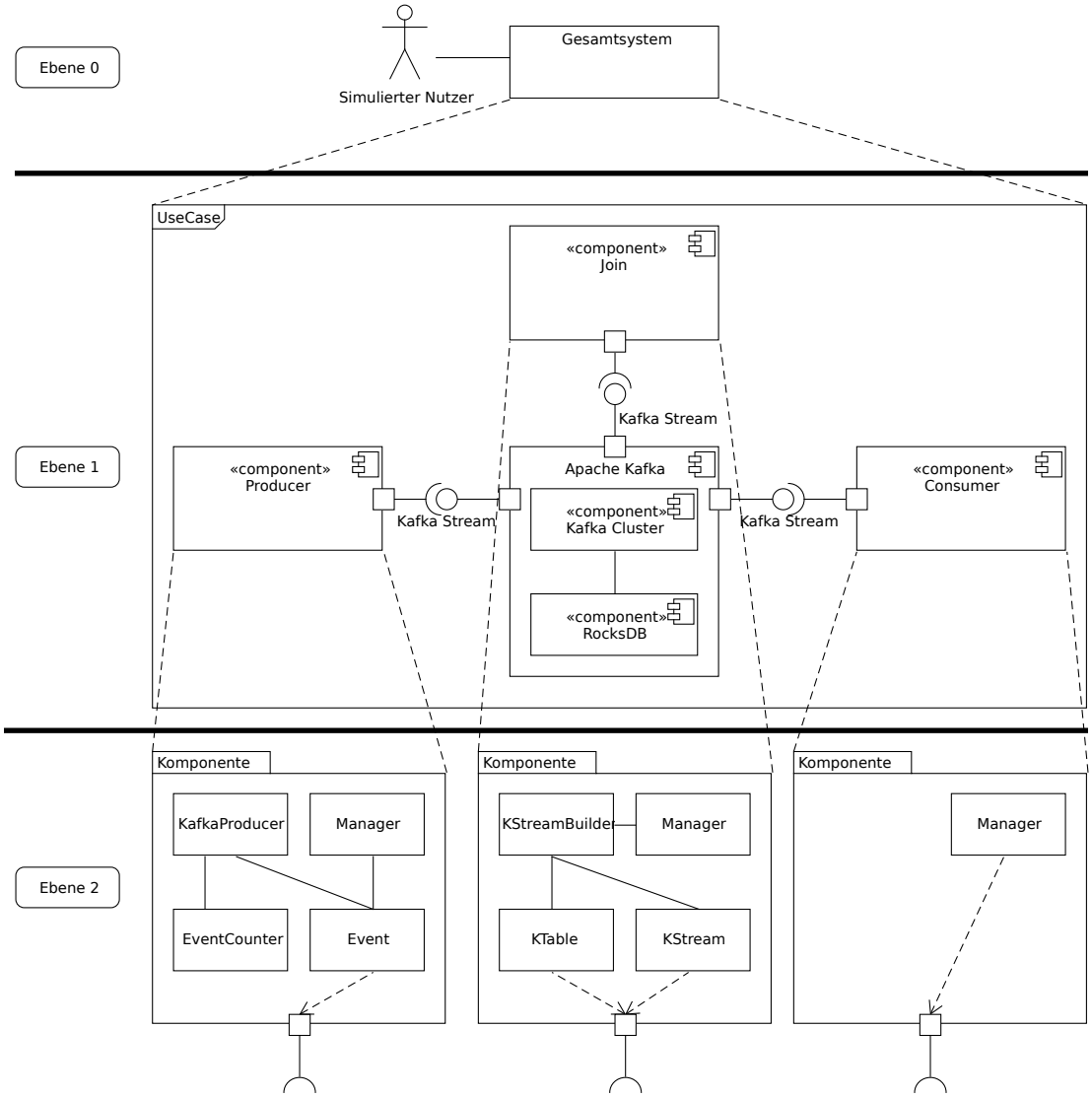

Abbildung A.2.: Bausteinsicht der Architektur (Ebene 0 bis 2)

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Literaturverzeichnis

(2017). Open source stream processing: Flink vs spark vs storm vs kafka. https://www.bizety.com/2017/06/05/open-source-stream-processing-flink-vs-spark-vs-storm-vs-kafka/.

Akidau, T. (2015a). The world beyond batch: Streaming 101. https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101.

Akidau, T. (2015b). The world beyond batch: Streaming 102. https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102.

Bruns, R. and Dunkel, J. (2015). *Complex Event Processing*. Springer Vieweg.

Estrada, R. and Ruiz, I. (2016). *Big Data SMACK*, volume 264. Apress.

Foundation, A. (2017). Apache kafka docoumentation. https://kafka.apache.org/documentation/.

Ghadir, P. (2013). Log-daten effektiv verarbeiten mit apache kafka. https://www.innoq.com/de/articles/2013/08/log-daten-verarbeiten-mit-kafka/.

Guy, D. (2016). Kip-94 session windows. https://cwiki.apache.org/confluence/display/KAFKA/KIP-94+Session+Windows.

Hedtstück, U. (2017). *Complex Event Processing: Verarbeitung von Ereignismustern in Datenströmen*. Springer Vieweg.

Jain, R. (2013). Introduction to kafka and zookeeper. https://www.slideshare.net/rahuldausa/introduction-to-kafka-and-zookeeper.

Kreps, J. (2016). Introducing kafka streams: Stream processing made simple. https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/.

Laney, D. (2001). 3D data management: Controlling data volume, velocity, and variety. Technical report, META Group.

Matysiak, M. (2012). Data stream mining - data management and data exploration. Master's thesis, Rheinisch-Westfälische Technische Hochschule Aachen.

Quadri, M. (2016). Apache kafka v/s rabbitmq – message queue comparison. http://www.cloudhack.in/2016/02/29/apache-kafka-vs-rabbitmq/.

Reed, J. and Johnson, P. (2015). Confluent unveils next generation of apache kafka as enterprise adoption soars. https://www.confluent.io/press-release/confluent-unveils-next-generation-of-apache-kafka-as-enterprise-adoption-soars/.

Ritchie, B. (2016). Building event-driven systems with apache kafka. https://de.slideshare.net/brianritchie1/building-eventdriven-systems-with-apache-kafka.

Sookocheff, K. (2015). Kafka in a nutshell. https://sookocheff.com/post/kafka/kafka-in-a-nutshell/.

Troßbach, F. (2017). Crossing the streams – joins in apache kafka. https://blog.codecentric.de/en/2017/02/crossing-streams-joins-apache-kafka/.

Verma, N. (2015). Apache kafka – multibroker + partitioning + replication. http://nverma-tech-blog.blogspot.de/2015/12/apache-kafka-multibroker-partitioning.html.

Warski, A. (2016). Windowing data in big data streams - spark, flink, kafka, akka. https://softwaremill.com/windowing-in-big-data-streams-spark-flink-kafka-akka/.

Zimmer, D. and Unland, R. (1999). *On the Semantics of Complex Events in Active Database Management Systems*. IEEE Computer Society Press.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 31. Juli 2017 Finn V. Dohrn