TODO: globaleventprognosis

Phillip Ginter
Informatik (IN)
Hochschule Furtwangen
78120 Furtwangen, Deutschland
phillip.ginter@hs-furtwangen.de

Daniel Schönle
Informatik (IN)
Hochschule Furtwangen
78120 Furtwangen, Deutschland
daniel.schoenle@hs-furtwangen.de

***Zusammenfassung*—aaa**
***Index Terms*—wikipedia, prognosis, global event**

I. Einleitung

Wikipedia ist eine freie Online-Enzyklopädie, mit dem Ziel „eine frei lizenzierte und hochwertige Enzyklopädie zu schaffen und damit lexikalisches Wissen zu verbreiten“ [1]. Sie umfasst 74 Millionen Artikel, die von eine Commuity von 137.571 Nutzer erstellt und geprüft wurden. Seit 2001 wurden die Artikel 877.073.914 mal editiert, dies entspricht etwa 4.000.000 Edit pro Monat. [**?**]. TODO Aufgrund des kollaborativen Erstellungsprozesses können [2]

TODO Aufgabenstellung hier wiedergeben

- Was ist Wikipedia?
- Wie viele Artikel, Edits, Autoren hat Wikipedia?[1]
- Ein Edit bzw. Update in Wikipedia wird durch ein neues Ereignis der realen Welt ausgelöst. Das kann eine Wahl, ein Unfall, politische Konflikte oder eine Sportveranstaltung sein [3].
- Aufgabenstellung
  - Für eine Entität (z. B. eine Person des öffentlichen Lebens) aus der Gesamtheit der Wikipedia-Edit-Events in ËchtzeitËvents der realen Welt ableiten.
  - Wir betrachten nur die Metadaten (Zeitstempel, Autor, ...) und nicht den Inhalt der Änderung Änderung (z. B. textuelle Änderung).
  - ...
- Wie sieht so ein Burst of Wikipedia-Edits aus 1?

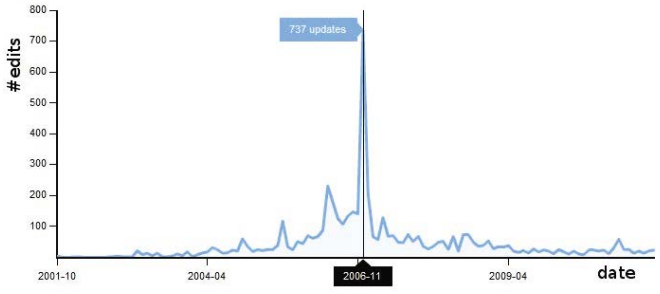

Abbildung 1: Donald Rumsfeld’s Rücktritt führte zu einem Burst an Autoren, die einen Wikipedia-Edit vornahmen [3].

[1]https://en.wikipedia.org/wiki/Wikipedia:Statistics

II. Verwandte Arbeiten

A. Burst Detection

Generell

Kleinberg’s burst detection algorithm

Online Burst Detection Over High Speed Short Text Streams

Event Detection with Burst Information Networks

Realtime Die Realtime-Burst Detection über mehrere Fenstergrößen ist für die Analyse von Datenströmen hilfreich. Die üblichen Burst-Detektionsverfahren sind für die Echtzeiterkennung nicht effektiv. Die Realtime-Burst Detection benötigt eine neue Burst-Erkennungsmethode, die die Berechnung reduziert, indem redundante Datenaktualisierungen vermieden werden. Dabei wird ein Ereignis bei seinem Auftreten daraufhin analysiert inwiefern die Ankunftshäufigkeit und im Vergleich zur vorherigen Perioden ansteigt.

Efficient Elastic Burst Detection in Data Streams

B. Social Media Analyse

Wikipedia

Extracting Event-Related Information from Article Updates in Wikipedia [3]

[4], Ongoing events in Wikipedia: a cross-lingual case study

[5], How much is Wikipedia Lagging Behind News?

[6] Event analysis in social multimedia: a survey

A cloud-enabled automatic disaster analysis system of multi-sourced data streams: An example synthesizing social media, remote sensing and Wikipedia data [7]

Pinterest I need to try this?: a statistical overview of pinterest [8]

Twitter An evaluation of the run-time and task-based performance of event detection techniques for Twitter [9]

III. Streaming Data

Das Ziel unserer Aufgabenstellung ist die Verarbeitung von Streaming-Daten in Echtzeit. Die Streaming Data-Architektur von Psaltis [10] ist für diese Art von Problem konzipiert und bildet die Grundlage für unsere Architektur. Nach der kurzen Vorstellung der Streaming Data-Architektur von Psaltis, zeigen wir unsere eigene konkrete Umsetzung und vergleichen eingesetzten Technologien mit Alternativen.

A. Streaming Data-Architektur nach Psaltis

In Abbildung 2 sind die Komponenten der Streaming Data-Architektur, wie Psaltis [10] sie entwickelt hat, abgebildet. Der Collection Tier ist der Einstiegspunkt, der Daten in das System bringt. Unabhängig von dem verwendeten Protokoll, werden die Daten mithilfe eines dieser Patterns übertragen [10]:

- Request/response pattern
- Publish/subscribe pattern
- One-way pattern
- Request/acknowledge pattern
- Stream pattern

Bei der Datenquelle kann es sich sowohl um von Hardware und auch von Software generierte Events handeln. Beispiele hierfür sind Temperatur, Lautstärke oder auch Browser-Clicks.

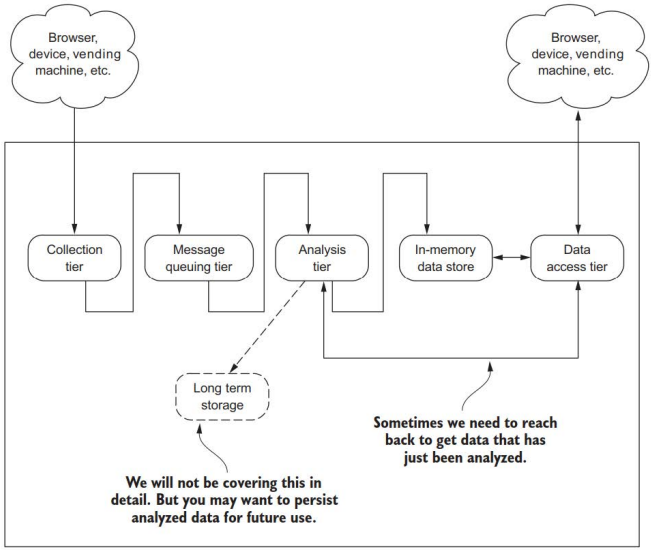

Abbildung 2: Streaming Data-Architektur [10]

Um die Daten vom Collection Tier auf den Rest der Pipeline zu verteilen wird ein Messaging Queueing Tier implementiert. Das Message Queueing Model ist ein Modell zur Interprozesskommunikation. Anwendungen schicken Nachrichten an eine Nachrichtenschlange, von der andere Anwendungen diese abholen können [11]. Dadurch werden die eingesetzten Systeme voneinander entkoppelt und die Kommunikation findet nicht durch direkte Aufrufe, sondern über die Queue statt [10]. Im vorliegenden Fall wird der Collection Tier vom Rest der Pipeline entkoppelt.

Im Analysis Tier werden auf die Streams des Message Queueing Tiers kontinuierlich Queries angewandt um Muster in den Daten zu erkennen. Hier können Event Stream Processing-Systeme (ESP) wie z. B. Apache Flink, Apache Spark oder Apache Storm oder auch Complex Event Processing-Systeme (CEP) wie z. B. Esper oder Apache Samza eingesetzt werden. [10] In ESP wird die Verarbeitungslogik imperativ umgesetzt, wohingegen in CEP einen deklarativen Ansatz verfolgt. Die Event Processing Languages (EPL) sind Sprachen, die über Sequenz-, Konjunktions-, Disjunktions- und Negationsoperatoren verfügen und für CEP entwickelt wurden [12]. Als Basis der Operatoren werden Windows eingesetzt. Ist die Analyse vorbei und die Ergebnisse stehen fest, können diese verworfen werden, zurück in die Streaming-Plattform gespeichert werden, für eine Echtzeitnutzung oder die Stapelverarbeitung gespeichert werden [10]. Wie der Abbildung 2 zu entnehmen ist, schlägt Psaltis hierfür eine In-Memory-Datenbank (IMDB), einen Data Access Tier und ein Long Term Storage vor.

B. Unsere Streaming Data-Architektur

In diesem Kapitel geht es um die Frage, welche Technologie wir in dem jeweiligen Tier einsetzen und wieso wir uns dafür entschieden haben. Technische Details, die die Implementierung betreffen, erläutern wir im nächsten Kapitel. TODO: Quellen zu den einzelnen Punkten/Systemen

- *Collection Tier.* Als Datenquelle haben wir Daten von Wikipedia. Konkret handelt es sich um die RecentChanges, also die aktuellen Änderungen an Wikipedia-Einträgen. Details hierzu sind im nächsten Kapitel beschrieben. Wir entschieden uns für Wikipedia als Datenquelle, weil es eine offene Plattform ist, die einen freien Zugang zu allen Daten gibt.
- *Messaging Queuing Tier.* Wir nutzen Kafka als Messaging-System, weil es skalierbar ist, einen hohen Datendurchsatz ermöglicht, Real-Time-Messaging unterstützt, eine einfache Client-Anbindung ermöglicht und Events standardmäßig persistiert. Der letzte Punkt ist besonders relevant, da für die Entwicklung der Regeln eine statische Datenmenge einfacher zu handhaben ist.
- *Analysis Tier.* Die Analyse der Wikipedia-EditEvents sollte deklarativ erfolgen. Dadurch erhofften wir uns in kurzer Zeit eine Vielzahl an Regeln zu testen und so ein gutes Verständnis für die Daten zu erhalten. Aufgrund dieser Vorgabe entschieden wir uns für Esper. Es bietet eine CEP-Implementierung, die nach einem regelbasierten Ansatz (somit deklarativ) arbeitet. Ein weiterer Punkt, der für Esper sprach, war zum einen die Esper Processing Language und die Implementierung der Anwendung in Java.
- *In-Memory Data Store, Long Term Storage und Data Access Tier.* Für diese drei Komponenten haben wir selbst keine Implementierung vorgesehen. Im Ausblick geben wir hierzu Ideen zu möglichen Erweiterungen.

IV. Prototyp

Zur Lösung der Aufgabenstellung haben wir einen lauffähigen Prototypen entwickelt, der die Machbarkeit demonstriert. Dafür haben wir die im vorhergehenden Kapitel genannten Technologien eingesetzt. Die Details zu den jeweils entstandenen Anwendungen stellen wir in diesem Kapitel vor.

Abbildung 3 zeigt die Teilsysteme und deren Interaktion untereinander.

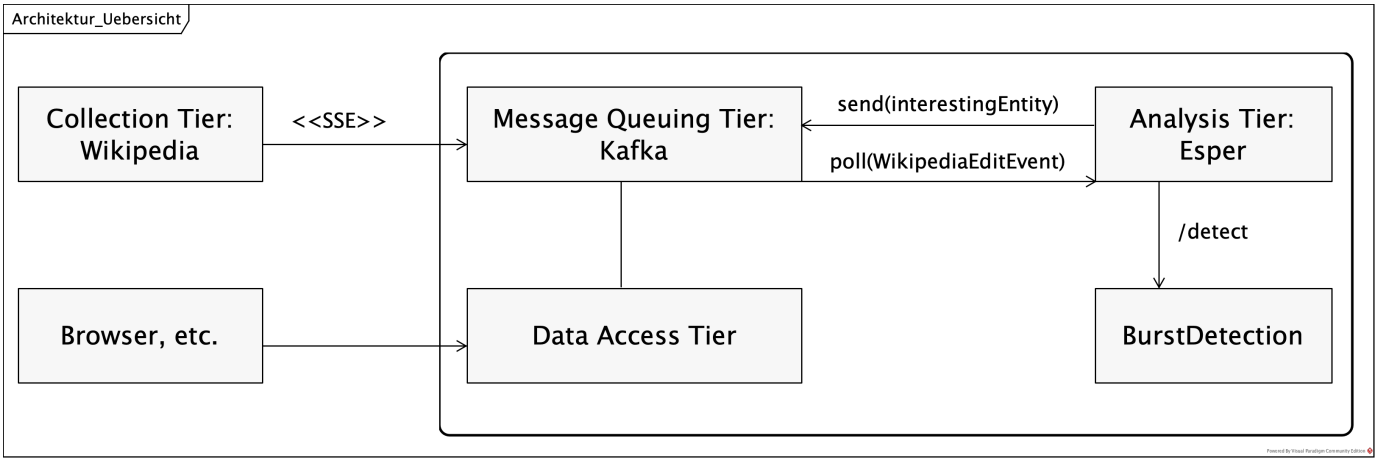

Abbildung 3: Architekturübersicht

A. *Collection Tier: Wikipedia*

Wie zuvor beschrieben, stützt sich unser System auf Daten von Wikipedia. Wir nutzen den RecentChanges-Stream, der Events zu neu erstellten, aktualisierten und gelöschten Wikipedia-Seiten enthält[2]. Der Quellcode 1 zeigt einen Ausschnitt eines solchen Events. Es sind nur die Attribut abgebildet, die in einem der nachfolgenden Teilsysteme relevant sind.

```
{
    "bot": false,
    "comment": "/* Politischer Werdegang
       */",
    "id": 269108829,
    "meta": {
        "uri": "https://de.wikipedia.org/
           wiki/Ingeborg_H%C3%A4ckel",
        "partition": 0,
        "offset": 1329388977
    },
    "timestamp": 1547910734,
    "title": "Ingeborg H\"ackel",
    "user": "Eszet2000",
    "wiki": "dewiki"
}
```

Quellcode 1: Wikipedia RecentChange-Event

Die beiden Parameter `offset` und `uri`, innerhalb des `meta`-Objektes, stammen aus einem Kafka-System und dienen der Wiederaufnahme eines abgebrochenen Streams. Wikipedia setzt intern Apache Kafka als Messaging-System ein. Nach außen nutzt Wikipedia EventStreams. Das ist ein Webservice der kontinuierliche Datenströme mit strukturierten Daten über HTTP sendet[3]. Die Basis-Technologie dessen, ist das Server-Sent Event (SSE) Protokoll. Der RecentChanges-Stream ist ein solcher Stream und kann über eine Client-Bibliothek konsumiert werden.

Wie im vorherigen Kapitel beschrieben, werden Daten vom Collection Tier mithilfe eines der aufgelisteten Pattern übertragen. SSE funktioniert nach dem One-way pattern, indem eine HTTP-Verbindung aufgemacht wird um Nachrichten zu empfangen [13].

B. *Implementierungsdetails zum Messaging Queuing Tier: Kafka*

Im Messaging Queuing Tier setzen wir Apache Kafka in der Version 2.1 ein, um die im Collection Tier beschriebenen Wikipedia-Events in unser eigenes Messaging-System zu überführen. Hierfür haben wir eine Java-Anwendung entwickelt, in der die folgenden Schritte nacheinander ausgeführt werden:

1) *Kafka Initialisierung.* Den Host des Bootstrap-Servers setzen um eine Verbindung zu erzeugen. Als Key- und Value-Serialisierer setzen wir jeweils den `StringSerializer` von Kafka ein. Das heißt, die Events werden als JSON-String in das Topic `wikiEdit` eingespeist. Zum Senden von Events erzeugen wir ein Producer-Objekt mit dem passenden Typ `Producer<String, String>`. An dieser Stelle war die Überlegung, anstelle eines String-Serialisierers für den Wert des Producers, einen eigenen Serialisierer für die WikipediaEditEvent-Klasse einzusetzen. Wir entschieden uns aber für den String-Serialisierer, da Wikipedia die Events auch als JSON-String überträgt und wir in Kafka selbst keine weiteren Operationen an den Daten vornehmen. Eine mögliche Operation könnte ein Filter sein. Aber das einzige Ziel von Kafka soll die Persistierung und Weitergabe von Daten sein und dafür ist keine Serialisierung in einem bestimmten Typen notwendig. Außerdem verschiebt es Komplexität aus der Kafka-Anwendung in die Consumer-Anwendungen.
2) *Erzeugung eines EventHandlers.* Für das Empfangen von EventSource-Nachrichten nutzen wir die Java-

[2]https://www.mediawiki.org/wiki/Manual:RCFeed
[3]https://wikitech.wikimedia.org/wiki/EventStreams

Bibliothek *okhttp-eventsource*[4]. Zur Verarbeitung der Events `onOpen`, `onClose`, `onMessage`, `onComment` und `onError` muss das Interface `EventHandler` von *okhttp-eventsource* implementiert werden.

3) *Erzeugung und Starten einer EventSource.* Mit der Stream-URI der Wikipedia-EventSource (für die RecentChanges ist das: https://stream.wikimedia.org/v2/stream/recentchange) und des implementierten EventHandler-Interfaces kann ein `EventSource`-Objekt erzeugt werden. Das Objekt dient dem Starten und Beenden eines EventSource-Streams. Die Daten werden dann, wie zuvor beschrieben, durch das SSE-Protokoll von Wikipedia an die Anwendung gesendet.
4) *Beim Eintreffen eines Events: Senden einer Nachricht in ein Kafka-Topic.* Tritt ein Wikipedia-Event auf, wird die `onMessage`-Methode des implementierten EventHandlers-Interface aufgerufen. Der zweite Parameter enthält die Daten des aufgetretenen Events. Der Zugriff auf die als JSON-String codierte Nachricht erfolgt über die `getDate()`-Methode. Diese Daten sendet die Anwendung, über den zuvor erzeugten Producer, ohne eine weitere Verarbeitung in das Kafka-Topic `wikiEdit`.

Die Konfiguration der Kafka-Topics ist sehr einfach gehalten, da es sich bei der Anwendung nur um einen Prototypen handelt. Wir haben ein Topic mit dem Namen `wikiEdit`. Da die Anwendung auf nur einem Server läuft, setzen wir auch nur eine Partition ein und haben keine Replikation. Eine Skalierung der Kafka-Anwendung auf mehrere parallelarbeitende Server ist jedoch nicht ausgeschlossen für eine Produktivanwendung.

C. Implementierungsdetails zum Analysis Tier: Esper

In unserer Esper-Anwendung, die das Hauptsystem des Analysis Tier ist, nutzen wir Esper in Version 7.1 als Complex Event Processing-Werkzeug. Zur Verarbeitung der Wikipedia-Events haben wir zwei Lösungen umgesetzt, da die erste Lösung nicht zum Erreichen der Ziele führte. Für ein besseres Verständnis geben wir einen kurzen Überblick über den gemeinsamen Aufbau beider Lösungen. Danach beschreiben wir die Details und Unterschiede der jeweiligen Lösungen und analysieren diese hinsichtlich der Zielerfüllung. In unserer Esper-Anwendung sind wir wie folgt vorgegangen:

1) *Esper Initialisierung.* Die Initialisierung von Esper besteht aus der Erzeugung einer `Configuration`, der Erstellung und dem Starten von EPL-Statements, sowie dem Erzeugen von Listener-Klassen.
2) *Kafka initialisieren und starten.* Um die Daten aus dem Messaging Queuing Tier zu bekommen haben wir einen `KafkaConsumer<String, String>` implementiert. Wir pollen in einer Endlosschleife alle 10 Millisekunden die Daten vom Kafka-Topic `wikiEdit`. Bei den empfangenen Daten handelt es sich um einen String, der JSON enthält. Mithilfe der Gson-Bibliothek[5] konvertieren wir die JSON-Strings in Java-Objekte. Die daraus resultierenden Java-Objekte senden wir wiederum in das Esper-System, damit darauf die EPL-Statements angewandt werden können.
3) *Aktion ausführen, sobald das Muster erfüllt ist.* Ist das Muster erfüllt, wird die `update`-Methode der Listener-Klasse aufgerufen. Es werden die alten und neuen Events übergeben. Daraus kann eine Aktion erfolgen, z. B. dass erzeugen eines neuen komplexen Events.

Abbildung 4

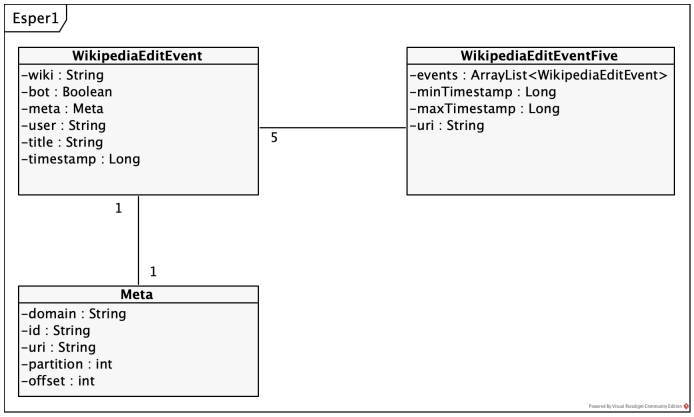

Abbildung 4: Klassendiagramm der zwei Ereignistypen WikipediaEditEvent, WikipediaEditEventFive und der Helferklasse Meta

Genaueres zu den erzeugten Expressions im nächsten Kapitel.

Einsatz von GSON

D. BurstDetection-Implementierung

V. Anwendungsfälle

A. Entworfene Anwendungsfälle

Die Zielsetzung, den Fokus auf das Erkennen von Ereignismustern in Wikipedai-Edit-Eventsstrom zu setzen, schränkt die Komplexität der Datenanlyse ein. Ebenso ist ein Hinzuziehen externer Informationen mit hohem Aufwand verbunden. Deswegen sind Abfragen externer Quellen, Textanalyse der Änderungen, Verortung der Seite in der Wikipedia-Hierarchie und Parsen externer Web-Seiten nur in geringer Frequenz möglich. Die Bildung komplexer Events erhöht der Informationsdichte und vergrößert die Granularität, verringt die Frequenz. Aufwändigeren Analysen werden daher an komplexe Events gebunden.

- Global Event Detection
  TODO
- Edit-Wars Detection
  Bei Edit-Wars stimmen die Vorstellungen über den Inhalt des Artikels der Autoren nicht überein. Die Folge sind

[4]https://github.com/launchdarkly/okhttp-eventsource

[5]https://github.com/google/gson

wiederholtes Revidieren der Änderungen, häufig in hoher Frequenz.

- Fraud Detection
  Ab wann Vandalismus
  Vorrausgesetzt Muster, die sich temporal entwickeln
- Load Prediction
  TODO
- Semantic-Clustering
  TODO

Schauen, was man aus dem Paper bekommt: [3]

1) Voraussetzung: 2 oder mehr Autoren (die kein Bot sind) bearbeiten ein Ergebnis:

VI. Praktische Analyse

- Wie sieht ein konkretes Wikipedia-Edit-Event aus / aus welchen Bestandteilen besteht es? siehe [3] Kapitel 2 Anfang
- Anhand von Burst Detection, wollen wir Events der realen Welt ableiten [14]
- Mit welchen Expressions decken wir welche Use Cases ab?
- Wie sind wir auf die Expressions gekommen? Nur durch ausprobieren?
- Reale Beispiele für "passendeËvents
- Welche neuen "Komplexen Eventsërzeugen wir?
- Reale Beispiele für komplexe Events
- Welche Ergebnisse liefert das System?
- Übersicht der Hierarchie von Ereignistypen: siehe EP_5_CEP_1ṗdf

VII. Ergebnisse

- Der Einsatz von Esper im Analysis Tier war für anfängliche Versuche richtig. Mit zunehmender Komplexität bekamen wir hierdurch Schwierigkeiten. Durch die Abstraktion, die Esper bietet konnten wir beispielsweise nicht auf vergangene Events zugreifen. Die Funktionalität, die in Flink unterstützt und dort ...

VIII. Diskussion

aa

IX. Ausblick

X. Beiträge der Autoren

Phillip Ginter und Daniel Schönle haben gleichermaßen zu dieser Arbeit beigetragen und sind Erstautoren.

Literatur

[1] Wikinews, "Interview mit jimmy wales: Wie geht es weiter mit wikipedia?" [Online]. Available: https://de.wikinews.org/wiki/Special:PermanentLink/577184%3Ftitle%3DInterview_mit_Jimmy_Wales:_Wie_geht_es_weiter_mit_Wikipedia%3F

[2] Wikipedia, "Wikipedia." [Online]. Available: https://de.wikipedia.org/wiki/Wikipedia

[3] M. Georgescu, N. Kanhabua, D. Krause, W. Nejdl, and S. Siersdorfer, "Extracting event-related information from article updates in wikipedia," in *Advances in Information Retrieval*, P. Serdyukov, P. Braslavski, S. O. Kuznetsov, J. Kamps, S. Rüger, E. Agichtein, I. Segalovich, and E. Yilmaz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 254–266.

[4] S. Gottschalk, E. Demidova, V. Bernacchi, and R. Rogers, "Ongoing events in wikipedia: a cross-lingual case study," in *Proceedings of the 2017 ACM on Web Science Conference*. ACM, 2017, pp. 387–388.

[5] B. Fetahu, A. Anand, and A. Anand, "How much is wikipedia lagging behind news," in *Proceedings of the ACM Web Science Conference*. ACM, 2015, p. 28.

[6] X. Liu, M. Wang, and B. Huet, "Event analysis in social multimedia: a survey," *Frontiers of Computer Science*, vol. 10, no. 3, pp. 433–446, 2016.

[7] Q. Huang, G. Cervone, and G. Zhang, "A cloud-enabled automatic disaster analysis system of multi-sourced data streams: An example synthesizing social media, remote sensing and wikipedia data," *Computers, Environment and Urban Systems*, vol. 66, pp. 23–37, 2017.

[8] E. Gilbert, S. Bakhshi, S. Chang, and L. Terveen, "I need to try this?: a statistical overview of pinterest," in *Proceedings of the SIGCHI conference on human factors in computing systems*. ACM, 2013, pp. 2427–2436.

[9] A. Weiler, M. Grossniklaus, and M. H. Scholl, "An evaluation of the run-time and task-based performance of event detection techniques for twitter," *Information Systems*, vol. 62, pp. 207–219, 2016.

[10] A. Psaltis, *Streaming Data: Understanding the Real-time Pipeline*. Manning Publications, 2017.

[11] J. Gray, *Interprocess Communications in Linux*. Prentice Hall PTR, 2003.

[12] U. Hedtstück, *Complex Event Processing: Verarbeitung von Ereignismustern in Datenströmen*, ser. eXamen.press. Springer Berlin Heidelberg, 2017.

[13] I. Hickson, "Server-sent events," October 2015 (Zugegriffen am: 04.02.2019). [Online]. Available: https://www.w3.org/TR/eventsource/

[14] Y. Zhu and D. Shasha, "Efficient elastic burst detection in data streams," in *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '03. New York, NY, USA: ACM, 2003, pp. 336–345. [Online]. Available: http://doi.acm.org/10.1145/956750.956789