https://github.com/ottogroup/SPQR

SPQR:
Ein Framework für die effiziente Ad-hoc-Datenstrom-Verarbeitung

Aktuelle Datenstrom-Frameworks erlauben den Aufbau kontinuierlich laufender Verarbeitungstopologien. Für Live-Analysen von Datenströmen durch Fachexperten sind sie jedoch zu statisch: Zum Informationsgewinn müssen diese Topologien ad hoc aufbauen, verändern und abreißen können. Mit SPQR stellen wir in diesem Artikel ein Open-Source-Framework für agile Datenstrom-Architekturen vor. Basierend auf Technologien aus dem High-Frequency-Trading-Umfeld und Kafka ermöglicht SPQR die Entwicklung neuer dynamischer Echtzeitanalyse-Systeme für Echtzeit-Datenströme.

Etablierte Datenstrom-Frameworks wie „Apache Storm“ (vgl. [Spa14]), „Apache Spark Streaming“ (vgl. [Spa14]) oder „Samza“ (vgl. [Sam14]) folgen einem statischen Ansatz zum Aufbau von Verarbeitungstopologien – das sind Operatorgraphen zur Datenstrom-Verarbeitung – in Rechen-Clustern:

- Die Topologien werden in einer Programmiersprache wie Scala oder Java programmiert.
- Der entwickelte Programmcode wird kompiliert und paketiert.
- Die resultierenden Build-Artefakte werden in einen Cluster verteilt.
- Schließlich werden die Topologien instanziiert und die Verarbeitung läuft an.

Die hieraus entstehende Latenz vom Zeitpunkt der Fertigstellung der Definition einer Topologie bis hin zu dem Zeitpunkt, wo diese zur Ausführung kommt, bemisst sich realistischerweise bei größeren Clustern in Minuten und nicht in Sekunden. Wenn auch nur eine einfache Änderung an der Topologie erfolgen soll, muss der gesamte Prozess erneut durchlaufen werden.

Diese Herangehensweise ist adäquat, wenn es darum geht, kontinuierlich laufende *ETL*-Prozesse (*Extract, Transform, Load*) für Datenströme zu realisieren. Bei der Otto Group verfolgen wir jedoch das Ziel, Fachbereiche in die Lage zu versetzen, auf Nahezu-Echtzeit-Datenströmen (deren Daten schnell an Wert verlieren) Nahezu-Echtzeit-Analysen durchführen zu können. In einem solchen Szenario sind die inhärenten Latenzen des statischen Ansatzes problematisch.

Zur Illustration dieses Punktes soll das so genannte „eCi-Live-Dashboard“ der Otto Group dienen. Am 8. Dezember 2014 gab der Blog „ottogroup unterwegs“ der Öffentlichkeit für 24 Stunden einen Live-Einblick in die Kaufaktivitäten der Kunden von 16 Unternehmen der Otto Group (vgl. [Häg14]). Dies geschah auf Basis der Klickströme dieser Online-Shops, die über eine dynamische Web-Oberfläche auf Basis von D3.js und Node.js visualisiert wurden (**siehe Abbildung 1**).

Abb. 1: Screenshot des eCi-Live-Dashboard.

Die Klickströme durchlaufen Datenstrom-Verarbeitungstopologien, die live die Orte von Käufen, aktuelle Suchbegriffe der Shop-Besucher, populäre Produkte sowie aktuell umgesetzte Warenkörbe ermitteln und diese Ereignisse an das Live-Dashboard über Web-Sockets weiterreichen.

Während eine solche Funktionalität durchaus auch mit statischen Topologien (z. B. auf Basis von Apache Storm) erreichbar wäre, hat das Live-Dashboard weitere Fähigkeiten, die über statische Topologien hinausgehen. Neben den in der Aktion gezeigten Widgets unterstützt es weitere Komponenten, die Analysten zu beliebigen Dashboards arrangieren und ad hoc umkonfigurieren können. So ist es zum Beispiel möglich, Shop- oder Produktkategorie-spezifische Dashboards auf- oder umzubauen. Um dies mit minimalen Latenzen für die Nutzer zu ermöglichen und unnötigen Verbrauch von Ressourcen für Topologien nicht mehr existierender Dashboard-Arrangements im Rechencluster zu vermeiden, müssen Topologien ad hoc instanziiert, umgebaut und nach Schließen eines Dashboards auch wieder abgebaut werden können.

Das von uns entwickelte und kürzlich als Open-Source veröffentlichte Framework für *Stream Processing and Querying in Real-Time (SPQR)*[1] legt die Grundlage für agile Datenstrom-Architekturen, wie sie in dem beschriebenen Szenario benötigt werden. SPQR stellt eine *dynamische* Laufzeitumgebung für Java zur performanten Verarbeitung hoch-volumiger und hoch-frequenter Datenströme mittels *Pipelines* zur Verfügung. SPQR ermöglicht die Ad-hoc-Instanziierung und -Modifikation von Datenstomverarbeitungs-Pipelines in

[1] Gesprochen „Spooker“.

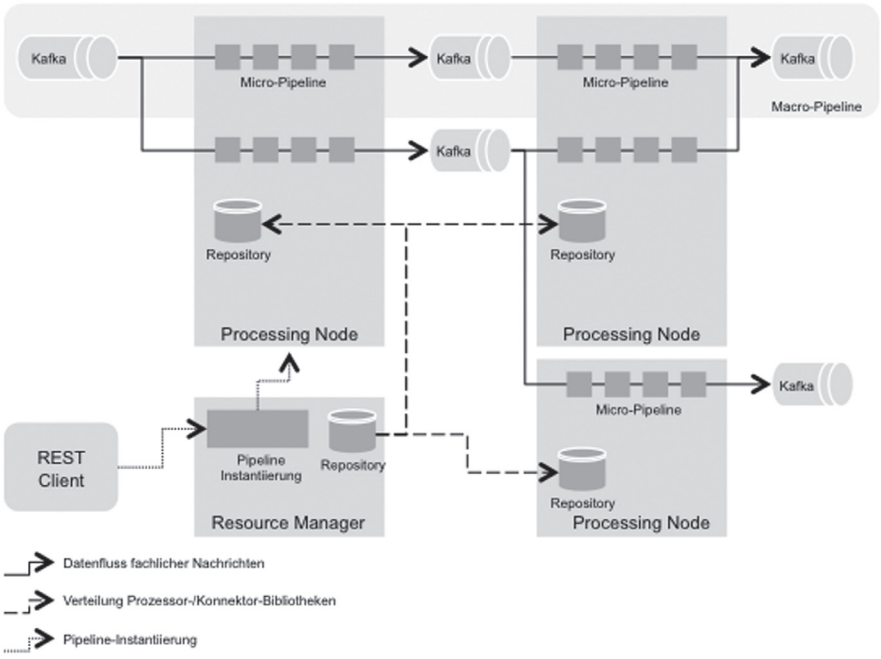

Abb. 2: SPQR-Architekturübersicht.

Sekundenbruchteilen.

Im Folgenden beleuchten wir zunächst die grundsätzlichen Architekturüberlegungen hinter SPQR. Danach geben wir einen Überblick über die SPQR-Architektur. Wir greifen das einführende Beispiel für den Einsatz von SPQR in der Otto Group noch einmal auf und schließen mit einem Ausblick.

Architekturüberlegungen

Unser Ziel ist eine agile Architektur, die es erlaubt, Datenstrom-Verarbeitungstopologien in einem Cluster ad hoc zur Laufzeit zu verändern. Vor diesem Hintergrund trifft SPQR die folgenden beiden grundlegenden Architekturentscheidungen.

Message-Orientierung

Ein klassischer Ansatz zur verteilten Datenstrom-Verarbeitung basiert auf Message-Broker-Systemen wie *Kafka* (vgl. [Kaf14]) oder *Java Messaging Service (JMS)*. Datenströme werden hier als Warteschlange implementiert und ihre Elemente als Nachrichten in den Warteschlangen. Die Verarbeitung erfolgt über externe Prozesse, die die Nachrichten konsumieren und ihre Transformationsergebnisse wiederum als Nachrichten in Warteschlangen publizieren. Neben dem Vorteil, dass man Skalierbarkeit und Fehlertoleranz auf den verwendeten Message-Broker abwälzen kann, ist dieser Ansatz auch hinsichtlich der gewünschten Ad-Hoc-Dynamik attraktiv: Nachrichten werden in den Warteschlangen gepuffert, sodass man die Prozesse zwischen diesen ohne Nachrichtenverlust umbauen kann.

Aus diesen Erwägungen heraus wählen wir für SPQR ebenfalls eine nachrichtenorientierte Architektur. Die Cluster-Knoten von SPQR kommunizieren miteinander in der Regel über Kafka, aber auch über andere Message-Broker, für die entsprechende Konnektor-Implementierungen vorliegen.

Trennung von Implementierung, Deployment und Instanziierung

Die Statik etablierter verteilter Datenstrom-Frameworks wie Storm, Samza oder Spark Streaming liegt darin begründet, dass sie vom Anwender das Durchlaufen des gesamten Prozesses der Implementierung einer Topologie zur Verarbeitung von Datenströmen inklusive Deployment und Instanziierung im Cluster verlangen, mit entsprechenden durch Code-Transport und Kommunikation entstehenden Latenzen.

Wenn jedoch der Code für Datenstrom-Operatoren auf den Cluster-Knoten schon vorläge, ließen sich diese sehr schnell instanziieren.

Aus dieser Erkenntnis heraus trennt SPQR die Schritte „Implementierung“, „Deployment“ und „Instanziierung“ strikt voneinander. Neue Datenstrom-Operatoren oder Konnektoren zu externen Systemen werden unabhängig von Cluster-Deployments entwickelt und getestet. Fertige Operator-Bibliotheken werden nach Abschluss der Entwicklungsarbeiten auf den Knoten eines SPQR-Clusters verteilt.

SPQR stellt nun eine Laufzeitumgebung auf den Cluster-Knoten bereit, die per REST-Web-Service das dynamische Starten von Operatoren mit individuellen Konfigurationen sowie deren Verknüpfung zu Topologien mit geringer Latenz ermöglicht. Laufende Topologien und Operatorinstanzen können per Service-Aufruf dynamisch umkonfiguriert und auch wieder abgebaut werden.

SPQR-Architektur

Aus diesen Überlegungen heraus entwickelten wir für SPQR die in **Abbildung 2** illustrierte Grobarchitektur.

Der Idee der *Message-Orientierung* folgend werden Datenstrom-Verarbeitungstopologien bei SPQR in Form so genannter *Pipelines* realisiert. Pipelines laufen auf den Cluster-Knoten in Laufzeitumgebungen, die *Processing-Nodes* genannt werden. Solche *Micro-Pipelines* innerhalb einer Processing-Node können über Node-Grenzen hinweg mittels des Message-Brokers Kafka zu *Macro-Pipelines* kombiniert werden.

Processing-Nodes können Pipelines ad hoc

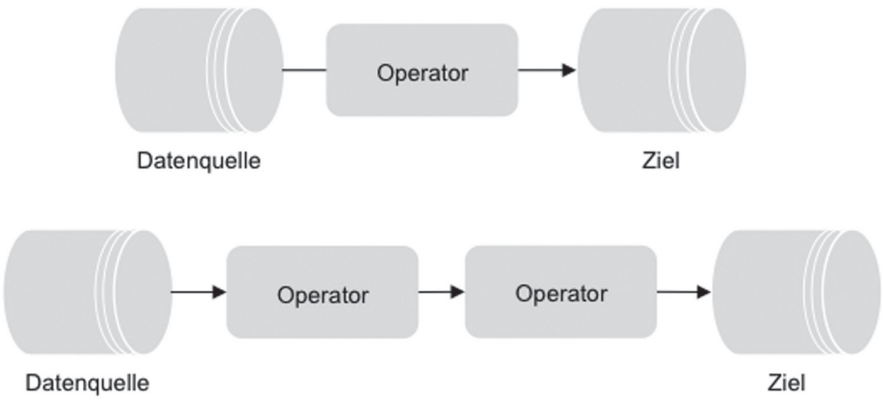

Abb. 3: Ein Micro-Pipeline.

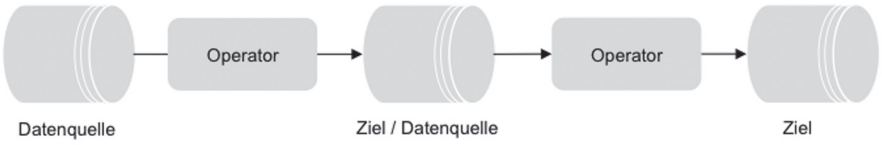

Abb. 4: Eine Macro-Pipeline.

instanziieren. Zu diesem Zweck verfügen sie über lokale *Repositories*, in denen – der Idee der *Trennung von Deployment und Instanziierung* folgend – Operator-Bibliotheken verwaltet werden.
Zentraler Ansprechpunkt für Clients in der SPQR-Architektur ist der *Ressourcen-Manager*. Dieser verteilt sowohl die Operator-Bibliotheken als auch die Pipeline-Instanziierungsanfragen zwischen den Processing-Nodes eines SPQR-Clusters. Im Folgenden stellen wir diese Konzepte näher vor.

Pipelines
Für die Abbildung von Datenstrom-Verarbeitungstopologien innerhalb eines Cluster-Knotens nutzt SPQR eine nachrichtenorientierte Micro-Pipeline-Metapher. Komponenten – zum Beispiel ein Datenstrom-Operator oder ein Konnektor zu einem externen System – werden in einer linearen Producer-Consumer-Kette angeordnet, um Datenströme – eine Nachricht pro Element – zu verarbeiten (**siehe Abbildung 3**).
Wenngleich es so möglich ist, Micro-Pipelines beliebiger Länge zu konstruieren, ist es sowohl zur Reduzierung der Komplexität als auch der mehrfachen Nutzung von Zwischenergebnissen wegen sinnvoll, lange Verarbeitungsketten in kleine, beherrschbare Teile aufzugliedern. Innerhalb von SPQR wird dieses Konzept durch Macro-Pipelines abgebildet, die sich aus einer Reihe von Micro-Pipelines zusammensetzen (**siehe Abbildung 4**).
In einer Macro-Pipeline dient der Zielpunkt einer Micro-Pipeline als Datenquelle anderer Micro-Pipelines. Da Datenquellen und Zielpunkte Nachrichten puffern, sorgt man so zum einen für eine Entkopplung von Verarbeitungsschritten. Zum anderen ist es so möglich, die Operatorketten zwischen Quelle und Ziel einer Micro-Pipeline zur Laufzeit ad hoc zu modifizieren: Sowohl die Ergebnisse der Pipeline vor der Änderung im Ziel als auch neu ankommende Nachrichten in der Quelle werden bis zum Abschluss der Modifikation gepuffert.
Ebenfalls kann mittels Macro-Pipelines über die Grenzen von Cluster-Knoten hinaus skaliert werden. Dies geschieht dadurch, dass die Quell- und Zielpunkte von Micro-Pipelines auf verschiedenen Knoten im Cluster über einen externen Transportmechanismus miteinander verbunden werden (**siehe Abbildung 5**).
SPQR macht keine Vorgaben hinsichtlich des Transportmechanismus: Es muss lediglich ein entsprechender Konnektor implementiert werden. Dies gibt dem Anwender die Möglichkeit, applikationsbezogen einen geeigneten Transportmechanismus zu wählen. Hierdurch wird die Menge der möglichen Einsatzszenarien von SPQR deutlich vergrößert, erlaubt dies doch eine gezielte Entscheidung für oder gegen bestimmte Funktionen einzelner Transportsysteme, wie z. B. Transaktionssicherheit, Durchsatz oder Persistierung von Nachrichten.
Im Rahmen interner Projekte kommt bei der Otto Group häufig Apache Kafka zum Einsatz. Hier spielen insbesondere folgende Aspekte eine Rolle:

- Persistenz von Nachrichten und die Möglichkeit, Nachrichten erneut ausspielen zu können.
- Erhalt der Nachrichtenreihenfolge.
- Sicherstellen der singulären Verarbeitung von Nachrichten auch bei steigender Konsumentenzahl.
- Hoher Durchsatz.
- Weitgehende horizontale Skalierbarkeit.

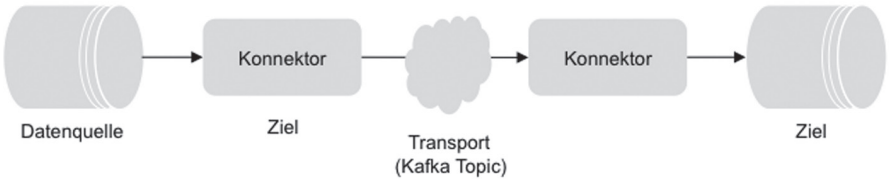

Abb. 5: Eine Macro-Pipelines über externen Transportmechanismus.

Aus diesem Grund stellt SPQR einen Standardkonnektor für Apache Kafka zur Verfügung. Wir können so bei der Entwicklung von SPQR die Problematik der Skalierbarkeit und Zuverlässigkeit ausblenden und uns auf die Skalierungs- und Zuverlässigkeitsmerkmale von Kafka verlassen.
Weiterhin existieren Konnektor-Implementierungen für den Konsum von Tracking-Events aus der Streaming-Schnittstelle der Firma Webtrends (vgl. [Web14]) und für das Schreiben von Nachrichten in einen ElasticSearch-Cluster (vgl. [Ela14]).

Operatoren und Konnektoren
Operatoren und Konnektoren sind zweifelsohne erfolgskritische Komponenten für SPQR. Erst die Möglichkeit, unterschiedliche Quellen und Ziele ansprechen zu können, erlaubt es, interessante Verarbeitungen durchzuführen. Ähnliches gilt auch für Operatoren, die schlussendlich die konkreten Analysefunktionen für Datenströme vorhalten.
Wenngleich das Framework bereits heute eine Anzahl von Operatoren und Konnektoren mitliefert, ist die Architektur so angelegt, dass anwendungsspezifische Implementierungen ohne viel Vorwissen realisiert werden können. Durch Abstraktion der eingesetzten Frameworks verlangt SPQR für die Umsetzung eigener Operatoren nur die Implementierung eines einzigen einfachen Callback-Interface mit einer einzigen Methode `onMessage()`, der jede an den Operator adressierte Nachricht im Datenstrom einzeln und asynchron übergeben wird und die – die wie auch immer gearteten – Ergebnisnachrichten eines Operators als Rückgabewert zurückliefert. Ein Beispiel für einen einfachen Operator zum Zählen durchlaufender Nachrichten und das Ersetzen des Inhalts durch einen neuen String zeigt **Listing 1**.
Die Realisierung eigener Konnektoren gestaltet sich ähnlich einfach und setzt lediglich voraus, dass die anzusprechenden Datenquellen und -ziele in der Lage sind, Daten auf Nachrichtenbasis bereitzustellen bzw. diese entgegenzunehmen.

Processing-Nodes
Die Processing-Nodes stellen innerhalb der SPQR-Architektur die Laufzeitumgebung

für Micro-Pipelines zur Verfügung. Um innerhalb eines SPQR-Cluster sichtbar und somit in der Lage zu sein, Anforderungen zur Instanziierung von Micro-Pipelines erhalten zu können, registriert sich jeder Processing-Node zu Beginn seines Lebenszyklus beim Ressourcen-Manager des Clusters. Technisch basiert dieser Schritt auf einem einfachen HTTP-POST-Request, der gegen die REST-Schnittstelle des Ressourcen-Managers abgesetzt wird. Unter anderem weist der Ressourcen-Manager dem Node eine eindeutige Kennung zu und übermittelt diese in seiner Antwort.

Jeder Processing-Node verfügt zudem über ein lokales Repository, in dem die deployten Operator- und Konnektor-Bibliotheken in Form von JAR-Dateien versioniert zur Instanziierung vorgehalten werden.

Eine solche Versionsverwaltung ist wünschenswert, da die Bibliotheken ständig für die Instanziierung zur Verfügung stehen sollen. Über einen längeren Zeitraum ergibt sich aber realistischerweise der Bedarf, diese weiterzuentwickeln – sei es um Fehler zu beheben oder Funktionalität zu erweitern. Unterschiedliche Versionen eines Operators oder Konnektors sollten somit gleichzeitig zu verwenden sein. Um dies zu realisieren, greift das Repository auf dedizierte Java-Class-Loader zurück, wobei pro geladener Bibliotheksversion eine Class-Loader-Instanz erzeugt wird. Dieser Mechanismus erlaubt es, auch unterschiedliche Versionen eines Operators gleichzeitig in verschiedenen oder sogar derselben Micro-Pipeline zu verwenden.

Als dritten zentralen Baustein stellt der Processing-Node einen Container zur Verfügung, der den Kontext zur Ausführung einer Micro-Pipeline liefert. Der Container führt Pipeline-Instanziierungen durch, überwacht diese und behandelt Fehlerfälle. Für die Implementierung wurde unter anderem eine Queue-Implementierung (vgl. [Thom14]) von Martin Thompson herangezogen, die auf hohen Durchsatz und geringe Latenz optimiert ist und eine optimierte Inter-Thread-Kommunikation ermöglicht. Daneben existiert eine Implementierung basierend auf dem OpenHFT-Chronicle-Framework (vgl. [Law14]), das maßgeblich von Peter Lawrey für den Einsatz im High-Frequency-Trading-Kontext implementiert wurde. Auch hier sind hoher Durchsatz und geringe Latenz von entscheidender Bedeutung.

Ressourcen-Manager

Im Gegensatz zum Processing-Node gibt es vom Ressourcen-Manager in der aktuellen Implementierung nur eine einzelne Instanz pro Cluster. Seine Aufgabe ist die Verwaltung und Überwachung sämtlicher Ressourcen – insbesondere Processing-Nodes – sowie die Verteilung von Anfragen zur Instanziierung von Pipelines auf Processing-Nodes.

Zudem orchestriert der Ressourcen-Manager das Deployment von Operatorbibliotheken. Zu diesem Zweck verfügt er wie die Processing-Nodes über ein lokales Bibliotheks-Repository.

Beim Deployment lädt man zunächst eine Bibliothek in das Repository des Ressourcen-Managers, der diese zu den registrierten Nodes weiterleitet. Meldet sich eine neue Node beim Ressourcen-Manager an, werden die registrierten Bibliotheken abgeglichen. **Abbildung 2** zeigt das Zusammenspiel zwischen Ressourcen-Manager und Processing-Nodes.

In der aktuellen Implementierung erfolgt die Zuweisung von Micro-Pipelines zu Nodes noch zufällig, um eine Lastverteilung zu erreichen. Ebenfalls ist die aktuelle Fehlerbehandlung nur rudimentär: Verschwindet eine Node, werden deren Pipelines in einer anderen Node reinstanziiert.

Zukünftig wollen wir intelligentere lastabhängige Zuweisungen und Fehlerbehandlungen in elastischen Container-Frameworks wie YARN oder Mesos ermöglichen.

```
/**
 * @see DirectResponseOperator#onMessage(StreamingDataMessage)
 */
public StreamingDataMessage[] onMessage(StreamingDataMessage message) {
 messageCount++;
 message.setBody("msg #"+messageCount+": " + message.getBody());
 return new StreamingDataMessage[]{message};
}
```

Listing 1: Codebeispiel für einen Operator.

SPQR-Anwendung: Live-Dashboard

Das oben erwähnte eCI-Live-Dashboard nutzt SPQR, um die für das jeweilige Widget-Arrangement notwendigen Verarbeitungs-Pipelines ad hoc zu instanziieren, umzukonfigurieren und mit dem Schließen des Dashboards abzubauen.

Bei der Verwendung in der Datenversorgung des eCi-Live-Dashboards hat sich SPQR als äußerst performant erwiesen. Die Klickstrom-Rate von ca. 500 Events pro Sekunde konnte problemlos auf einer einzelnen virtuellen Maschine mit vier virtuellen Prozessorkernen und 8GB Hauptspeicher für das Dashboard verarbeitet und aggregiert werden. Auf derselben Maschine liefen ebenfalls Kafka und die auf Node.js basierende Dashboard-Web-Oberfläche.

Im Rahmen eines synthetischen Tests – bestehend aus einer Sequenz von drei Komponenten (Zufallsgenerator, Identitäts-Operator und Empfänger) – konnten wir auf der gleichen Hardware eine Verarbeitungsrate

Links

[Ela14] Elasticsearch, siehe: http://www.elasticsearch.org/

[Git14] GitHub, ottogroup / SPQR, 2014, siehe: https://github.com/ottogroup/SPQR

[Häg14] R. Hägelen, E-Commerce in Echtzeit – eine kleine Bilanz, siehe: http://www.ottogroupunterwegs.com/blog/blog/posts/E-Commerce-in-Echtzeit-eine-kleine-Bilanz.php

[Kaf14] Apache Kafka, siehe: http://kafka.apache.org/

[Sam14] Samza Apache Incubator, siehe: http://samza.incubator.apache.org/

[Spa14] Apache Spark Webseite, siehe: http://spark.apache.org/

[Sto14] Apache Storm Webseite, siehe: http://storm.apache.org/

[Thom14] M. Thompson GitHub-Repository, siehe: https://github.com/mjpt777/

[Web14] Webtrends Website, siehe: http://www.webtrends.com/

[Law14] OpenHFT GitHub-Repository, siehe: https://github.com/OpenHFT/

von 5.000.000 Nachrichten pro Sekunde nachweisen.

Ausblick

Mit dem SPQR-Framework für ad hoc veränderbare Datenstrom-Verarbeitungs-Pipelines haben wir den Grundstein für die Echtzeitanalyse von Datenströmen durch die Fachbereiche gelegt.

Wir haben den Quellcode von SPQR auf GitHub publiziert (vgl. [Git14]) und stellen diesen der Allgemeinheit unter der Apache-Open-Source-Lizenz zur Verfügung. Wir sehen folgende anspruchsvolle Baustellen, an denen wir gerne zusammen mit der Community arbeiten möchten:

- Bislang müssen Nodes in einem Cluster manuell verteilt und gestartet werden. Wir möchten den Ressourcen-Manager in die Lage versetzen, elastisch nach Last Nodes allokieren und wieder deallokieren zu können. Hierfür möchten wir ein Container-Framework wie YARN oder Mesos nutzen.
- SPQR stellt zum jetzigen Zeitpunkt lediglich eine Ablaufumgebung für Datenstrom-Verarbeitungsoperatoren zur Verfügung, die für jede Anwendung erst noch implementiert werden müssen. Mit den Erfahrungen aus unseren Anwendungsfällen möchten wir eine Bibliothek von Standardoperatoren entwickeln, die sofort nutzbar sind.
- Um das Ziel einer direkten Analyse von Datenströmen durch Fachbereiche zu erreichen, bedarf es ebenfalls einer graphischen Darstellung der Verarbeitungstopologien für SPQR – ähnlich zu Modellierungswerkzeugen im Kontext von ETL oder EAI –, die den aktuellen Aufbau und Zustand von Pipelines intuitiv visualisiert, deren Manipulation leicht macht und das Abgreifen und die Präsentation von Analyseergebnissen ermöglicht. ||

Die Autoren

|| Christian Kreutzfeldt
(christian.kreutzfeldt@ottogroup.com)
ist Senior Solution Developer im Bereich Business Intelligence der Otto Group. Er entwirft und implementiert Softwarelösungen zur Verarbeitung großer Datenmengen im Stream-Processing-Kontext.

|| Dr. Utz Westermann
(gerdutz.westermann@ottogroup.com)
ist Senior Data-Architect im Bereich Business Intelligence der Otto Group. Er arbeitet an der Vereinnahmung, Speicherung und Verarbeitung großer Datenmengen, unter anderem für die Web-Analyse mit Technik aus dem Umfeld Messaging, NoSQL, Hadoop und Spark.