# 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

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 [1].
- Aufgabenstellung
  - Für eine Entität (z. B. eine Person des öffentlichen Lebens) aus der Gesamtheite 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?

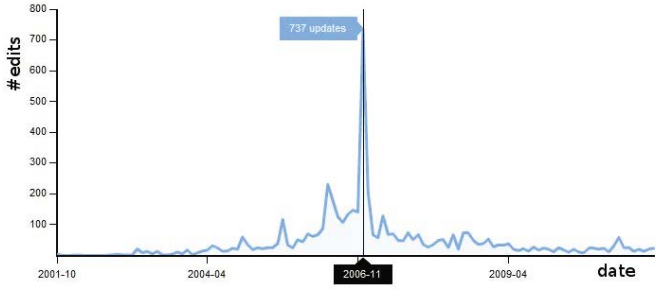


Abbildung 1: Donald Rumsfeld's Rücktritt führte zu einem Burst an Autoren, die einen Wikipedia-Edit vornahmen [1].

## 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

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

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 Ankunftshäufigkeitund eim Vergleich zur vorherigen Perioden ansteigt.
Efficient Elastic Burst Detection in Data Streams
[1]

## III. Streaming Data

Das Ziel unserer Aufgabenstellung ist die Verarbeitung von Streaming-Daten in Echtzeit. Die Streaming Data-Architektur von Psaltis [2] 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 [2] sie entwickelt hat. 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 [2]:

- 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.

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 Anwendnungen diese abholen können [3]. Dadurch werden die eingesetzten Systeme voneinander entkoppelt und die Kommunikation findet nicht durch direkte Aufrufe, sondern über die Queue statt [2]. Im

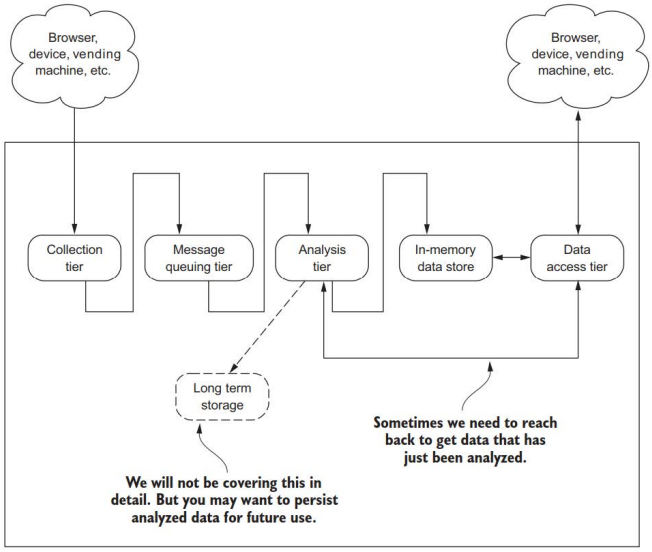


Abbildung 2: Streaming Data-Architektur [2]

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. [2] 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 [4]. 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 [2]. 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.

- *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 ein 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 Prototyp entwickelt, der die Machbarkeit demonstriert. Dafür haben wir die im vorhergehenden Kapitel genannten Technologien eingesetzt. Die Details zu den entstandenen Anwendungen stellen wir in diesem Kapitel vor.

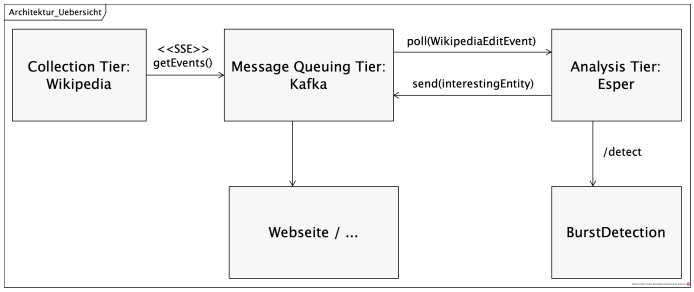


Abbildung 3: Architekturübersicht

Abbildung 3 zeigt bisher nur eine sehr rudimentäre Übersicht und soll als Grundlage dienen. Die Verbindung zwischen Esper und Kafka muss noch genauer werden: Protokoll, senden von komplexen Events in neue Kafka-Topics.

### A. Collection Tier: Wikipedia

- Was für Daten nutzen wir? WikipediaEditEvents
- Welche konkreten Daten hat ein WikipediaEditEvent?
- Wie sieht das System dahinter aus; wie verarbeitet Wikipedia die Daten: Kafka
- Was ist EventSource und was bringt es hier?
- Welches Pattern verwenden wir und wieso? Wie funktioniert das Pattern genau: siehe Psaltis https://goo.gl/qkgB8z

### B. Implementierungsdetails zum Messaging queuing tier: Kafka

Im Messaging Queuing Tier setzen wir Apache Kafka in der Version 2.1 ein, um die Wikipedia-Events aus dem Collection Tier in unser eigenes Messaging-System zu überführen. In der Java-Anwendung werden die folgenden Schritte nacheinander ausgeführt:

1) *Kafka Initialisierung.* Den Host des Bootstrap-Servers setzen. 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>`. TODO: Vor- und Nachteile für eine De-/Serialisierung von der eigenen WikipediaEditEvent-Klasse diskutieren
2) *Erzeugung eines EventHandlers.* Für das Empfangen von EventSource-Nachrichten nutzen wir die Java-Bibliothek *okhttp-eventsource*[2]. 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 und des implementierten EventHandler-Interfaces kann ein `EventSource`-Objekt erzeugt werden. Das Objekt dient dem Starten und Beenden eines EventSource-Streams.
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. Einer der beiden 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, in das Kafka-Topic `wikiEdit`.

TODO: - Die Konfiguration von Kafka: Welche Topics gibt es? Partitionen? Consumer Groups? Replication? Persistence? (oder alles schon im vorherigen Kapitel schreiben) - Vor- und Nachteile für eine De-/Serialisierung von der eigenen WikipediaEditEvent-Klasse diskutieren und der Einsatz von GSON?

### C. Implementierungsdetails zum Analysis tier: Esper

In unserer Esper-Anwendung, die Teil des Analysis Tier ist, nutzen wir Esper in Version 7.1 als Complex Event Processing-Werkzeug. Zur Verarbeitung der Wikipedia-Events implementieren wir die folgenden Schritte:

Genaueres zu den erzeugten Expressions im nächsten Kapitel.

1) *Esper Initialisierung.* Die Initialisierung von Esper besteht
2) *Expression erzeugen.*
3) `UpdateListener` *implementieren.*
4) *Kafka initialisieren.*
5) *Kafka Consumer erzeugen und in Endlosschleife Events pollen.*
6) *Empfangene Events auswerten.*

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

### D. BurstDetection-Implementierung

## V. Anwendungsfälle

Schauen, was man aus dem Paper bekommt: [1]

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 [1] Kapitel 2 Anfang
- Anhand von Burst Detection, wollen wir Events der realen Welt ableiten [5]
- 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

## VIII. Diskussion

aa

## IX. Ausblick

## X. Beiträge der Autoren

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

## Literatur


[1] 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.

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

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

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

[5] 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