h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Hochschule Darmstadt
-Fachbereich Informatik-

Evaluierung der Qualität von Open Source Stream Processing Frameworks

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

vorgelegt von

Simon Kern
737708

Referentin:	Prof. Dr. Uta Störl
Korreferentin:	Prof. Dr. Inge Schestag
Ausgabedatum:	16.03.2015
Abgabedatum:	16.09.2015

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, den 16.09.2015

Simon Kern

Kurzfassung

Evaluierung der Qualität von Open Source Stream Processing Frameworks

Big Data ist im Bereich der Informationstechnologie aktuell eines der großen Schlagworte. Technologisch bezieht sich der Begriff meist auf die Stapelverarbeitung von gesammelten Daten. Im Gegensatz dazu verarbeiten Stream Processing Frameworks Datenströme direkt. Innerhalb der Open-Source-Community sind mehrere konkurrierende Systeme für diesen Aufgabenbereich entstanden. Bei der Neuentwicklung einer Anwendung muss jedoch ein Framework als Basis ausgewählt werden. Ziel der Arbeit ist es daher, aktuelle Open Source Stream Processing Frameworks hinsichtlich ihrer Qualität zu evaluieren, um diese Auswahl zu unterstützen.

Hierzu stellt diese Arbeit eine Referenzarchitektur vor, anhand derer die Komponenten eines Stream Processing Frameworks zur Verbesserung der Vergleichbarkeit zugeordnet werden können. Mithilfe des Evaluierungsprozesses nach ISO 25000 und insbesondere des Softwarequalitätsmodells nach ISO 25010 werden Kriterien zur Evaluierung von Stream Processing Frameworks entwickelt. Zur Erhöhung der praktischen Relevanz werden die Methoden Cognitive Walkthrough und Cognitive Dimensions zur Feststellung der Benutzerfreundlichkeit genutzt. Dafür werden Aufgaben definiert, die mithilfe der ausgewählten Frameworks implementiert werden. Sie werden zusätzlich zur Durchsatz- und Latenzmessung herangezogen. Als Rahmen der Aufgaben dient die Beobachtung von Wertpapieren, die an der New York Stock Exchange gelistet sind.

Für die Evaluierung werden Apache Flink, Apache Samza, Apache Spark und Apache Storm als Kandidaten identifiziert. Sie werden anhand der entwickelten Kriterien verglichen. Dabei zeigt sich, dass kein Framework alle Anforderungen vollständig abdeckt. Vielmehr müssen die Ergebnisse dieser Arbeit nach den konkreten Anforderungen einer Neuentwicklung gewichtet werden, um eine Auswahlentscheidung zu treffen.

Abstract

Evaluation of the Quality of Open Source Stream Processing Frameworks

Big Data is a huge trend in current information technology. This term mostly refers to software that does batch processing of stored data. But there is also software, namely stream processing frameworks, which operates directly on data streams. Several competing projects have emerged in the open source community, with the goal to provide software that addresses this issue. When developing a new application, one has to choose between these alternatives. Therefore, the goal of this thesis is to evaluate the quality of open source stream processing frameworks, in order to support decision making.

For this purpose, a reference architecture is proposed. Stream processing frameworks are composed of components that can be mapped onto this reference architecture. This facilitates an easy comparison of multiple frameworks and establishes a common understanding of terminology. The evaluation criteria are developed according to the process described in ISO 25000 and especially derived from the software quality model specified in ISO 25010. To increase the practical relevance of this evaluation, the usability inspection methods cognitive walkthrough and cognitive dimensions are used. These methods are based on predefined tasks that are implemented utilizing the frameworks under test. This implementation is later used to benchmark the frameworks in terms of throughput and latency. The tasks are built around the observation of stock pricing at the New York Stock Exchange.

The candidates for the evaluation are Apache Flink, Apache Samza, Apache Spark and Apache Storm. They are evaluated according to the developed criteria. The results show that no candidate meets all requirements. In fact, the results of this thesis have to be weighted to match the specific requirements of the application to be implemented, to support the decision making.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

1 Einführung

Aktuell ist *Big Data* eines der großen Schlagworte im Bereich der Informationstechnologie. Es deckt einen großen Themenbereich ab und bezieht sich nicht nur auf große Datenmengen sondern umfasst noch weitere Aspekte [BIT12, S. 7 ff.]. Weitgehend unumstritten sind die drei Vs [Lan01; BIT12]:

Volume Große, wachsende Datenmengen (> Terabyte)

Variety Große Vielfalt der Daten und Formate

Velocity Hohe Geschwindigkeit der Datengenerierung

Für den aktuellen Big Data Trend gibt es mehrere Gründe. In Unternehmen gibt es ein wachsendes Datenaufkommen, vor allem von unstrukturierten Daten abseits der klassischen transaktionalen Systeme, die sich mit etablierten Werkzeugen nur schwer analysieren lassen. Außerdem erhoffen sich Unternehmen durch die Integration verschiedener Datenquellen eine umfassende Kundensicht. Nicht zuletzt wird die Entwicklung auch von technischen Neuerungen angetrieben, die im Folgenden kurz beschrieben werden. [BIT14, S. 100 ff.]

Zur Bewältigung der Menge und Vielfalt an Daten, wurde initial das *MapReduce*-Paradigma [DG04] entwickelt. Es begegnet den steigenden Datenmengen mit massiver Parallelisierung mithilfe einer verteilten Architektur auf Basis von Standardhardware. Aufbauend auf dem MapReduce-Paradigma erlangte insbesondere das Werkzeug Apache Hadoop weite Verbreitung [Apa15u] bei der *Batch*-Verarbeitung im Big Data Umfeld. Batch-Verarbeitung bezeichnet dabei den Vorgang, Daten über einen bestimmten Zeitraum zu sammeln und anschließend stapelweise zu verarbeiten [Bib15]. Der Geschwindigkeitsaspekt ist bei Batch-basierten Systemen bisher nachrangig gewesen [BIT14, S. 52]. Es gibt eine Reihe von Anwendungsfällen, bei denen die Geschwindigkeit der eingehenden Daten eine besondere Rolle spielt: unmittelbare Reaktion auf ankommende Informationen, Aggregation oder Transformation von ankommenden Daten [BIT14, S. 53]. Diesen Bereich decken *Stream*

Processing Frameworks ab. Sie verarbeiten kontinuierlich eingehende Daten und führen eine beliebige Anwendungslogik aus.

1.1 Motivation

Während bei den Batch-Systemen Hadoop als etabliert gilt [BIT12, S. 27], hat sich im Bereich der Stream Processing Frameworks noch kein Produkt als allgemeine Basis herausgebildet. Es existiert eine Vielzahl von proprietären und quelloffenen Stream Processing Frameworks, die sich im Hinblick auf verschiedene Kriterien unterscheiden. Allen Frameworks ist gemein, dass sie das Entwickeln und den Betrieb von Anwendungen erleichtern, indem sie essentielle Funktionen, wie beispielsweise die Verteilung der Anwendung über mehrere Knoten, übernehmen. Es existieren aber auch Unterschiede zwischen den Frameworks: etwa bei den Programmiermodellen, der Fehlertoleranz oder den Integrationsmöglichkeiten.

Da sich noch kein Framework als Quasi-Standard etabliert hat und einige Unterschiede zwischen den Frameworks bestehen, muss bei Neuentwicklungen von Anwendungen im Bereich Stream Processing geprüft werden, welches Framework für die Entwicklung und den Betrieb am besten geeignet ist. Eventuell lassen sich je nach Anwendungsfall schon anhand der Funktionen Frameworks vom Vergleich ausschließen. Andere Kriterien lassen sich schwer nur anhand der Dokumentation bewerten, etwa die Benutzerfreundlichkeit der Programmierschnittstelle oder des Betriebs. Es existieren zwar verschiedene Vergleiche [LIX14; Boc14; Apa15ad; Nab+14], allerdings nutzen diese Arbeiten hauptsächlich die Dokumentationen der Frameworks als Informationsquelle.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, aktuelle Stream Processing Frameworks im Hinblick auf deren Qualität zu evaluieren. Hier liegt der Fokus des Vergleichs auf der praktischen Einsatzfähigkeit der Frameworks. Es werden ein Kriterienkatalog, eine Beispielanwendung und geeignete Methoden erarbeitet, anhand derer aktuelle Stream Processing Frameworks verglichen werden. Die Ergebnisse sollen die Entscheidungsfindung bei der Auswahl von Frameworks für Neuentwicklungen unterstützen und als Leitfaden für die Evaluierung von künftigen Frameworks dienen.

1.3 Aufbau der Arbeit

Zu Beginn wird kurz das Unternehmen Infomotion vorgestellt, mit dem diese Arbeit in Zusammenarbeit entstanden ist. Anschließend werden im Kapitel Grundlagen einige zentrale Begriffe wie Evaluierung und Qualität erläutert. Zusätzlich werden die Grundlagen von Information Flow Processing, dem Überbegriff von Stream Processing dargelegt.

Das folgende Kapitel Stream Processing Frameworks beschäftigt sich mit den zugrundeliegenden Modellen, Anforderungen und Prinzipien. An dieser Stelle wird eine Referenzarchitektur erarbeitet, in die später die konkreten Frameworks eingeordnet werden.

Anschließend wird das Konzept der Arbeit erläutert. Die untersuchten Frameworks und die zur Evaluierung verwendeten Methoden werden beschrieben. Mithilfe des vorgestellten Softwarequalitätsmodells werden die zu untersuchenden Kriterien abgeleitet. Um die praktischen Aspekte eines Frameworks zu evaluieren, werden Aufgaben ausgearbeitet, die im nächsten Kapitel implementiert werden.

Das Kapitel Praktische Umsetzung beschreibt zunächst die Entwicklungs- und Betriebsumgebung. Anschließend wird die Architektur der Implementierung vorgestellt und letztlich die Umsetzung der Aufgaben in den Frameworks beschrieben.

Im danach folgenden Kapitel werden die Ergebnisse für jedes Kriterium und Framework beschrieben und pro Kriterium in einer Tabelle zusammengefasst.

Abschließend werden die Ergebnisse dieser Arbeit im Kapitel Zusammenfassung kurz dargestellt und diskutiert.

2 Vorstellung des Unternehmens

Diese Masterarbeit wurde in Zusammenarbeit mit dem Unternehmen Infomotion GmbH in Frankfurt am Main verfasst. Infomotion zählt zu einem der führenden deutschen Beratungsunternehmen im Bereich Business Intelligence [Fre14]. Das Unternehmen beschäftigt insgesamt etwa 200 Mitarbeiter am Hauptsitz in Frankfurt am Main sowie an acht weiteren Standorten in Deutschland, Österreich und der Schweiz. Die Beratung der über 250 Kunden in verschiedenen Branchen erfolgt herstellerunabhängig und zugeschnitten auf die inhaltlichen und technischen Anforderungen des Kunden. [End15]

Für Infomotion besteht der Mehrwert dieser Arbeit in der Einordnung der aktuellen Stream Processing Frameworks anhand der verschiedenen evaluierten Kriterien. Dies erleichtert bei Neuentwicklungen die Entscheidungsfindung, welches Framework eingesetzt werden soll. Gleichzeitig dienen die erarbeiteten Kriterien und Methoden als Leitfaden, um künftige Frameworks zu evaluieren. Die implementierten Anwendungsfälle dienen als Beispiel, wie die Frameworks benutzt und konfiguriert werden können. Sie bieten einen leichten Einstieg in die Benutzung eines Frameworks und die Implementierung einer eigenen Anwendung.

3 Grundlagen

Dieses Kapitel erläutert die notwendigen Voraussetzungen für die nachfolgende Konzeption und Durchführung der Evaluierung. Dabei werden zunächst die allgemeinen Grundlagen einer Evaluierung dargestellt. Anschließend wird der Evaluierungsprozess von Softwareprodukten vertieft ausgeführt. Danach wird die technische Grundlage, das Stream Processing, beleuchtet. Zuletzt wird der Begriff Framework kurz definiert.

3.1 Evaluierung

Es gibt eine Vielzahl an Definitionsversuchen zum Begriff „Evaluierung“ (synonym zu Evaluation) [WT03, S. 13]. Die Gesellschaft für Evaluation definiert Evaluierung als „systematische Untersuchung des Nutzens oder Wertes eines Gegenstandes.“ [Gla08, S. 15]. Sie definiert eine Reihe von Merkmalen [Gla08, S. 17] unter denen vier besonders entscheidend für ein erfolgreiches Evaluierungsvorhaben sind [Kro01, S. 3]: der Gegenstand, der Evaluator, die Kriterien und das Verfahren.

Den Gegenstand einer Evaluierung zu definieren erscheint unproblematisch. Allerdings ist es nicht ausreichend nur das Objekt der Evaluierung zu definieren, da eine umfassende Evaluierung in jeder erdenklichen Hinsicht schwer zu leisten ist und meist auch nicht angestrebt wird. Daher ist der Gegenstand nicht das Objekt an sich, sondern spezielle Teilaspekte, die im Bezug zu den Zielen der Evaluierung stehen. [Kro01, S. 3-5]

Der Evaluator beziehungsweise das Evaluierungsteam ist die ausführende Instanz der Evaluierung. Den Beteiligten obliegt die Erarbeitung der Kriterien, die Informationsbeschaffung, die Ausführung der Evaluierung und die Ableitung von Konsequenzen. Die Aufteilung dieser Funktionen sollte zwischen den Beteiligten verbindlich erfolgen. [Kro01, S. 5]

Die Kriterien der Evaluierung werden im Vorfeld festgelegt. Dies gewährleistet eine nachvollziehbare, überprüfbare und kritisierbare Bewertung. Kriterien können aus verschiedenen Quellen stammen. Im Sinne der Nachvollziehbarkeit bieten sich etablierte Standards als Grundlage für die Erarbeitung der Kriterien an. [Kro01, S. 5-6]

Das Verfahren der Evaluierung muss auf den Gegenstand angepasst sein. Es gibt verschiedenste Ansätze zur Informationsgewinnung, dem Forschungsdesign und dem Zeitpunkt der Evaluierung. Bewährte Verfahren sind durch ihre größere Nachvollziehbarkeit auch hier zu bevorzugen. Ein etabliertes Evaluierungsverfahren für Softwareprodukte wird mit dem Standard ISO 25000 bereitgestellt. Dieser wird im nachfolgenden Unterkapitel ausführlich beschrieben. [Kro01, S. 6]

3.2 ISO 25000

Die Internationale Organisation für Normung (ISO) bietet zur Evaluierung von Softwareprodukten die Normenreihe 25000 an: „Software engineering – Software product Quality Requirements and Evaluation (SQuaRE)“ [ID05].

> „Zweck des SQuaRE-Satzes Internationaler Normen ist es, diejenigen durch die Angabe und Bewertung von Qualitätsanforderungen zu unterstützen, die Softwareprodukte entwickeln und erwerben. Er führt Kriterien für die Angabe von Qualitätsanforderungen an Softwareprodukte, deren Messung und Bewertung ein.“ [ID05, S. 5]

Qualität ist dabei als „Fähigkeit eines Softwareproduktes, angegebene und verdeckte Bedürfnisse zu befriedigen, wenn es unter den festgelegten Bedingungen genutzt wird“ [ID05, S. 14] definiert. Die Normenreihe ist in mehrere Einzelnormen gegliedert und bearbeitet alle Bestandteile eines Evaluierungsprozesses. Zunächst wird ein Qualitätsmodell (ISO 25010 [ISO11a]) mit einer Hierarchie an Charakteristiken definiert, „das eine Rahmenstruktur für die Bestimmung qualitätsbezogener Anforderungen und die Bewertung von Qualität bereitstellt.“ [ID05, S. 13]. Anschließend wird diese Rahmenstruktur durch konkrete Maße zur Messung der Qualität gefüllt (ISO 25020). Danach wird ein Evaluierungsprozess (ISO 25040 [ISO11b]) beschrieben, der auf die jeweilige Zielgruppe angepasst ist.

Das Qualitätsmodell aus ISO 25010 liefert einen Überblick, welche Charakteristiken zur Feststellung der Softwarequalität zu berücksichtigen sind. Diese Charakteristiken werden mittels Subcharakteristiken weiter verfeinert. Der Standard gibt acht fundamentale

Charakteristiken vor, die an dieser Stelle mitsamt ihrer Definition aufgelistet sind [ISO11a, S. 10 ff.]:

Functional Suitability „degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions“

Performance Efficiency „performance relative to the amount of resources used under stated conditions“

Compatibility „degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment“

Usability „degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use“

Reliability „degree to which a system, product or component performs specified functions under specified conditions for a specified period of time“

Security „degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization“

Maintainability „degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers“

Portability „degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another“

Anhand dieser Charakteristiken werden die konkreten Maße in Abhängigkeit vom Evaluierungsziel und dem Gegenstand der Evaluierung gebildet. Zusätzlich zu den Charakteristiken gibt es noch *Assigned Properties*, die keine Eigenschaften des Softwaresystems an sich sind, sondern durch externe Gegebenheiten zugewiesen werden. Sie sind daher keine Qualitätscharakteristik des Softwaresystems, da sie sich ändern können, ohne das die Software verändert wird. Nichtsdestotrotz sind sie für eine ganzheitliche Evaluierung zu betrachten. [ISO11a, S. 30]

Der Evaluierungsprozess aus ISO 25040 liefert ein detailliertes Referenzmodell, das an die Grundlagen aus Unterkapitel 3.1 anknüpft. Demnach werden zuerst die Anforderungen

an die Evaluierung definiert. Das umfasst das Ziel der Evaluierung, die Qualitätsanforderungen, den Umfang, den Fokus und die Strenge der Evaluierung. Darauf aufbauend werden die konkreten Maße und Kriterien spezifiziert. Auf der Grundlage der Spezifikation wird die Ausführung der Evaluierung geplant. Im nächsten Schritt wird die Evaluierung anhand des Plans durchgeführt. Abschließend werden die erfassten Daten ausgewertet, bewertet und in einem Bericht festgehalten. [ISO11a, S. 18-27]

Neben den Grundlagen der Evaluierung ist für das Verständnis der Arbeit die Kenntnis des technischen Umfelds von Stream Processing Frameworks nötig, die das folgende Unterkapitel vermittelt.

3.3 Information Flow Processing

Der Begriff *Information Flow Processing* beschreibt einen Anwendungsbereich, in dem Informationen aus mehreren Quellen gesammelt und möglichst schnell weiterverarbeitet werden, um Wissen zu extrahieren [CM12, S. 3]. Der eingehende Datenstrom wird zu diesem Zweck kontinuierlich verarbeitet [AGT14, S. 33]. Möglichst schnell bedeutet in diesem Zusammenhang, abhängig vom Anwendungsfall, entweder ein möglichst hoher Durchsatz oder eine möglichst geringe Latenz. Durchsatz beschreibt die Menge der verarbeiteten Informationen pro Zeiteinheit. Latenz steht für die vergangene Zeit zwischen dem Auftreten einer neuen Information und dessen vollständiger Verarbeitung. Es existieren mehrere technische Ansätze um Information Flow Processing zu implementieren [SÇZ05, S. 1]: Traditionell werden Applikationen für die schnelle Verarbeitung von großen Datenmengen als Individualsoftware entwickelt. Dies führt zu teuren, unflexiblen und schwer wartbaren Lösungen [SÇZ05, S. 2; Aba+03, S. 1].

Ein erster generischer Ansatz waren *Active Databases*. Sie erweitern vormals passive *Datenbankmanagementsysteme* um einen Mechanismus, der auf eingehende Daten reagieren kann. Dazu werden *Event-Condition-Action*-Regeln benutzt. Sie reagieren auf Ereignisse wie etwa Insert-Anweisungen. Die konfigurierten Bedingungen werden anhand des Ereignisses geprüft und im Erfolgsfall wird eine Aktion ausgeführt [MD89]. Dieses Konzept ist vergleichbar mit den in *Structured Query Language (SQL)* realisierten *Triggern* [ISO99a, S. 90]. Active Databases zeigen jedoch Limitationen bei hohen Datenraten und beim Programmiermodell [AGT14, S. 41].

Ein weiterer Ansatz sind *Continous Queries* [Ter+92]. Statt einem Trigger werden normale SQL-Anweisungen genutzt, die inkrementell auf neue Datensätze angewendet werden. Es resultiert ein Datenstrom aus den Ergebnissen der Abfrage. Continous Query Systeme sind hauptsächlich auf eine effiziente Abfrage und weniger auf die Datenhaltung ausgelegt, daher sind sie in dieser Hinsicht leistungsfähiger als Trigger-basierte Systeme [Che+00].

Beide Lösungen sind konzeptuell an Datenbanken angelehnt. Demgegenüber stehen *Publish-Subscribe*-Systeme, die auf die Verteilung von Nachrichten ausgelegt sind [AGT14, S. 42]. Erzeuger stellen Nachrichten in das System ein (Publish) und Verbraucher abonnieren (Subscribe) sie anhand von bestimmten Filtern [RW97]. Manche Publish-Subscribe-Systeme nutzen leistungsfähige Filter, sodass sie den Verbraucher erst benachrichtigen, wenn eine Kombination von mehreren Nachrichten aufgetreten ist [Eug+03, S. 7].

Aus den bisher vorgestellten Ansätzen entwickelten sich *Complex Event Processing (CEP)* und *Data Stream Processing* Systeme. Ihre Grundlagen werden im Folgenden vorgestellt.

3.3.1 Complex Event Processing

CEP-Systeme sind stark auf die Erkennung von Mustern ausgelegt. Semantisch arbeiten sie nicht mit einem generischen Datenstrom, sondern mit einem Strom von Ereignissen. Sie knüpfen historisch gesehen an Publish-Subscribe-Systeme an, sind aber bezüglich der Operationen mächtiger als diese [CM12, S. 7]. Sie können Ereignisse sammeln, filtern, aggregieren, kombinieren, korrelieren und an andere Systeme weiterleiten. Typischerweise nutzen CEP-Systeme einfache Ereignisse als Eingabedaten und erzeugen daraus höherrangige Ereignisse. Dafür werden Regeln angewendet, die höherrangige Ereignisse unter anderem mithilfe folgender Verfahren beschreiben: *Window*-Operationen (Betrachtung eines bestimmten Zeitraums), Temporallogik und boolesche Logik [AGT14, S. 43-44].

Der Quelltext 3.1 illustriert, wie Muster anhand von mehreren einfachen Ereignissen erkannt werden. Die Abfrage nutzt vier aufeinanderfolgende Temperaturereignisse. Die erste Temperatur muss über 100 liegen, die drei darauffolgenden Temperaturen müssen jeweils höher sein als ihr Vorgänger und die vierte Temperatur eineinhalb mal höher sein als die Erste. Das Beispiel nutzt die *Esper Processing Language (EPL)* von Esper [Esp15]. Die deklarative Abfragesprache und die ausdrucksstarken Sprachmittel erleichtern die Aufgabenstellung stark [CA08, S. 8]. CEP-Systeme können beispielsweise zur Überwachung von Systemen, für Echtzeit-Geschäftsprozesse oder Prognose von Ereignissen auf Basis

```
select * from TemperatureEvent
match_recognize (
  measures A as temp1, B as temp2, C as temp3, D as temp4
  pattern (A B C D)
  define
    A as A.temperature > 100,
    B as (A.temperature < B.temperature),
    C as (B.temperature < C.temperature),
    D as (C.temperature < D.temperature)
    and D.temperature > (A.temperature * 1.5)
)
```

Quelltext 3.1: Mustererkennung über mehrere Ereignisse in Esper Processing Language [Mil13]

vorhergehender Ereignisse verwendet werden [VRR10, S. 13; EN11, S. 16-21]. Am Markt existieren zahlreiche proprietäre und freie CEP-Systeme [VRR10].

3.3.2 Data Stream Processing

Data Stream Processing Systeme (in Anlehnung an *Database Management Systeme (DBMS)* auch *Data Stream Management Systeme (DSMS)* [CM12, S. 3] oder *Stream Processing Engines* [SÇZ05, S. 5] genannt) benutzen im Gegensatz zu CEP-Systemen nicht das semantische Konzept der Ereignisse, sondern Arbeiten auf generischen Datenströmen. Das können beliebige, vom Anwender definierte Datenstrukturen sein. Dies umfasst unstrukturierte, semi-strukturierte oder strukturierte Daten [AGT14, S. 45-45]. Mögliche Anwendungen umfassen die Extraktion von Informationen aus Webseiten oder ein klassischer *Extract, Transform, Load (ETL)* Prozess für ein *Data-Warehouse* [Apa14c]. Bei Data Stream Processing Systemen spielen Operationen wie Verknüpfungen von verschiedenen Datenströmen, mathematische, analytische, anwendungsspezifische Operationen und Transformationen der Daten eine große Rolle. Die Abarbeitung von Regeln steht anders als bei CEP-Systemen weniger im Vordergrund [Cha09, S. 195-200]. Das Beispiel aus Quelltext 3.1 ist bei Data Stream Processing Systemen durch die eingeschränkten Operatoren schwer zu implementieren. Ein Mechanismus, der Mustererkennung unterstützt und automatisch Window-Operationen anwendet, ist bei Data Stream Processing Systemen normalerweise nicht vorhanden. Es gibt allerdings Bestrebungen beide Ansätze zu vereinen [CA08], etwa durch IBM InfoSphere Streams [IBM15] oder eine Kombination aus den freien Systemen Apache Storm und Esper [Fue15].

Innerhalb der Data Stream Processing Systeme gibt es zwei Ansätze: Abfrage-basierte Systeme wie STREAM [Ara+04] und universelle Stream Processing Frameworks wie Storm [Tos+14]. Bei Ersteren liegt der Fokus auf der Benutzung einer deklarativen Abfragesprache. Solche Sprachen sind oft ähnlich wie SQL aufgebaut. STREAM verwendet eine Abfragesprache namens *Continous Query Language (CQL)* [Ara+04]. Es gibt Anstrengungen eine einheitliche Sprache zu entwickeln, ein übergreifender Standard hat sich aber bisher nicht herausgebildet [Jai+08]. Universelle Stream Processing Frameworks bieten keine Abfragesprache an, sondern stellen eine Plattform für beliebige Berechnungen bereit, die selbst implementiert werden müssen. Beide Ansätze schließen sich nicht aus, sondern können auch in einem Framework kombiniert werden.

3.4 Framework

Ein *Framework* ist im Bereich der Informationstechnologie ein Gerüst, mit dessen Hilfe andere Anwendungen gegliedert und entwickelt werden. Es unterstützt beispielsweise durch Hilfsprogamme, Bibliotheken, Skriptsprachen, allgemeine Dienste, Schnittstellen oder andere Programme die Entwicklung von Anwendungen [SH06, S. 1].

Es gibt mehrere Typen von Frameworks. Der im Kontext von Stream Processing Frameworks verwendete Typ ist das *Platform Framework*. Dieser Typ stellt ein Programmiermodell und eine Laufzeitumgebung bereit [SH06, S. 1]. Ein Platform Framework beeinflusst daher maßgeblich die Entwicklung einer Applikation.

4 Stream Processing Frameworks

In Abschnitt 3.3.2 wurden schon die Grundlagen von Stream Processing Frameworks dargelegt. Dieses Kapitel erläutert eingehend das zugrunde liegende Modell, die Kernaspekte und stellt eine Referenzarchitektur für selbige vor.

Eine einheitliche Terminologie hat sich noch nicht durchgesetzt, daher werden bei der Einführung eines neuen Begriffs meist mehrere Alternativen genannt.

4.1 Stream Processing Modell

Stream Processing lässt sich mit drei wesentlichen Komponenten konzeptionell modellieren: Datenquellen, Daten und Verarbeitungsknoten [AGT14, S. 45-52; Boc14, S. 8]. Dieses abstrakte Modell ist in Abbildung 4.1 dargestellt.

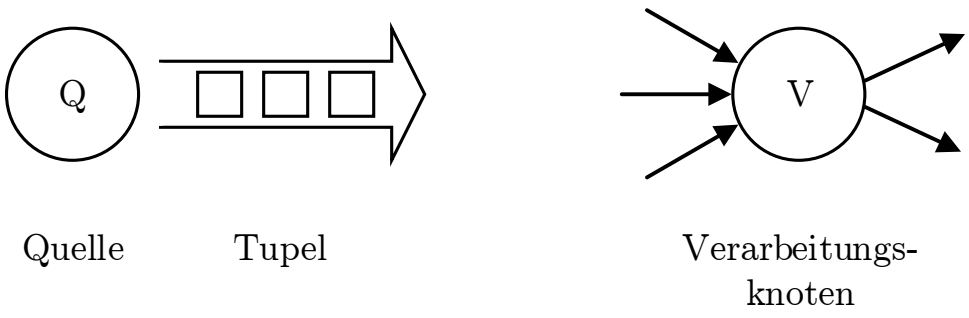


Abbildung 4.1: Grundlegendes Modell von Stream Processing nach [Boc14, S. 8]

Datenquellen emittieren einen kontinuierlichen Datenstrom, der aus elementaren Datenelementen besteht. Elementare Datenelemente (Tupel) können beliebige Daten enthalten.

Mögliche Datenquellen sind: Sensordaten von *Internet of Things* Geräten [Gui+11, S. 120], Traffic-Daten von Netzwerkknoten [Lal+06], Finanznachrichten [FYL02] oder Aktienkurse [Zha+09]. Je nach Framework sind die Quellen per *Push*- oder *Pull*-Prinzip angebunden. Beim Push-Prinzip werden neue Tupel sofort an das Framework zur Weiterverarbeitung übergeben. Beim Pull-Prinzip fragt das Framework zyklisch bei der Datenquelle nach, ob neue Tupel vorhanden sind.

Verarbeitungsknoten konsumieren Tupel, führen eine beliebige Anwendungslogik aus und können neue Tupel emittieren. Sie können zustandslos oder zustandsbehaftet sein. Zustandsbehaftete Verarbeitungsknoten steigern die Komplexität der Anwendung, da der Zustand gehalten und für einen Fehlerfall gesichert werden muss.

Die verknüpften Datenquellen und Verarbeitungsknoten bilden einen Graph, der den Datenfluss der Anwendung abbildet. Der Graph stellt die logische Struktur der Anwendung dar. Abbildung 4.2 zeigt einen beispielhaften Graphen. Bei typischen Stream Processing Frameworks werden spezielle Graphen namens *Directed Acyclic Graph (DAG)* genutzt, deren Kanten entlang des Datenflusses gerichtet und zyklenfrei sind. Die Aufgabe des Stream Processing Frameworks ist es, ein Programmiermodell und eine Laufzeitumgebung anzubieten, mit der man Anwendungen aus Datenquellen und Verarbeitungsknoten aufbauen und implementieren kann. Anschließend sorgt das Framework für die Bereitstellung und die Ausführung der Anwendung auf *Hosts* (physische Knoten) und die Weiterleitung der Tupel entlang dem Graphen.

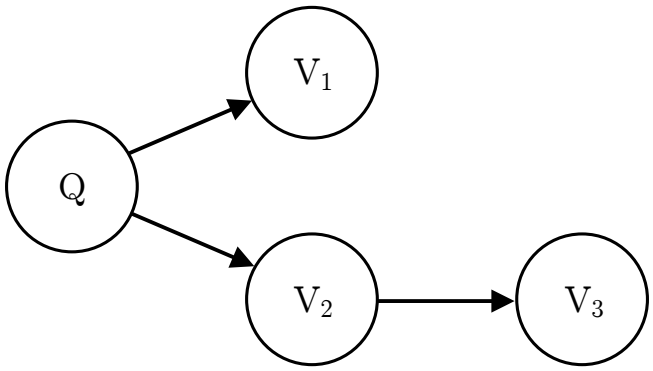


Abbildung 4.2: Logisches Modell einer Stream Processing Anwendung mit einer Quelle und drei Verarbeitungsknoten

4.2 Anforderungen an Stream Processing Frameworks

Die Anforderungen an Stream Processing Frameworks unterscheiden sich in vielerlei Hinsicht von klassischen Batch-basierten Systemen. Um einen Leitfaden für Anwender zu bieten, aber auch, um die Entwicklung von entsprechenden Frameworks zu forcieren, stellen Stonebraker, Çetintemel und Zdonik acht Anforderungen auf [SÇZ05]:

1. **Keep the Data Moving**
   Datenverarbeitung als Datenstrom ohne Zwischenspeicherung durch Verwendung des Push-Prinzips.
2. **Query using SQL on Streams**
   Nutzung einer Abfragesprache mit vordefinierten Operationen.
3. **Handle Stream Imperfections**
   Eingebaute Behandlung von fehlenden oder verspäteten Daten.
4. **Generate Predictable Outcomes**
   Deterministische Ausführung unabhängig von Fehlersituationen.
5. **Integrate Stored and Streaming Data**
   Kombination von aktuellen und historischen/zustandsbehafteten Daten.
6. **Guarantee Data Safety and Availability**
   Verfügbarkeit und Integrität der Daten unabhängig von Fehlersituationen.
7. **Partition and Scale Applications Automatically**
   Verteilte Architektur möglichst mit automatischer Skalierung.
8. **Process and Respond Instantaneously**
   Geringe Latenz auch bei großem Datenvolumen.

Es ergibt sich ein Zielkonflikt zwischen den Anforderungen: Verfügbarkeit und Integrität (6.) sicherzustellen erzeugt zusätzlichen Aufwand, der sich auf die Latenz (8.) auswirkt. Um diesen Konflikt aufzulösen, gibt es verschiedene Optionen, je nach dem, welchem Ziel mehr Gewicht beigemessen wird. Die Forderung einer Abfragesprache (2.) ist abhängig vom Anwendungsfall. Wie in Abschnitt 3.3.2 dargelegt, gibt es sowohl abfrageorientierte als auch universelle Stream Processing Frameworks. Letztere sind besonders für anwendungsspezifische Operationen geeignet. Diese Forderung lässt sich als Wunsch nach einer

unterstützenden, anwenderfreundlichen Schnittstelle umdeuten. Die Forderung nach einer Behandlung von fehlenden/verspäteten Daten (3.) ist bei universellen Frameworks ebenfalls als anwendungsspezifisch zu sehen. Die anderen Anforderungen werden durch aktuelle Systeme adressiert, wie im Verlauf der Arbeit dargestellt wird.

Die vorgestellten Anforderungen führen zu den zwei wesentlichen Merkmalen von Stream Processing Frameworks: ihre verteilte Architektur und ihre Fehlertoleranzmechanismen. Beide Bereiche werden in den folgenden Unterkapiteln erläutert.

4.2.1 Verteilte Architektur

Grundlage für ein leistungsfähiges Stream Processing System ist eine verteilte Architektur (Anforderung 7). Eine Hauptaufgabe des Frameworks ist es, die logische Struktur der Anwendung auf Hosts abzubilden. Zwei Aspekte spielen dabei eine große Rolle: die Verteilung der Verarbeitungsknoten auf die Hosts und der Transport der Daten zwischen den Knoten [AGT14, S. 45-50].

Bei der Abbildung wechselwirken beide Aspekte miteinander, denn der Transport zwischen den Hosts ist im Vergleich zur Kommunikation innerhalb des Hosts sehr langsam. Eine Konzentration von Verarbeitungsknoten auf wenige Hosts senkt den Kommunikationsaufwand. Andererseits erhöhen möglichst viele Hosts die Leistungsfähigkeit des Gesamtsystems im Hinblick auf die Rechenleistung. Das Framework hat die Aufgabe diesen Zielkonflikt zu lösen. Je nach Framework geschieht die Verteilung automatisch oder unterstützt durch den Entwickler.

Verteilung der Verarbeitungsknoten

Bei der Verteilung der Verarbeitungsknoten müssen unterschiedliche Aspekte berücksichtigt werden: Kommunikationskosten, Host-Ressourcen, Latenz- oder Durchsatzorientierung und anwendungsspezifische Restriktionen. Hirzel u. a. beschreiben in [Hir+14] einige Strategien, um die Verteilung auf Hosts zu optimieren. Eine sehr wichtige Optimierung ist die *Partitionierung* (*Data Parallelism*) [Hir+14, S. 13]. Sie ermöglicht Verarbeitungsknoten auf viele Hosts zu skalieren, indem der Datenstrom in mehrere Datenströme aufgeteilt wird. Technisch wird die Verteilung durch einen *Master* koordiniert. Auf den Hosts sind *Worker*-Prozesse installiert, die vom Master gesteuert werden. Vom Master zugeteilte Verarbeitungsknoten werden durch den Worker gestartet und überwacht. Abbildung 4.3

zeigt das logische Modell aus Abbildung 4.2 abgebildet auf physische Knoten. Je nach Framework werden die Verarbeitungsknoten in eigenen Prozessen oder nur in *Threads* des Worker Prozesses ausgeführt.

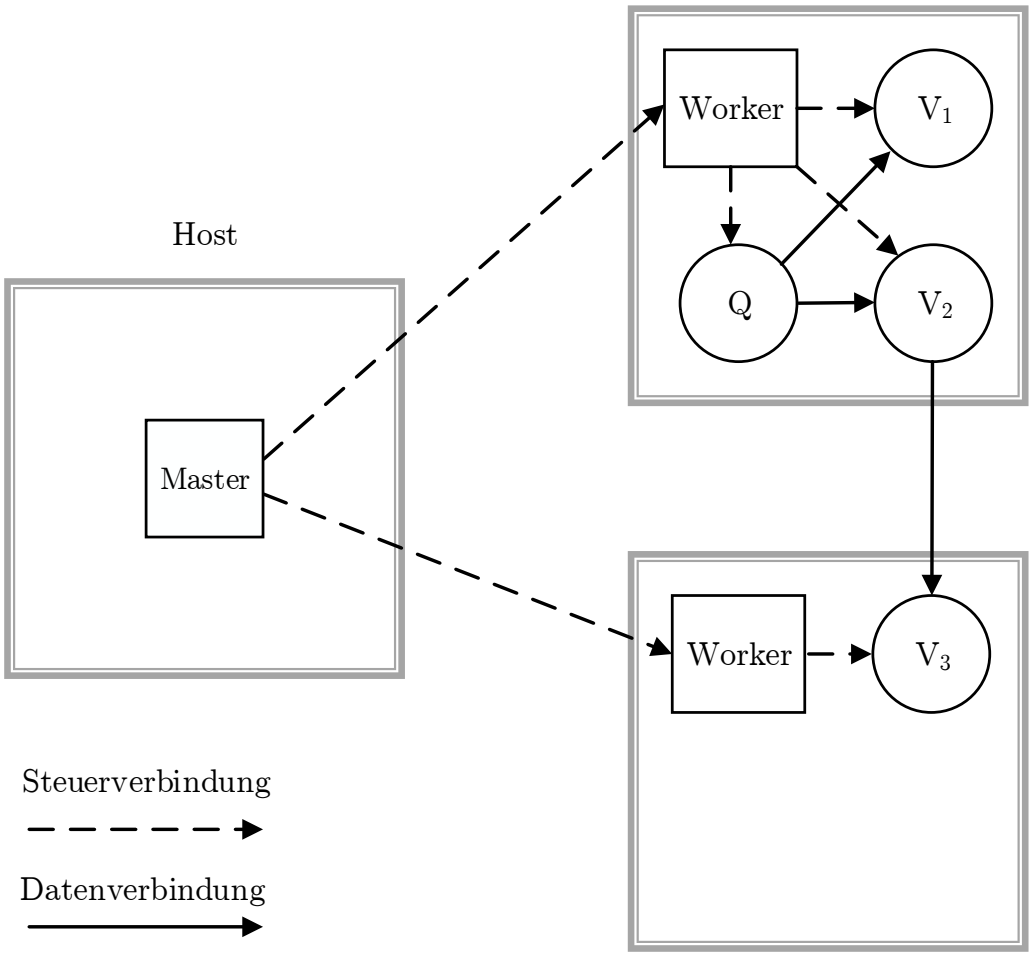


Abbildung 4.3: Physisches Modell des logischen Modells aus Abbildung 4.2

Transport der Daten

Die Verteilung der Anwendung auf mehrere Hosts bedingt, dass das Framework die Daten zwischen den Knoten transportieren muss. Das Transportsystem muss die Daten verlässlich, und, je nach Anwendungsfall, in der richtigen Reihenfolge weiterleiten. Zusätzlich sind die zwei Varianten Latenz- oder Durchsatzorientierung zu berücksichtigen. Latenzorientierte Anwendungen benötigen idealerweise einen sofortigen Transport von einzelnen

Datensätzen, während bei durchsatzorientierten Anwendungen aus Effizienzgründen eher Pufferung und die Übertragung in größeren Einheiten vorzuziehen ist. Technisch gibt es mehrere Möglichkeiten zur Implementierung. Wenn Verarbeitungsknoten auf dem gleichen Host in einem Thread ausgeführt werden, müssen die Daten gar nicht transportiert werden, da sie im gleichen Adressraum (Prozess) vorliegen. Falls die Verarbeitungsknoten in verschiedenen Prozessen ausgeführt werden, können Techniken der *Interprozesskommunikation* wie *Shared Memory* zum Einsatz kommen [AGT14, S. 171]. Der Datenaustausch zwischen Hosts wird mit *Remote Procedure Calls* oder *Message Brokern* abgewickelt [Boc14, S. 13]. Remote Procedure Calls sind über das Netzwerk ausgeführte Methodenaufrufe, die direkt zwischen den Workern ausgeführt werden, um Tupel auszutauschen. Das Framework muss den korrekten Transport selbst sicherstellen. Ein Message Broker ist eine Instanz, die zwischen den Workern steht. Sie sorgen im Zusammenspiel mit dem Framework für einen korrekten Transport der Tupel. Dort können die Tupel auch zwischengespeichert oder Datenströme neu zusammengestellt werden.

4.2.2 Fehlertoleranz

Stream Processing Frameworks skalieren auf mehrere Hundert Hosts [Tos+14, S. 5]. Dabei wird aus Kostengründen auf handelsübliche Standardhardware und nicht etwa auf hochverfügbare Spezialhardware zurückgegriffen. Durch die hohe Anzahl an Hosts werden Ausfälle von der Ausnahme zur Regel und müssen daher berücksichtigt werden [VN10]. Nicht nur Hardwareausfälle können Hosts beeinträchtigen, auch normale Softwareupdates, bei denen etwa ein Neustart erforderlich ist, sind einfacher zu handhaben, wenn das Framework fehlertolerant arbeitet. Außerdem können Fehler bei der Ressourcenverteilung zu Problemen führen, woraufhin ein Verarbeitungsknoten beendet werden muss. Das Framework muss sowohl mit Fehlern des Masters, als auch der Worker, umgehen können. Das Sicherstellen der Fehlertoleranz ist eine maßgebliche Aufgabe eines Stream Processing Frameworks. Sie erhöht die Komplexität des Frameworks und beeinträchtigt die Leistungsfähigkeit durch den zusätzlichen Aufwand. Daher gibt es unterschiedliche Ansätze, um je nach Anwendungsfall die geeigneten Maßnahmen treffen zu können.

Der Master übernimmt nur organisatorische Aufgaben, deshalb werden laufende Anwendungen idealerweise nicht durch einen fehlerhaften Master beeinträchtigt. Der Zustand des Masters wird in einem verteilten Konfigurationsdienst [Hun+10] gespeichert. Ent-

weder wird der Master durch ein Überwachungsprogramm sofort neu gestartet oder ein redundanter Master übernimmt seine Aufgabe.

Für die Absicherung von Workern müssen besondere Maßnahmen getroffen werden. Worker melden sich periodisch beim Master. Wenn diese Meldung ausbleibt, wird ein fehlerhafter Worker angenommen. Neue Tupel werden auf einem anderen Worker bearbeitet, die Tupel die zwischen dem Eintritt des Fehlers und der Zeitüberschreitung an den Worker geschickt wurden, müssen gesondert behandelt werden. Hwang u. a. zeigen in [Hwa+05] auf, welche Möglichkeiten existieren diese Tupel zu bearbeiten.

Die *Precise Recovery*, auch als *Exactly-Once*-Semantik [MBW10, S. 323] bezeichnet, ist die stärkste Wiederherstellungsgarantie. Ihr Ausgabestrom im Fehlerfall ist identisch zum Ausgabestrom im Normalfall, eventuell mit einer höhren Latenz. Die *Rollback Recovery* bietet eine schwächere Wiederherstellungsgarantie. Ihr Ausgabestrom im Fehlerfall ist nicht identisch zum Ausgabestrom im Normalfall. Sie stellt lediglich sicher, dass jedes Tupel mindestens einmal vollständig verarbeitet wird. Zugunsten der Leistungsfähigkeit werden bei der Rollback Recovery duplizierte Tupel zugelassen. Dieses Verfahren wird auch als *At-Least-Once*-Semantik [MBW10, S. 323] bezeichnet. Beide Garantien können durch verschiedene Techniken realisiert werden. Etwa *Active/Passive Standby*, bei dem ein anderer Worker die Aufgaben des ausgefallenen Workers übernimmt oder *Upstream Backup*, bei dem die Tupel von der Quelle neu abgerufen werden. Diese Techniken unterscheiden sich hinsichtlich ihrer benötigten Rechenleistung, ihrer beanspruchten Netzwerkbandbreite und der Latenz bei der Wiederherstellung. [Hwa+05, S. 5-8]

Das dritte Verfahren ist die *Gap Recovery*. Sie bietet keine Wiederherstellungsgarantie. Tupel, die noch nicht vollständig verarbeitet wurden, sind verloren. Der Ausgabestrom wird fortgesetzt sobald ein neuer Worker zugewiesen wurde. Dadurch entsteht kein zusätzlicher Aufwand für die Wiederherstellung, allerdings ist dieses Verfahren mit Datenverlust verbunden [Hwa+05, S. 4]. Das Verfahren hat *At-Most-Once*-Semantik [MBW10, S. 323], das heißt ein Tupel wird maximal einmal, im Fehlerfall möglicherweise keinmal, verarbeitet.

Die Frameworks unterscheiden sich neben der Implementierung des Wiederherstellungsverfahrens auch in dessen Transparenz für Entwickler. Bei zustandsbehafteten Operationen müssen nicht nur die Tupel gesichert werden, sondern auch der interne Zustand der Operationen. Je nach Framework wird dem Entwickler eine Schnittstelle zur Verfügung gestellt, mit der diese Zustände in das Wiederherstellungsverfahren des Frameworks aufgenommen werden können. Das Framework kann im Wiederherstellungsfall auch Informationen an

den Entwickler geben, damit er geeignete Maßnahmen selbst implementieren kann, um fehlerhafte Werte durch duplizierte Tupel zu vermeiden.

4.3 Referenzarchitektur

Die bereits ausgeführten Anforderungen und deren Auswirkungen spiegeln sich in der Architektur von Stream Processing Frameworks wider. Im Folgenden wird eine Referenzarchitektur vorgestellt, welche die wesentlichen logischen Komponenten und deren Beziehungen innerhalb eines Stream Processing Frameworks beschreibt. Im späteren Verlauf der Arbeit werden die Komponenten der konkreten Frameworks mithilfe des Referenzframeworks zugeordnet. Trotz unterschiedlicher Namenskonventionen ist so eine einfache Zuordnung der Komponenten möglich. Die Beschreibung mancher Komponenten basiert auf der Arbeit von Andrade, Gedik und Turaga [AGT14]. Die Architektur ist in Abbildung 4.4 dargestellt.

Eine Anwendung, die auf das Framework aufsetzt, nutzt primär dessen **Schnittstelle** zur Anwendungsprogrammierung, auch *Application Programming Interface (API)* genannt. Sie stellt den Zugang zu den Transport- und Verarbeitungssystemen bereit. Aktuell nutzt jedes Framework eine andere Schnittstelle, sodass Anwendungen im Regelfall nicht portabel programmiert werden können. Es gibt zwei verschiedene Ansätze beim Aufbau der Schnittstelle: Im einfachen Fall wird nur eine Abstraktion des Graphs aus Datenfluss und Verarbeitungsknoten bereitgestellt. Das heißt, man definiert und implementiert die Verarbeitungsknoten und verschiedene Verbindungstypen zwischen ihnen. Dieser Ansatz verfolgt einen niedrigen Abstraktionsgrad. Der zweite Ansatz nutzt einen höheren Abstraktionsgrad. Es stehen Operatoren ähnlich wie bei einer Abfragesprache bereit, etwa Bedingungen oder Gruppierungen. Als Operator werden die grundlegenden Transformationen bezeichnet. Eine dem Operator übergebene benutzerdefinierte Funktion entscheidet, wie die Transformation konkret durchgeführt wird.

Das **Transportsystem** wird je nach Framework implizit oder explizit bereitgestellt. Bei Ersterem werden die Ein- und Ausgabewerte von Verarbeitungsknoten automatisch transportiert. Fehler bei der Verknüpfung der Verarbeitungsknoten werden bereits beim Kompilieren erkannt. Bei Letzterem werden Ein- und Ausgabewerte manuell definiert und für jeden Verarbeitungsknoten aufgeführt. Die Typsicherheit ist bei dieser Variante nicht gewährleistet.

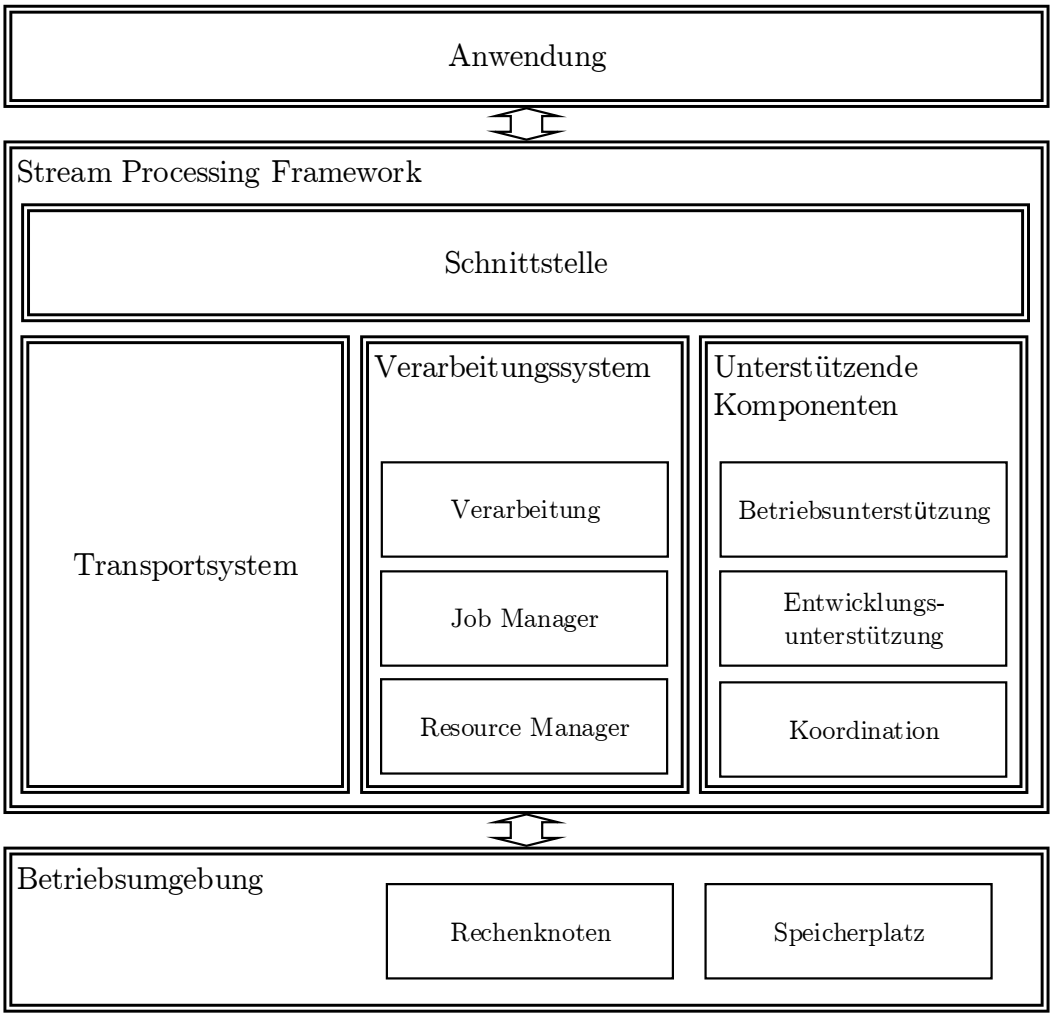


Abbildung 4.4: Referenzarchitektur von Stream Processing Frameworks

Das **Verarbeitungssystem** stellt die Laufzeitumgebung für die Verarbeitungsknoten bereit. Diese Umgebung umfasst Rechenzeit, Arbeitsspeicher, Netzwerkzugriff und gegebenenfalls Zugriff auf weitere Ressourcen wie verteilte Datenspeicher. Die Laufzeitumgebung wird vom Worker gesteuert. Der ***Job Manager*** verwaltet die Verteilung von Applikationen auf die Betriebsumgebung. Der Begriff Job kann je nach Framework für eine gesamte Applikation oder als Synonym für einen Verarbeitungsknoten stehen. Das *Lifecycle-Management* der Applikation wird vom Job Manager gesteuert [AGT14, S. 207]. Durch ihn werden Anwendungen bereitgestellt, beendet und deren Zustand überwacht. Anwendungen können isoliert voneinander auf dem gleichen Cluster ausgeführt werden. Eine Anwendung, die bereitgestellt werden soll, wird an den Job Manager übergeben und dieser

entscheidet in Verbindung mit dem ***Resource Manager*** in einem *Scheduling*-Prozess wie die Anwendung verteilt wird [AGT14, S. 208 f.]. Der Resource Manager überwacht den Zustand und die Leistungsfähigkeit der Worker. Beim Scheduling nutzt er diese Daten, um die Verteilung der Anwendung zu optimieren. [AGT14, S. 207 ff.]

Neben diesen hauptsächlichen Komponenten werden verschiedene Querschnittsfunktionen von unterstützenden Komponenten übernommen. Die ***Koordination*** und Konfiguration von verschiedenen Komponenten einer Verteilten Anwendung ist nicht trivial. Verteilte Algorithmen für gegenseitigen Ausschluss oder die Auswahl eines Anführers werden vom Framework benötigt. Sie werden meist durch eine externe Software wie Zookeeper bereitgestellt. [Hun+10]

Die **Entwicklungsunterstützung** beinhaltet unter anderem Komponenten für *Debugging* [Zel06, S. 145 ff.] und *Logging* [Zel06, S. 200 ff.] [AGT14, S. 213 f.]. Sie sind für die Entwicklung und den Betrieb einer Anwendung erforderliche Aspekte. Die Fehlersuche und -beseitigung wird durch die Verteilung auf verschiedene Hosts erschwert. Daher stellt das Framework Werkzeuge zur Vereinfachung bereit, etwa die lokale Ausführung aller Komponenten innerhalb eines Prozesses [AGT14, S. 214]. Logging an einen zentralen Ort erleichtert ebenfalls das Debuggen von Framework- oder Anwendungskomponenten und ermöglicht im Betrieb eine einfache Überwachung der Anwendung.

Komponenten zur **Betriebsunterstützung** ermöglichen den produktiven Betrieb einer Anwendung. Dazu zählen etwa Sicherheitsaspekte, Statistiken, Verwaltungs- und Überwachungswerkzeuge. Vor allem Sicherheitsaspekte spielen in jedem Bereich der Architektur eine Rolle, insbesondere wenn sensible Daten verarbeitet werden sollen. Einerseits ist damit eine sichere Datenübertragung gemeint, andererseits auch die Authentifizierung und Autorisierung von Nutzern. Statistiken und Metriken erleichtern es, Engpässe zu erkennen und darauf zu reagieren [AGT14, S. 214]. Verwaltungs- und Überwachungswerkzeuge erleichtern Wartungsaufgaben, das Bereitstellen von Anwendungen und die Reaktion auf Fehlersituationen. Zu den Werkzeugen zählen sowohl *Command Line Interfaces (CLIs)*, als auch webbasierte *Graphical User Interfaces (GUIs)*.

Die **Betriebsumgebung** stellt im Kontext von verteilten Stream Processing Frameworks einen Rechnerverbund (*Cluster*) aus mehreren Hosts dar. Alle Mitglieder eines Clusters befinden sich in einem Rechnernetzwerk und werden über ein Verwaltungswerkzeug verknüpft. Die Ressourcen des Clusters können über dieses Werkzeug abgerufen werden. Es kann ein universeller Dienst wie Apache Hadoop YARN [Vav+13] oder ein Framework-spezifischer Dienst sein. Zusätzlich zur Ressourcenverwaltung benötigen einige

Frameworks Speicherplatz, um *Checkpoints* zur Fehlerwiederherstellung zu speichern. Um den Speicherplatz ausfallsicher bereitzustellen, wird auf verteilte Dateisysteme wie das *Hadoop Distributed File System (HDFS)* [Apa15s] zurückgegriffen.

5 Konzept

Das Konzept der Evaluierung ist auf den Grundlagen aus Unterkapitel 3.1 und Unterkapitel 3.2 aufgebaut. Es wird eine analytische, vergleichende Evaluierung durchgeführt. Analytisch bedeutet, dass die Evaluierung von einem Evaluator anhand eines Leitfadens durchgeführt wird und keine Tests durch echte Nutzer durchgeführt werden. Die Frameworks werden in ihrer vorliegenden Version als abgeschlossen betrachtet. Die Evaluierung ist summativ, wird also nicht im Rahmen der Entwicklung eines Frameworks durchgeführt und nimmt auch keinen Einfluss auf deren Weiterentwicklung [Heg03, S. 8-15]. In den folgenden Unterkapiteln werden die Inhalte der Evaluierung konkretisiert.

5.1 Evaluator

Der gesamte Evaluierungsprozess wird vom Autor dieser Arbeit durchgeführt. Die Verteilung der verschiedenen Aspekte einer Evaluierung auf verschiedene Instanzen ist grundsätzlich wünschenswert, um die Qualität der Evaluierung zu steigern. Im Rahmen dieser Arbeit wird auf eine Verteilung aufgrund der beschränkten Ressourcen verzichtet. Um dennoch eine hohe Qualität und vor allem die Nachvollziehbarkeit der Ergebnisse zu gewährleisten, wird der Evaluierungsprozess ausführlich dokumentiert.

5.2 Gegenstand

Der Gegenstand der Evaluierung sind ausgewählte, freie, lokal installierbare Stream Processing Frameworks. Proprietäre und Cloud-basierte Frameworks wie IBM InfoSphere Streams [IBM15] oder Amazon Kinesis [Ama15c] sind von der Evaluierung ausgeschlossen. Der Aufwand für die Beschaffung, die eingeschränkte Verfügbarkeit von Evaluierungslizenzen und die Kosten für die Nutzung von Cloud-basierten Lösungen würde den Rahmen dieser

Arbeit übersteigen. Das vorgestellte Evaluierungskonzept ist jedoch insoweit generisch gefasst, dass es sich auch auf weitere Frameworks übertragen lässt.

Anhand einer vorgelagerten Literatur- und Internetrecherche [Apa15ad; Boc14; CM12; Ell14; LIX14; Sic15] wurden zum Stichtag 1. Juni 2015 folgende Frameworks identifiziert: Apache Flink [Apa15q], Apache S4 [Apa13a], Apache Samza [Apa15ae], Apache Spark [Zah+10], Apache Storm [Tos+14], MUPD8 [Alm12], STREAM [Ara+04] und Tigon [Cas15]. Für die Aufnahme in die Evaluierung muss das Framework aktiv gepflegt und weiterentwickelt werden. Dies wird anhand der freigegebenen Versionen und der Entwicklungsaktivität in der Versionsverwaltung festgestellt. Die Systeme Apache S4, MUPD8, STREAM und Tigon verzeichnen keine freigegebene Version in den letzten zehn Monaten. Abgesehen davon ist fast keine Aktivität in deren Versionsverwaltungen erkennbar.

Die folgenden Abschnitte beschreiben den Aufbau und die Funktionen der ausgewählten Frameworks. Sie orientieren sich an den Anforderungen und der Referenzarchitektur aus Kapitel 4. Die Frameworks sind in alphabetischer Reihenfolge gelistet.

5.2.1 Apache Flink

Apache Flink [Apa15q] ging aus dem Stratosphere Projekt [Dat15b] hervor, das an der Technischen Universität Berlin, an der Humboldt-Universität zu Berlin und am Hasso-Plattner-Institut in Potsdam angesiedelt ist [Ale+14]. Es ist eine Plattform zur verteilten Datenverarbeitung, die sowohl im Batch- als auch im Streaming-Modus arbeitet. Durch eine umfangreiche API können Anwendungen mit einem hohen Abstraktionsgrad geschrieben werden. Flink bietet neben der grundlegenden API weitere Bibliotheken im Bereich Graphverarbeitung, Maschinenlernen und Datenverarbeitung mit SQL-ähnlicher Syntax an. Die Ausführung ist als *Pipeline* mit dauerhaft aktiven Operatoren ausgelegt. Sie werden während der Ausführungszeit immer mit neuen Tupeln versorgt [Apa15g]. Flink ist in Java und Scala [Éco15] implementiert.

Die Architektur von Flink ist in Abbildung 5.1 dargestellt. Zur Erstellung von Anwendungen wird die ***DataStream API*** verwendet. Sie stellt Funktionen zur Streamverarbeitung bereit. Die ***DataSet API*** bezieht sich auf Batch-Verarbeitung. Neben diesen beiden Schnittstellen gibt es noch weiterführende Schnittstellen, etwa für maschinelles Lernen. Diese Schnittstellen sind nicht auf die Streamverarbeitung angepasst und sind für diese Arbeit daher nicht direkt relevant. Die Aufgaben des Workers übernimmt der ***TaskManager***. Er verwaltet die Tasks, die für die Datenverarbeitung zuständig sind. Zusätzlich

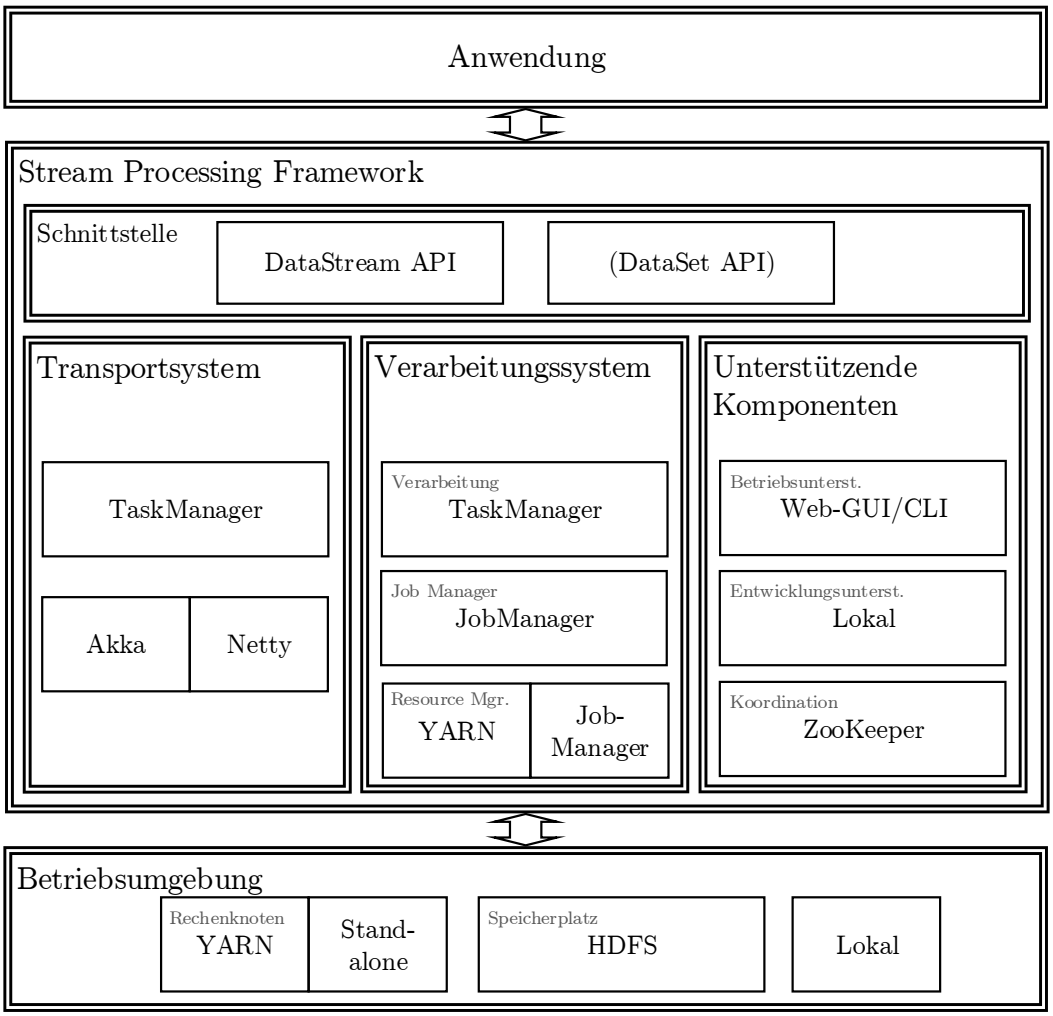


Abbildung 5.1: Architektur von Apache Flink

führt er Fehlertoleranzmaßnahmen wie periodisches Sichern von Zwischenständen (Checkpointing) durch. Er steht mit TaskManagern auf anderen Hosts und dem ***JobManager*** über die Kommunikationsframeworks ***Akka*** [Typ15] und ***Netty*** [The15] in Kontakt. Der JobManager übernimmt die Funktionen des Masters. Er verwaltet die Anwendungen auf dem Cluster und plant deren Ressourcen. Zusammen mit **ZooKeeper** stellt er die fehlertolerante Ausführung von Anwendungen sicher. Für die Clusterverwaltung nutzt Flink entweder den JobManager im ***Standalone***-Betrieb, also ohne ein externes Werkzeug, oder beispielsweise Apache Hadoop **YARN**. Das verteilte Dateisystem **HDFS** dient der sicheren Ablage von Checkpoints. Der **lokale** Betriebsmodus erleichert das Debuggen, da

alle Komponenten lokal ausgeführt werden. Zum Starten, Beenden und Verwalten der Anwendungen kann eine **CLI** oder eine **Web-GUI** verwendet werden.

5.2.2 Apache Samza

Apache Samza [Apa15ae] wurde ursprünglich von LinkedIn [Ric13] entwickelt. Samza ist ausschließlich auf die Stream-Verarbeitung ausgelegt. Im Gegensatz zu den anderen Frameworks hat es kein eigenes Transport- und Verarbeitungssystem. Es nutzt Apache Kafka [KNR11] als Transportsystem und Apache Hadoop YARN als Verarbeitungssystem. Die API lehnt sich stark an das reine Stream Processing Model an, das keine höheren Abstraktionen als Verarbeitungsknoten und Datenströme kennt. Die Verarbeitungsknoten werden bei Samza *Tasks* genannt. Sie werden beim Starten der Anwendungen auf die Hosts verteilt und die Daten per Kafka zwischen ihnen transportiert. Samza ist in Java und Scala implementiert.

Die Architektur von Samza ist in Abbildung 5.2 dargestellt. Samza bietet als einzige Schnittstelle die **Samza API** zur Anwendungsentwicklung an. Die Verarbeitung findet durch ***TaskRunner*** statt, die in *SamzaContainern* arbeiten. Der ***ApplicationMaster*** verwaltet die Anwendung und fordert vom ***YARN ResourceManager*** die nötigen Ressourcen an. Sie werden durch die ***YARN NodeManager*** gewährt, die sich auf den einzelnen Hosts befinden. Das Transportsystem wird vollständig von Apache **Kafka** bereitgestellt. Zusammen mit **ZooKeeper** sorgt es für die nötige Fehlertoleranz, indem der Verarbeitungsstand dort gesichert wird. Für zustandsbehaftete Operationen wird ***RocksDB*** [Fac15] als *Key-Value-Store* angeboten. Auf Kafka wird ein *Changelog* gespeichert, sodass bei einem TaskRunner-Fehler die Datenbank wiederhergestellt werden kann. Mithilfe der CLI startet man Anwendungen auf dem Cluster. Die Verwaltung geschieht mit üblichen YARN-Werkzeugen. Durch die **Web-GUI** von Samza und YARN kann die Anwendung überwacht werden. Samza stellt über Kafka **Metriken** bereit, die durch anwendungsspezifische Metriken erweitert werden können. Samza stellt von diesen Komponenten nur die API, den TaskRunner, den ApplicationMaster und den Client bereit, alle anderen Komponenten werden durch andere Systeme bereitgestellt.

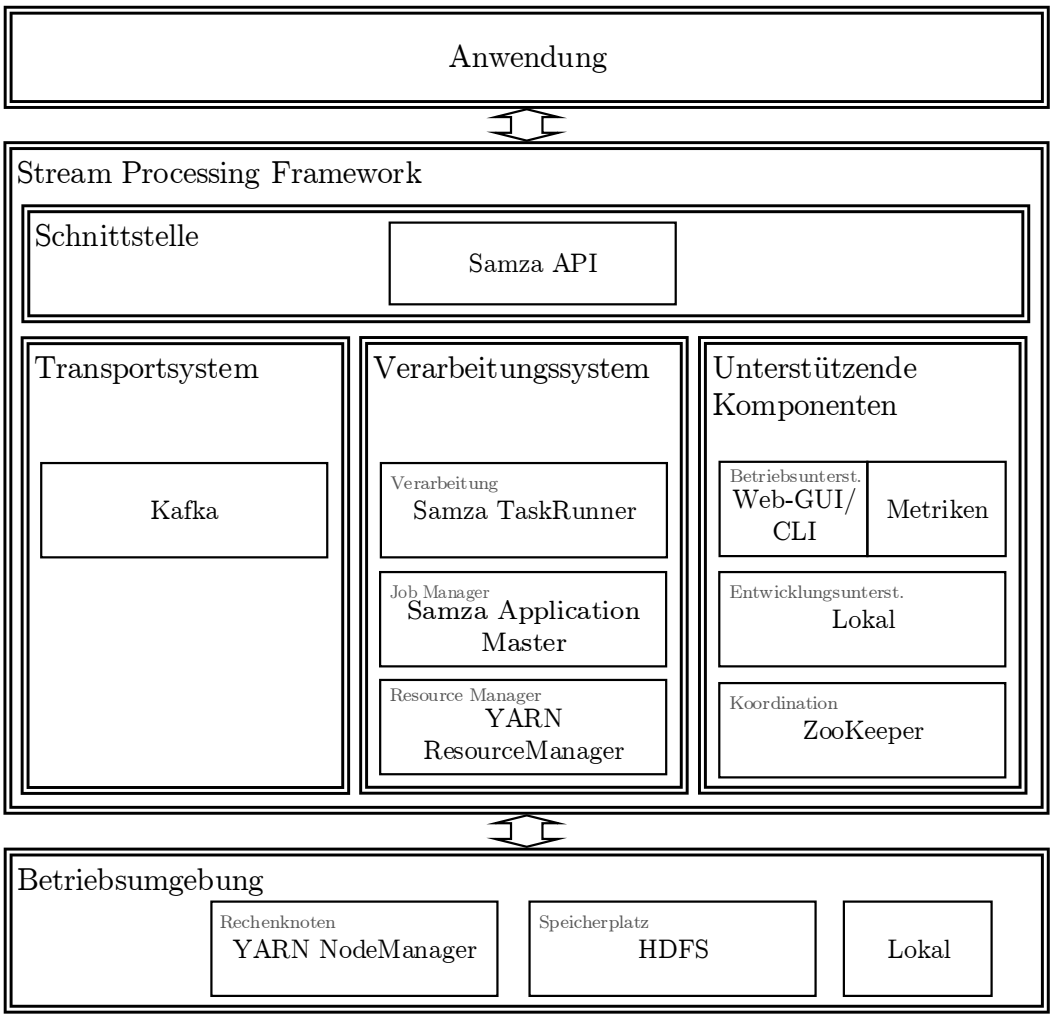


Abbildung 5.2: Architektur von Apache Samza

5.2.3 Apache Spark

Apache Spark [Zah+10] ging aus einem Forschungsprojekt der University of California, Berkeley hervor. Es war ursprünglich als Ersatz für das Apache Hadoop MapReduce Framework konzipiert. Spark nutzt eine Abstraktion namens *Resilient Distributed Dataset (RDD)*. RDDs sind schreibgeschützte Daten-Partitionen, die auf den Rechenknoten vorliegen. Sie können im Fehlerfall einfach aus den Ausgangsdaten wiederhergestellt werden. Dieses Batch-orientierte Konzept wurde später für Streaming-Daten angepasst [Zah+12]. Es nutzt kein natives Streaming, sondern Micro-Batching, das heißt, es sammelt eingehende Daten für einen gewissen Zeitraum und verarbeitet diese in bestimmten

Zeitabständen. Bei jedem Verarbeitungsschritt werden die Operatoren neu auf die Hosts verteilt. Ähnlich wie bei Flink stehen eine umfangreiche API und einige Bibliotheken bereit. Spark ist hauptsächlich in Scala implementiert.

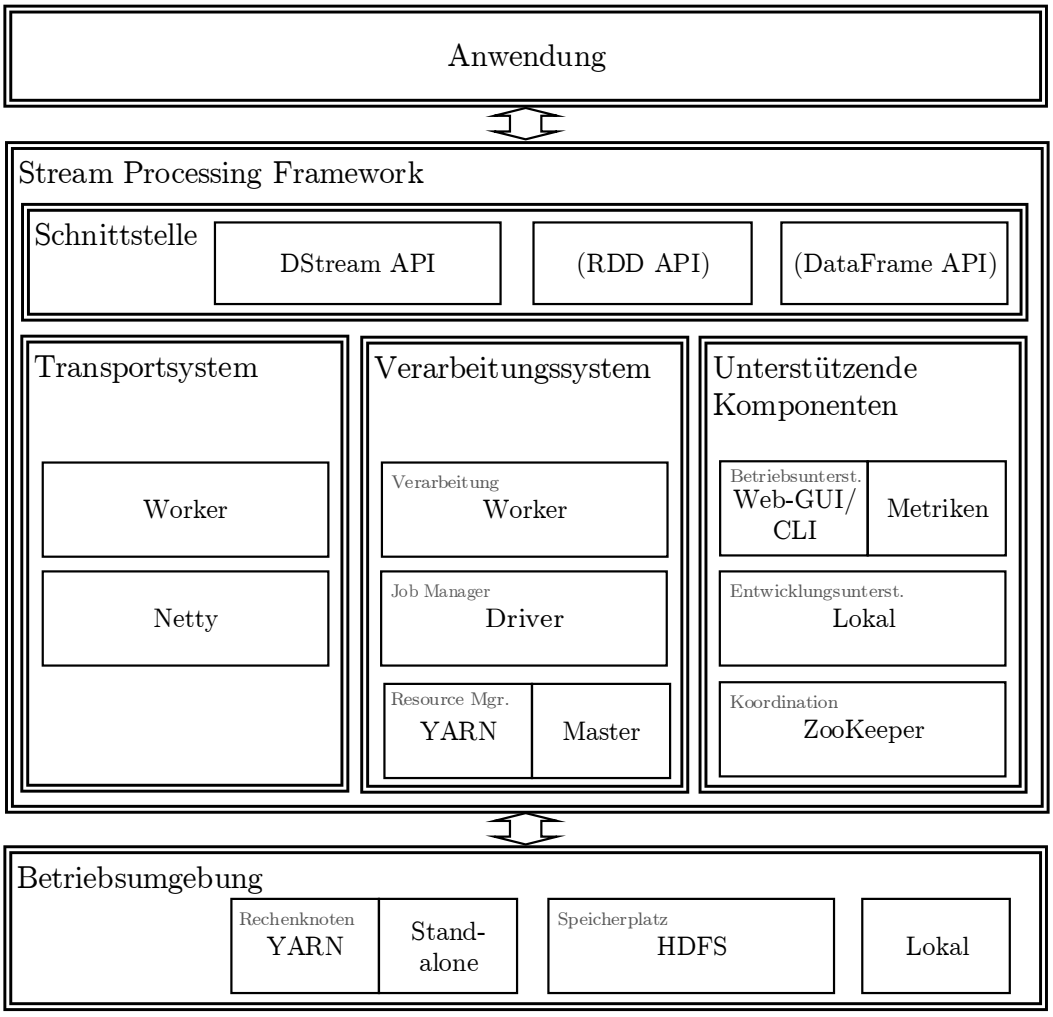


Abbildung 5.3: Architektur von Apache Spark

Die Architektur von Spark ist in Abbildung 5.3 dargestellt. Spark bietet wie Flink mehrere Schnittstellen zum Erstellen von Anwendungen an. Für die Streamverarbeitung ist nur die ***DStream API*** relevant. **Worker** verarbeiten die Daten, transportieren sie an andere Worker mithilfe des Frameworks **Netty** weiter und nutzen Checkpoints, um Fehlertoleranz herzustellen. Der **Master** verwaltet die Ressourcen und ist beim Starten der Anwendung beteiligt. Mithilfe von **ZooKeeper** können mehrere Master eingesetzt werden, um eine

höhere Ausfallsicherheit zu erreichen. Die Clusterverwaltung kann entweder direkt durch Spark im **Standalone**-Betrieb erfolgen oder durch ein externes Werkzeug wie Hadoop **YARN**. Jede Anwendung hat einen eigenen ***Driver***, der den Anwendungskontext namens *SparkContext* hält. Der Driver verwaltet die Anwendung und plant deren Ausführung. Als Ablage für Checkpoints wird das verteilte Dateisystem **HDFS** genutzt. Durch den **lokalen** Betriebsmodus ist einfaches Debuggen möglich, da alle Komponenten in einem Prozess ausgeführt werden. Anwendungen werden durch die **CLI** gestartet und verwaltet. Die **Web-GUI** zeigt laufende Anwendungen und umfangreiche Metriken an. Es lassen sich noch detailliertere Statistiken konfigurieren, die sich auch über eine Web-API abrufen lassen.

5.2.4 Apache Storm

Apache Storm wurde ursprünglich von Twitter entwickelt [Tos+14]. Es ist wie Samza auschließlich auf die Stream-Verarbeitung spezialisiert. Storm besitzt zwei Schnittstellen: die normale Storm API und Storm Trident. Erstere unterstützt keine höheren Abstraktionen, sondern lehnt seine API, wie Samza, an das reine Stream Processing Model an. Die Datenquellen werden *Spouts*, die Verarbeitungsknoten *Bolts* und der Graph *Topology* genannt. Die API bearbeitet jedes Tupel einzeln und nutzt die At-Least-Once-Semantik. Storm Trident bietet eine höhere Abstraktionsebene an und arbeitet mit Micro-Batches. Es stellt eine Exactly-Once-Semantik bereit. Storm ist in Clojure [Hic15] implementiert.

Die Architektur von Storm ist in Abbildung 5.4 dargestellt. Storm bietet sowohl die normale **Storm API** als auch die **Storm Trident API** an. Der **Supervisor** führt die Streamverarbeitung auf den Hosts durch. Die Kommunikation zwischen den Supervisoren wird über **Netty** abgewickelt. ***Acker***, die auf den Supervisoren ausgeführt werden, überwachen die Datenverarbeitung und wiederholen die Verarbeitung bei Fehlern. Die Verwaltung des Clusters und der Ressourcen wird vom Nimbus übernommen. Nur der **Standalone**-Modus ist möglich, die Nutzung von externen Clusterwerkzeugen ist nicht vorgesehen. Anwendungen werden über den ***Nimbus*** bereitgestellt und per ***Thrift*** [ASK07], einem Datenaustauschformat, zu den Supervisoren übertragen. **ZooKeeper** stellt die Verbindung zwischen Nimbus und Supervisor her. Für vereinfachtes Debugging ist ein **lokaler** Betrieb von allen Komponenten in einem Prozess möglich. Anwendungen

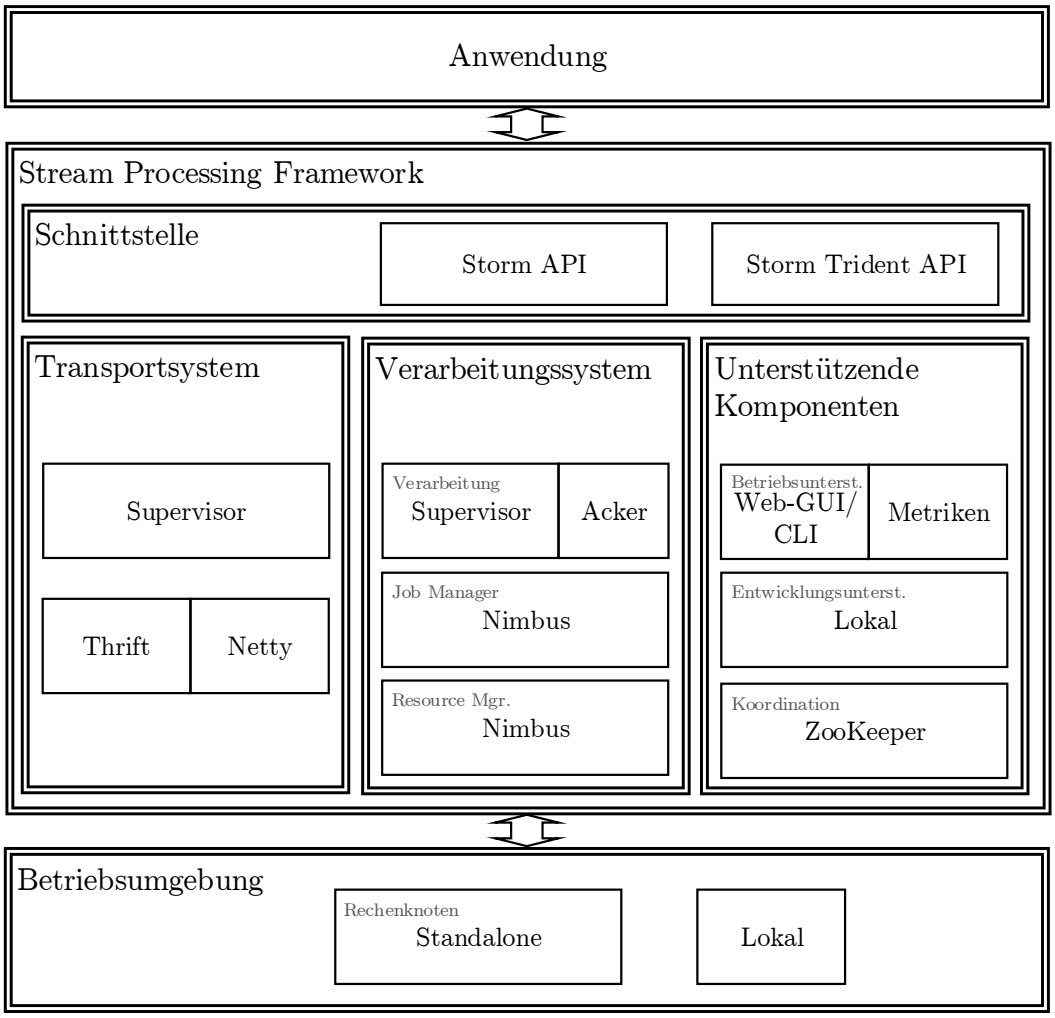


Abbildung 5.4: Architektur von Apache Storm

werden durch die **CLI** bereitgestellt und über die **Web-GUI** überwacht. Storm stellt eigene **Metriken** und eine Schnittstelle für anwendungsspezifische Metriken bereit.

5.3 Methoden

Im Rahmen dieser Evaluierung werden Informationen erhoben, Vergleichskriterien identifiziert und die gewonnenen Informationen bewertet. Jeder dieser Schritte erfordert

spezifische Methoden, um die Objektivität der Evaluierung zu sichern. Die angewandten Methoden werden im Folgenden vorgestellt.

5.3.1 Gewinnen der Informationen

Bei der Informationsgewinnung kommen verschiedene Methoden zum Einsatz. Die bisherige Beschreibung der Frameworks bezog sich ausschließlich auf die vorhandene Literatur. Die im weiteren Verlauf der Evaluierung gesammelten Informationen greifen ebenfalls auf vorhandene Quellen, wie die Dokumentation der Frameworks, wissenschaftliche Arbeiten und sonstige relevante Literatur zurück.

Vorhandene Quellen liefern aber nur einen eingeschränkten Blick auf die praktische Einsatzfähigkeit eines Frameworks. Entscheidende Faktoren, wie die Benutzerfreundlichkeit und die Praxistauglichkeit, lassen sich dadurch nur unzureichend bewerten. Es werden verschiedene Verfahren kombiniert, um eine umfassende Informationsgewinnung zu ermöglichen. Dabei werden sowohl subjektive als auch objektive Methoden genutzt.

Zur Ermittlung der Einsatzfähigkeit und um eine Implementierung für die Leistungsmessung zu erhalten, wird die Methode des *Cognitive Walkthrough* verwendet [Pol+92]. Es handelt sich um eine aufgabenorientierte Usability-Inspektionsmethode [Heg03, S. 24]. Der Evaluator versetzt sich in die Lage des Nutzers und löst typische Aufgaben. Die Erwartung ist, dass Nutzer den Weg des geringsten kognitiven Aufwands gehen und die Software explorativ erkunden. Die Methode bezieht sich ursprünglich auf die Evaluierung von grafischen Oberflächen, lässt sich aber mit Anpassungen auch für Programmieraufgaben nutzen [Bea+08, S. 32-33].

Beim Cognitive Walkthrough werden in der Vorbereitungsphase Aufgaben und entsprechende Handlungssequenzen vorbereitet. Danach wird jede Aufgabe durchgeführt. Abschließend werden die erkannten Probleme analysiert. Um den Rahmen dieser Arbeit zu wahren wird beim Cognitive Walkthrough auf die Vorbereitung von Handlungssequenzen verzichtet und nur die Aufgabenstellung spezifiziert. Wie Beaton u. a. darstellen, ist die genaue Vorbereitung von Handlungssequenzen für Programmieraufgaben aufgrund der vielen Möglichkeiten zur Realisierung eines Ablaufs nicht trivial.

Zusätzlich kommen bei der Bewertung der API *Cognitive Dimensions* [CB03] zum Einsatz. Dies ist eine Methode zur Analyse und Bewertung der Benutzerfreundlichkeit einer

API. Dazu gehören etwa das Abstraktionslevel, die Konsistenz oder die Lesbarkeit der Schnittstelle.

Beide vorgestellten Methoden sind eher den subjektiven Vorgehensweisen zuzuordnen. Sie erfassen weiche Kriterien, die vom Evaluator interpretiert werden müssen. Objektivere eingesetzte Methoden sind Durchsatz- und Latenz-Messungen und der Test der Fehlertoleranz beim Ausfall eines Hosts. [Heg03, S. 15]

5.3.2 Auswahl der Vergleichskriterien

Die Auswahl der Vergleichskriterien erfolgt auf Basis des Qualitätsmodells von ISO 25010 [ISO11a]. Die dort genannten Charakteristiken werden in Subcharakteristiken und letztlich in Qualitätsmerkmale gegliedert. Diesen Qualitätsmerkmalen werden Maße zugeordnet, mit deren Hilfe die Erreichung des Qualitätsmerkmals gemessen werden kann. Grundlage für diesen Prozess ist die Arbeit von Franch und Carvallo [FC02]. Sie stellen im Bezug auf ISO 9126 [ISO99b; ISO01a; ISO01b; ISO01c], einen der Vorläufer von ISO 25000, einen Prozess vor, um aus dem Qualitätsmodell Kriterien für die Evaluierung abzuleiten. Im Folgenden werden die nötigen Schritte erläutert.

1. **Definieren der Domäne**
   Die zu evaluierende Domäne muss hinreichend verstanden werden. Eine uneinheitliche Terminologie kann den Vergleich erschweren. Aus diesem Grund wurden in den vorangegangenen Abschnitten die Grundlagen für die Evaluierung mit der Beschreibung der Domäne, des Gegenstands und der Referenzarchitektur gelegt.

2. **Bestimmung der Subcharakteristiken**
   Die im Standard genannten Subcharakteristiken verfeinern die Charakteristiken weiter. Im Allgemeinen werden sie als Startpunkt genutzt und je nach Domäne angepasst.

3. **Aufbau einer Hierarchie von Subcharakteristiken**
   Subcharakteristiken können in weitere Subcharakteristiken zerlegt werden, falls eine weitere Unterteilung, abhängig von der Domäne, sinnvoll ist. Sie müssen das gleiche Abstraktionsniveau wie vorhandene Subcharakteristiken aufweisen, andernfalls wären es Qualitätsmerkmale, die erst im nächsten Schritt definiert werden.

4. **Zerlegung der Subcharakteristiken in Qualitätsmerkmale**
   Subcharakteristiken fassen einen Bereich des Qualitätsmodells zusammen. Qualitätsmerkmale verfeinern Subcharakteristiken. Sie beziehen sich auf eine bestimmte, erfassbare Eigenschaft des Gegenstands. Bedingt durch die Vielfalt der Eigenschaften des Gegenstands können, nicht alle Qualitätsmerkmale erschöpfend erfasst werden, daher beschränkt man sich auf die wichtigsten Konzepte. Es ist möglich, ein Merkmal unter mehreren Subcharakteristiken einzuordnen und dadurch später verschiedene Gesichtspunkte des Merkmals zu erfassen.
5. **Zerlegung von abgeleiteten Merkmalen in elementare Merkmale**
   Manche Qualitätsmerkmale sind noch zu grob, um sie direkt zu messen. Falls dies der Fall ist, wird das Merkmal so lange zerlegt, bis nur noch direkt messbare, elementare Merkmale vorhanden sind.
6. **Bestimmung der Maße für elementare Merkmale**
   Für jedes elementare Merkmal müssen Maße identifiziert werden, um eine Bewertung zu ermöglichen. Es bieten sich mathematische Verfahren zur Beschreibung der Maße an.

5.4 Kriterien

Wie im vorherigen Abschnitt genannt, werden die Evaluierungskriterien aus dem Qualitätsmodell von ISO 25010 mithilfe des dargestellten Prozesses abgeleitet. Da es sich bei Stream Processing Frameworks um Platform Frameworks handelt, haben sie weitreichenden Einfluss auf die darauf aufbauende Anwendung. Daher werden alle Charakteristiken in die Evaluierung einbezogen. Die im Standard erwähnten Subcharakteristiken werden als Leitfaden verwendet und anhand dessen die Qualitätsmerkmale gebildet. Entsprechend des Prozesses und dem Umfang dieser Arbeit werden die wichtigsten Merkmale erfasst. Im Folgenden werden die Kriterien den Charakteristiken nach aufgeführt.

5.4.1 Functional Suitability

Die Charakteristik Functional Suitablility ist in drei Subcharakteristiken aufgeteilt. *Functional Completeness* nimmt für diese Evaluierung die höchste Priorität ein. Sie beschreibt,

inwieweit der Funktionsumfang den Anforderungen entspricht. Die *Functional Correctness* wird nicht explizit berücksichtigt, wird aber durch die Durchführung der Aufgaben implizit geprüft. *Functional Appropriateness* beschreibt, wie gut die Lösung von Aufgaben unterstützt wird.

Unterstützte Programmiersprachen

Da das Framework die Grundlage für die Anwendung bildet, ist die Entwicklungs- und Laufzeitumgebung auf die Möglichkeiten des Frameworks beschränkt. Wenn noch andere Bibliotheken oder vorhandene Applikationen eingebunden werden sollen, ist es vorteilhaft, möglichst viele Programmiersprachen zu unterstützten. Zusätzlich gibt es den Ansatz, für jeden Einsatzzweck die am besten geeignete Programmiersprache zu benutzen. Dieses Paradigma ist unter dem Namen *Polyglot Programming* bekannt [WC10]. Es wird untersucht, welche Programmiersprachen unterstützt werden und ob diese innerhalb einer Applikation gemischt werden können.

Mächtigkeit der angebotenen API

Die API stellt im Entwicklungsprozess den größten Teil der Interaktion mit dem Framework dar. Eine mächtige, mit vielen Funktionen ausgestattete API ist daher von hoher Priorität. Je nach Ausrichtung des Frameworks hat die API einen anderen Abstraktionsgrad. Bei einem höheren Abstraktionsgrad sind die bereitgestellten Operationen der essentielle Teil der API. Bei einem niedrigeren Abstraktionsgrad gibt es Operationen in dieser Form nicht, stattdessen werden die Verarbeitungsknoten und deren Verbindungen definiert. Auch dort gibt es funktionale Unterschiede, etwa bei den möglichen Verbindungstypen. Die Art und die Anzahl der zur Verfügung stehenden Operationen und Verbindungstypen wird überprüft. Es wird untersucht, wie Window-Operationen genutzt beziehungsweise realisiert werden können. Zusätzlich wird erfasst, wie die Initialisierung und Finalisierung von Operatoren beziehungsweise Verbindungsknoten gehandhabt wird.

Unterstützung von zustandsbehafteten Operationen

Wie in den Grundlagen erläutert, müssen bei zustandsbehafteten Operationen zur Sicherstellung der Fehlertoleranz besondere Maßnahmen getroffen werden. Es wird untersucht,

inwieweit das Framework die Entwicklung und den Betrieb von zustandsbehafteten Operationen unterstützt. Dazu zählt das Lesen und Schreiben des Zustands und die Integration in die vorhanden Operationen.

5.4.2 Performance Efficiency

Die Leistungsfähigkeit eines Stream Processing Frameworks ist ein wesentliches Merkmal. ISO 25010 unterscheidet drei Arten, die Leistungsfähigkeit eines Systems zu beurteilen. Die Verarbeitungszeit wird mit der Subcharakteristik *Time Behaviour* erfasst. *Resource Utilization* umfasst die Ressourcennutzung des Systems bei der Erfüllung seiner Aufgabe. Die Kapazitätsgrenze, bis zu der Aufgaben des Systems mit akzeptabler Geschwindigkeit erledigt werden, erfasst die Subcharakteristik *Capacity*.

Mittlere Verarbeitungszeit von Tupeln

Die mittlere Zeit vom Absetzen des Tupels in das Quellsystem bis zum Speichern im Zielsystem wird ermittelt. Dies geschieht bei einem steigenden Durchsatz so lange, bis die Sättigung des Frameworks eintritt und die Latenz immer weiter steigt. Dieses Kriterium erfasst die Aspekte Time Behaviour und Capacity.

Skalierung auf mehrere Worker

Um die Resource Utilization zu prüfen, wird die obig vorgestellte Messung mit verschiedenen Anzahlen an Workern durchgeführt. Dadurch kann die Skalierung auf mehrere Hosts geprüft werden. Die Skalierung sollte möglichst linear sein, damit das System mit einer wachsenden Anzahl an Workern gleichsam leistungsfähiger wird. Zusätzlich wird geprüft, wie das Framework die Last auf die vorhandenen Worker verteilt.

5.4.3 Compatibility

Die Charakteristik Compatability beschäftigt sich damit, wie gut sich ein System in eine bestehende Umgebung einfügt. Sie wird in zwei Subcharakteristiken aufgeteilt: *Co-Existence* beschreibt, wie das System seine Aufgaben erfüllt, wenn es sich eine Umgebung und Ressourcen mit einem anderen System teilen muss. *Interoperability* beschreibt, wie

das System mit anderen Systemen zusammenarbeitet und Informationen austauschen kann.

Integration mit bestehenden Clusterwerkzeugen

Apache Hadoop ist im Big Data Umfeld der Quasi-Standard bei der Batch-Verarbeitung von Daten, deshalb sind Hadoop-Installationen bereits in einigen Unternehmen verbreitet [Gar15]. Da die Installation von mehreren Clusterwerkzeugen zu Ressourcenkonflikten führt und der Aufbau eines weiteren Clusters unter organisatorischen und finanziellen Gesichtspunkten nicht immer in Frage kommt, ist die Nutzung von bereits vorhanden Clusterwerkzeugen anzustreben. Es wird geprüft, welche Bereitstellungsoptionen ein Framework bietet. Dieses Kriterium deckt den Aspekt Co-Existence ab.

Anbindung von Ein- und Ausgabesystemen

Ein Stream Processing Framework kann von verschiedenen Datenquellen mit Tupeln versorgt werden. Ein Anwendungsfall ist die Integration von Daten aus verschiedenen Quellen. Mitgelieferte Anbindungen von Datenquellen an das Framework vereinfachen dies. Daher wird geprüft, welche Anbindungen beiliegen. Andererseits kann es nötig werden, eigene Anbindungen zu entwickeln. Daher wird geprüft, welche Schritte dafür nötig sind und welche Implikationen dies auf die Fehlertoleranz hat. Für die Anbindung von Ausgabesystemen wird ebenfalls geprüft, welche Systeme vom Framework direkt unterstützt werden und wie ein neues System angebunden werden kann. Hierbei wird der Aspekt Interoperability geprüft.

5.4.4 Usability

Die Benutzerfreundlichkeit ist ein wichtiger Faktor bei der Entwicklung und beim Betrieb einer Anwendung [Hen09]. Dementsprechend groß ist die Anzahl an Subcharakteristiken. Die *Appropriateness Recognizability* bezieht sich darauf, wie gut man die Tauglichkeit des Systems anhand der Dokumentation beurteilen kann. Wie einfach man die Benutzung eines Systems erlernen kann, wird als *Learnability* bezeichnet. Der einfache Betrieb eines Systems wird unter der Subcharakteristik *Operability* zusammengefasst. *User Error Protection* erfasst, wie das System Nutzer davor schützt Fehler zu machen. Normalerweise beschreibt

der Bereich *User Interface Aesthetics*, wie gut sich die Benutzeroberfläche bedienen lässt. Da der Nutzer bei einem Framework hauptsächlich mit der API interagiert, wird in diesem Fall die API auch zum User Interface hinzu gezählt. Die letzte Subcharakteristik *Accessibility*, also die Barrierefreiheit, wird für diese Arbeit als nicht relevant betrachtet, da kein Endbenutzer, sondern nur ein sehr eingeschränkter Nutzerkreis mit dem Framework interagiert.

Dokumentation des Frameworks

Der Begriff Dokumentation beschreibt in diesem Kontext mehrere Arten von Dokumenten, unter anderem eine Funktionsbeschreibung, die Benutzerdokumentation, die Beschreibung der API, Hilfetexte von Anwendungen, Beispiele und Tutorials. All diese Dokumente tragen zu einem besseren Verständnis des Frameworks und zum Erlernen der Benutzung bei. Es wird geprüft, welche Dokumentation vorhanden ist, wie umfangreich diese ist und ob die Dokumentation mit der Praxis übereinstimmt. Die Aspekte Appropriateness Recognizability und Learnability werden damit abgedeckt.

Ersteinrichtung des Frameworks

Das Framework muss sowohl auf dem Cluster, als auch auf dem Entwicklungssystem installiert und konfiguriert werden. Beide Varianten haben andere Anforderungen und nutzen daher potentiell andere Konfigurationen. Je nach Framework werden zusätzliche Komponenten benötigt, die ebenfalls installiert und eingerichtet werden müssen. Der gesamte Installationsprozess vom Download des Frameworks bis zum funktionierenden System wird anhand Dokumentation durchgeführt. Dieses Kriterium erfasst den Aspekt Learnability.

Bereitstellung von Anwendungen

Anwendungen müssen für die Ausführung auf dem Cluster vorbereitet und letztlich dort bereitgestellt werden. Dieser Prozess umfasst auch das Beenden und Aktualisieren von Anwendungen. Es wird untersucht, wie diese Prozesse mit den Frameworks realisiert werden. Der Aspekt Operability wird dadurch abgedeckt.

Benutzerfreundlichkeit der API

Anhand der Cognitive Dimensions wird geprüft, wie benutzerfreundlich die API ist. Sie sind mit zwölf Kriterien sehr umfangreich [CB03]. Um den Rahmen dieser Arbeit zu wahren, werden drei Dimensionen ausgewählt, die beim Erlernen des Frameworks eine Rolle spielen.

Das *Working Framework* beschreibt, aus welchem Kontext Informationen bei der Benutzung der API einbezogen werden müssen. Dabei muss entweder nur die aktuelle Aufgabe, die gesamte Anwendung oder das gesamte System beachtet werden. Ein möglichst geringer Kontext ist erstrebenswert, um sich auf die Aufgabe konzentrieren zu können. [Cla03]

Die *Penetrability* gibt an, wie stark Implementierungsdetails zur Benutzung der API verstanden werden müssen und wie gut sie durch die API und die Dokumentation erklärt werden. Es können nur wenige Details, Informationen über den Kontext oder ein tiefes Verständnis nötig sein. Diese Dimension ist vor allem beim Erlernen des Frameworks wichtig, um intuitiv die Benutzung zu verstehen. [Cla04b]

Die *Consistency* beschreibt, wie gut Konzepte aus einem Teilbereich der API auf die gesamte API übertragbar sind. Die API kann vollständig, nur in Teilbereichen oder nicht konsistent sein. Nutzer erwarten, dass sich verschiedene Bereiche einer API gleich benutzen lassen und sich ähnlich verhalten. [Cla04a]

Dieses Kriterium wird anhand der im Cognitive Walkthrough durchgeführten Aufgaben erfasst. Es umfasst die Subcharakteristiken User Error Protection, User Interface Aesthetics und Learnability.

5.4.5 Reliability

Die Zuverlässigkeit eines Systems wird mit vier Subcharakteristiken beschrieben. *Maturity* beschreibt die Zuverlässigkeit des Systems unter Normalbedingungen. Die weiteren Subcharakteristiken *Availability*, *Fault Tolerance* und *Recoverability* drehen sich um die Erkennung, Vermeidung und Behandlung von Fehlersituationen.

Aktive Community

Die evaluierten Frameworks werden als Open Source Software entwickelt, deshalb ist eine aktive Community ein bedeutender Faktor bei ihrem Einsatz. Da ein aktiv entwickeltes und benutztes Softwareprojekt einen höheren Reifegrad verspricht als ein Verwaistes, wird die Anzahl der freigegebenen Versionen, der Commits, der Commiter und der Mailinglistenbeiträge der letzten zwölf Monate erfasst und verglichen. Dieses Kriterium umfasst den Aspekt Maturity.

Verarbeitungsgarantien und Fehlertoleranz

Die Herstellung der Fehlertoleranz ist eines der Hauptfunktionen eines Stream Processing Frameworks. Deshalb wird die bereitgestellte Verarbeitungsgarantie dokumentiert und der Mechanismus dahinter erfasst. Dabei werden Master und Worker betrachtet. Zusätzlich wird die Auswirkung eines Worker-Ausfalls praktisch getestet und die resultierende Latenz untersucht. Die drei Subcharakteristiken Availability, Fault Tolerance und Recoverability werden von diesem Kriterium erfasst.

5.4.6 Security

Im Kontext von Big Data ergeben sich vor allem bei personenbezogenen Daten datenschutzrechtliche Anforderungen, die darauf abzielen, dass die Daten sicher verarbeitet und gespeichert werden [Dor15, S. 188]. Dies bedeutet, dass einerseits Zugriffe nur von berechtigten Benutzern erfolgen dürfen, und andererseits, dass die Daten verschlüsselt ausgetauscht werden sollten. Diese Aspekte umfassen die klassischen Subcharakteristiken *Integrity*, *Authenticity* und *Confidentiality* [WM12, S. 13 f.]. *Non-Repudiation*, also die Nichtabstreitbarkeit von Aktionen [WM12, S. 375], und *Accountability*, also die Rückverfolgbarkeit von Aktionen [WM12, S. 250], werden als Anforderungen an Stream Processing Systeme eher eine geringe Priorität zugesprochen, da Endnutzer normalerweise nicht direkt mit dem System interagieren. Daher werden die beiden Aspekte nur im Rahmen der Untersuchung der Logdateien in Abschnitt 5.4.7 betrachtet.

Nutzerbasierte Zugriffskontrolle

Wenn mehrere verschiedene Nutzer auf einem Cluster arbeiten, ist eine gegenseitige Isolation wünschenswert, sodass sich mehrere Nutzer bis auf ihren Ressourcenverbrauch nicht beeinflussen. Es wird geprüft, bis zu welchem Grad mehrere Nutzer voneinander isoliert werden können.

Verschlüsselung des Datenverkehrs

Wenn sensible Daten verarbeitet werden, ist es wünschenswert, dass sie während der Übertragung sicher verschlüsselt werden. Davon abgesehen sind Schnittstellen zur Verwaltung des Frameworks ebenfalls abzusichern. Daher wird erfasst, welche Sicherheitsmaßnahmen beim Transport von Daten über das Netzwerk konfiguriert werden können.

5.4.7 Maintainability

Einer der Vorteile von fertigen Frameworks gegenüber Eigenentwicklungen soll eine bessere Wartbarkeit sein. Daher sollte ein Framework eine gute Wartbarkeit der Anwendung gewährleisten. Die Charakteristik Maintainability umfasst fünf verschiedene Subcharakteristiken: Die *Modularity* wird nicht berücksichtigt, da diese hauptsächlich für die Weiterentwicklung des Frameworks selbst und nicht für die Nutzer des Frameworks relevant ist. Die *Reuseability* ist bei einem Framework generell gegeben, daher wird es nicht gesondert untersucht. *Analysability* beschreibt, wie effektiv die Ursache eines Problems untersucht werden kann. *Modifiability* bezeichnet die Möglichkeit das Framework zu modifizieren, ohne Einschränkungen zu verursachen. Die *Testability* charakterisiert, wie einfach überprüft werden kann, ob ein System den Anforderungen entspricht.

Aktualisieren des Frameworks

Alle getesteten Frameworks befinden sich noch stark in Entwicklung. Bisher nutzen alle Frameworks außer Apache Spark eine Hauptversionsnummer kleiner 1. Daher sind Aktualisierungen des Frameworks nicht ungewöhnlich. Es wird geprüft, welche Schritte für eine Aktualisierung nötig sind und inwieweit die Aktualisierung die Verfügbarkeit des Systems beeinträchtigt. Dieses Kriterium deckt den Bereich Modifiability ab.

Erfassung von Metriken

Metriken, wie etwa die Anzahl der verarbeiteten Tupel oder die Verarbeitungslatenz, können während der Entwicklung Hinweise liefern, wo weitere Optimierung nötig ist. Auch während dem Betrieb können so Fehlersituationen erkannt und eingegrenzt werden. Daher wird untersucht, welche Metriken abgerufen werden können und wie diese zugänglich gemacht werden. Der Aspekt Analysability wird durch dieses Kriterium erfasst.

Aussagekräftige Logdaten

Ähnlich wie Metriken sind aussagekräftige Logdaten während der Entwicklung und dem Betrieb notwendig. Vor allem in Fehlersituationen helfen Logdaten bei der Analyse der Ursache. Zusätzlich können sicherheitsrelevante Vorgänge, wie Anmeldungen oder Aktionen eines Nutzers, nachvollzogen werden. Es wird geprüft, welche Logdaten zur Verfügung stehen, wie sie konfiguriert werden und wie darauf zugegriffen wird. Die Analysability wird durch dieses Kriterium unterstützt.

Testen auf dem Entwicklersystem

Für eine einfache Entwicklung ist das Testen auf dem Entwicklersystem unerlässlich [GW04]. Es gibt mehrere Anforderungen an die Testbarkeit. Sowohl *Unit-Tests* von einzelnen Funktionen sollten möglich sein, als auch die Ausführung des kompletten Frameworks aus der Entwicklungsumgebung heraus, mit der Möglichkeit die Anwendung zu debuggen. Es wird geprüft, welche dieser Möglichkeiten vorhanden sind.

5.4.8 Portability

Die Charakteristik Portability beschreibt, wie gut ein System von einer Umgebung in eine andere Umgebung überführt werden kann. Die Subcharakteristiken dieses Aspekts werden bereits in anderen Charakteristiken behandelt. Die *Adaptability*, also die Anpassung an verschiedene Umgebungen, wird mit den Kriterien im Bereich Compatibility und mit dem Kriterium Skalierung auf mehrere Worker abgedeckt. Der Schwierigkeitsgrad der Installation, die *Installability*, wird bereits im Kriterium Ersteinrichtung des Frameworks geprüft. Das Ersetzen des Systems mit einer neueren Version wird mit dem Kriterium Aktualisieren des Frameworks getestet. Ob ein System durch ein anderes ersetzt werden

kann, wird nicht explizit geprüft, daher wird die Subcharakteristik *Replaceability* nicht berücksichtigt.

5.4.9 Assigned Property

Assigned Properties sind extern zugewiesene Eigenschaften des Softwaresystems. Sie werden nicht durch Subcharakteristiken verfeinert.

Verfügbarkeit externer Unterstützung

Neben der Unterstützung durch die Community, die im Abschnitt 5.4.5 untersucht wird, gibt es noch andere Formen der Unterstützung außerhalb des Projekts. Im Internet gibt es weitere externe Seiten mit Anleitungen, Tutorials, Blogeinträgen und Forenbeiträgen. Daher werden die Anzahl der Google-Treffer für die jeweiligen Frameworks und die Anzahl der Ergebnisse bei der beliebten[1] Frage- und Antwortplattform Stackoverflow.com ausgewertet. Außerdem wird die Anzahl der Bücher, in denen ein Framework erwähnt wird, auf Amazon.com ermittelt. Zusätzlich wird festgestellt, ob das Framework in eine Distribution mit kommerzieller Unterstützung eingebunden ist.

5.5 Aufgaben

Die Evaluierung wird anhand eines vorgegebenen Aufgabenkatalogs durchgeführt, der mit jedem Stream Processing Framework durchgeführt wird. Die Aufgaben werden sowohl für die Usability-Inspektion im Rahmen des Cognitive Walkthrough, als auch bei den Leistungsmessungen genutzt. Es werden typische Aufgabenstellungen prototypisch implementiert. Die Aufgaben werden primär zur Informationsgewinnung im Rahmen des Vergleichs genutzt, daher nehmen sie nur wenig Rücksicht auf spezielle fachliche Anforderungen.

Der Rahmen für alle Aufgaben ist die Echtzeitüberwachung von Geldkursen (*Bid Price*) der New York Stock Exchange (NYSE). Alle Frameworks werden mit der Abhängigkeitsverwaltung Maven [Apa15z] in der Entwicklungsumgebung Eclipse [Ecl15] genutzt. Die

[1] Platz 1 laut Alexa.com in der Kategorie Computer, Programmieren. http://www.alexa.com/topsites/category/Computers/Programming - zuletzt abgerufen am 12.08.2015

entsprechenden Angaben zur Maven-Abhängigkeit werden von der Webseite des Frameworks bezogen. Die Implementierung der Aufgaben erfolgt jeweils so, dass die verfügbaren Hilfestellungen des Frameworks bestmöglich genutzt werden, auch wenn dadurch die Vergleichbarkeit der Leistungsmessung beeinträchtigt wird. Dies trifft insbesondere auf Window-Operationen zu, die nicht bei allen Frameworks implementiert sind.

5.5.1 Installation des Frameworks

Die Installation des Frameworks ist der erste Schritt, um ein Framework zu nutzen. Das Framework wird auf einem Cluster installiert und konfiguriert. Es werden ein Master und vier Worker aufgesetzt. Zur Vereinfachung des Bereitstellungsprozesses wird eine Netzwerkfreigabe genutzt, in der das Framework für alle Hosts im Cluster abgelegt wird. Das Framework wird von der offiziellen Webseite in der aktuellen Version heruntergeladen und laut Anleitung im Standalone-Modus installiert und konfiguriert. Durch diese Aufgabe wird das Kriterium Ersteinrichtung des Frameworks geprüft.

5.5.2 Gleitende Statistik

Für alle eingehenden Wertpapier-Informationen wird eine gleitende Statistik des Geldkurses über die letzten fünf Minuten im Sekundentakt gebildet. Die Statistik wird mindestens den Durchschnitt und die Standardabweichung enthalten und jede Sekunde in einer Datenbank abgelegt, wo sie externen Anwendungen zugänglich ist. Zusätzlich wird die Statistik intern vorgehalten, um sie für weitere Auswertungen zu nutzen. Für die Implementierung wird die Apache Commons Math [Apa15a] Bibliothek verwendet. Diese Aufgabe wird zur Prüfung der Kriterien Unterstützte Programmiersprachen und Mächtigkeit der angebotenen API verwendet.

5.5.3 Meldung bei großen Kursabweichungen

Signifikate Abweichungen des Geldkurses können ein Signal für die Neubewertung eines Wertpapiers sein. Daher wird ein Verfahren implementiert, dass jeden eingehenden Geldkurs mit der aktuellen Statistik für das jeweilige Wertpapier vergleicht. Eine signifikante Abweichung wird als das 6-fache der Standardabweichung definiert. Beim Über- oder Unterschreiten dieser Schwelle wird eine Meldung ausgelöst, die an ein anderes System

zur weiteren Verarbeitung übergeben wird. Anhand dieser Aufgabe werden die Kriterien Mittlere Verarbeitungszeit von Tupeln und Verarbeitungsgarantien und Fehlertoleranz geprüft.

5.5.4 Bewertung von Wertpapierdepots

Auf der Grundlage der Statistik werden in einer Datenbank vorhandene Wertpapierdepots bewertet. Mindestens einmal pro Sekunde wird der Depotwert anhand der aktuellen Statistik der enthaltenen Wertpapiere berechnet. Dazu wird für jede Position im Depot der durchschnittliche Geldkurs mit der Menge an Wertpapieren multipliziert und über alle Positionen aufsummiert. Das Ergebnis wird zur Weiterverarbeitung in einer Datenbank abgelegt. Diese Aufgabe unterstützt die Bewertung des Kriteriums Anbindung von Ein- und Ausgabesystemen.

6 Praktische Umsetzung

Die im vorhergehenden Unterkapitel vorstellten Aufgaben werden im Verlauf dieses Kapitels implementiert. Zunächst wird die logische Architektur der Anwendung erläutert. Danach folgen die Implementierungen mit den konkreten Frameworks. Alle Artefakte die im Rahmen der praktischen Umsetzung erstellt wurden, wie der Quelltext und die Leistungsmessungen, sind auf dem beiliegenden Datenträger enthalten.

6.1 Entwicklungs- und Betriebsumgebung

Als Betriebsumgebung wird der Big Data Cluster der Hochschule Darmstadt verwendet [Stö15]. Es stehen sechs Rechenknoten zur Verfügung. In Tabelle 6.1 werden die technischen Daten der Rechenknoten aufgelistet.

Modell	Dell PowerEdge C6220
Prozessor	2x Intel Xeon E5-2609 (je 4 Cores)
Arbeitsspeicher	32 GB DDR3 ECC 1600 MHz
Festplatte	4x Seagate ST91000640NS 1 TB SATA
Netzwerk	Intel I350 1 Gbit/s
Ethernet Switch	Dell PowerConnect 7048R-RA

Tabelle 6.1: Technische Daten der Rechenknoten [Stö15]

Die zur Entwicklung und zum Betrieb eingesetzte Software wird in Tabelle 6.2 aufgelistet. Es wird die jeweils aktuellste veröffentlichte Version am Stichtag, den 20. Juni 2015, verwendet. Es sind nur die manuell installierten Komponenten aufgelistet. Abhängigkeiten dieser Komponenten werden automatisch durch Maven beziehungsweise die Paketverwaltung des Betriebssystems aufgelöst.

Die Rechenknoten werden vor der Installation der Frameworks vorbereitet. Auf jedem Host wird die aktuelle Java-Version installiert, die Zeitsynchronisierung per *Network Time Protocol* [NTP15] eingerichtet und ein gemeinsames Verzeichnis per Samba-Netzwerkfreigabe [Sam15] konfiguriert. Es wird sichergestellt, dass jeder Host mit jedem anderen per *Secure Shell (ssh)* mit *Public-Key-Authentifizierung* [Can15a] kommunizieren kann. Zusätzlich werden alle im nächsten Unterkapitel 6.2 vorgestellten Systeme laut deren Anleitung installiert und konfiguriert.

Name	Version	Quelle
Apache Commons Math	3.5	[Apa15a]
Apache Flink	0.9.0	[Apa15q]
Apache Hadoop	2.7.1	[Apa15s]
Apache Kafka	0.8.2.1 (Scala 2.10)	[Apa15x]
Apache Maven	3.0.5	[Apa15z]
Apache Samza	0.9.0	[Apa15ae]
Apache Spark	1.4.0 (Hadoop 2.6)	[Apa15av]
Apache Storm	0.10.0-beta1	[Apa15bj]
Apache ZooKeeper	3.4.6	[Apa15bn]
Eclipse IDE for Java Developers	4.5.0 Mars	[Ecl15]
Google Guava	18.0	[Goo15b]
HikariCP	2.3.8	[Woo15]
Jedis	2.7.3	[Lei15]
JUnit	4.12	[JUn15]
Kryo	3.0.1	[Eso15]
ntpd	4.2.6p5	[NTP15]
Oracle Java SE Development Kit	1.8.0_51 64-Bit	[Ora15b]
PostgreSQL	9.3.7	[Pos15]
Python	2.7.6	[Pyt15]
Redis	3.0.2	[San15a]
Samba	4.1.6-Ubuntu	[Sam15]
sql2o	1.5.4	[Aab15]
Ubuntu	14.04 LTS Server	[Can15b]

Tabelle 6.2: Genutzte Softwareversionen

6.2 Architektur

Die Architektur der Anwendung wird aus den drei in den Abschnitten 5.5.2, 5.5.3 und 5.5.4 beschriebenen Aufgaben hergeleitet. Die Aufgaben sollen typische Anwendungsfälle eines Stream Processing Frameworks abbilden. Außerdem wird die fertige Implementierung zur Leistungsmessung herangezogen. Daher werden neben dem Framework weitere Komponenten in die Architektur integriert. Die Erfassung der Ende-zu-Ende-Latenz, also vom Quell- bis zum Zielsystem, zur Leistungsmessung ist in diesem Fall am sinnvollsten, da dadurch keine speziellen Voraussetzungen bestehen oder Änderungen am Stream Processing Framework vorzunehmen sind. Die Messwerte werden nur untereinander verglichen, daher spielt die zusätzliche Latenz durch beteiligte Systeme keine Rolle. Es wird eine ausreichende Kapazität der Quell- und Zielsysteme sichergestellt, damit durch sie kein Flaschenhals entsteht.

In Abbildung 6.1 wird die Verteilung der einzelnen Komponenten auf die Hosts dargestellt. Harrenhal beherbergt die Master-Komponenten, auf Pinkmaiden befinden sich die

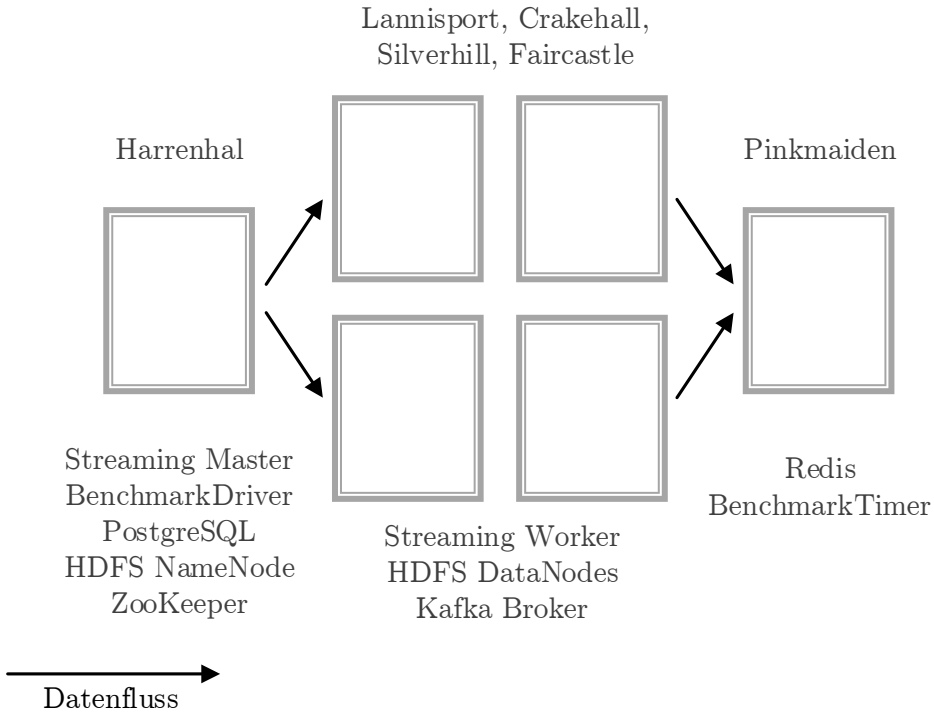


Abbildung 6.1: Physische Verteilung der Anwendung mit Zuordnung der Hostnamen und installierten Komponenten

Zielsysteme und die vier restlichen Rechenknoten sind Worker. Abbildung 6.2 zeigt die

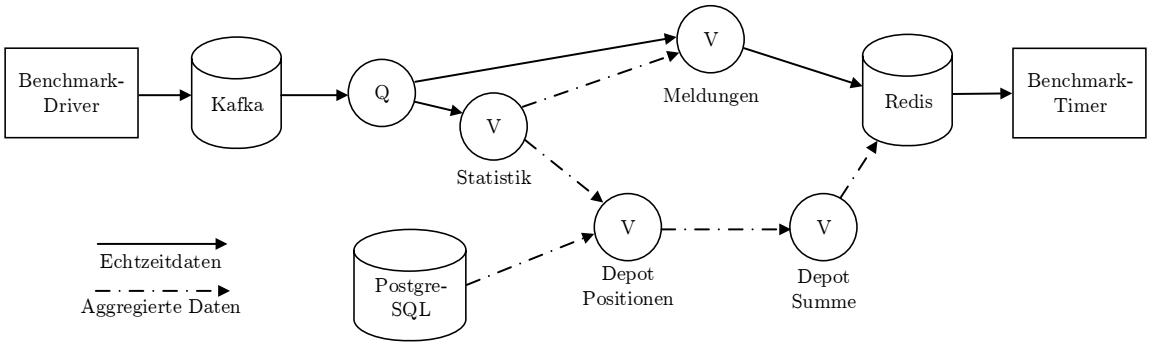


Abbildung 6.2: Logische Architektur der Anwendung

Architektur der kompletten Anwendung auf logischer Ebene. Die beteiligten Komponenten lassen sich in vier Gruppen einteilen, die im Folgenden einzeln beschrieben werden.

6.2.1 Datenquelle

Die Datenquelle umfasst die genutzten Testdaten, den Zuspieler BenchmarkDriver und das Publish-Subscribe-System Apache Kafka. Durch Messungen wurde sichergestellt, dass die hier vorgestellte Konfiguration mindestens 935.000 Tupel pro Sekunde emittieren kann.

Daten

Als Testdaten werden *Daily Quotes* der New York Stock Exchange verwendet [NYS14]. Diese Daten umfassen den niedrigsten Briefkurs (Verkaufspreis) und den höchsten Geldkurs (Kaufpreis) jedes an der NYSE gehandelten Wertpapiers. Zusätzlich sind Angaben wie der Zeitpunkt, das Symbol, die jeweilige Menge an Wertpapieren, die beteiligten Handelsplätze und weitere Daten enthalten. Sobald im Handelssystem eine Wertpapierorder platziert wird, die den aktuell besten Briefkurs unterschreitet beziehungsweise den Geldkurs überschreitet, wird ein neuer Datensatz generiert. Der Datensatz wird mit der aktuellen Zeit und einer Sequenznummer versehen und gespeichert. Die Daten können in Echtzeit von der NYSE bezogen werden. Bei diesem Test kommen historische Daten zum Einsatz, die im Wesentlichen den Echtzeitdaten entsprechen. Sie wurden gewählt, da sie in einem einfachen ASCII-basierten Datenformat vorliegen und die Echtzeitdaten nur als Netzwerk-Mitschnitt des echten Multicast-Datenstroms im *tcpdump*-Format vorliegen. Ein Beispieldatensatz

ist in Quelltext 6.1 abgebildet. Alle verwendeten Daten beziehen sich auf die Beispieldatei „EQY_US_ALL_BBO_20141030“ vom 30.10.2014. Es wird der Zeitraum von 10:00:00.000 Uhr bis 10:00:10.999 Uhr verwendet. Dieser Abschnitt enthält 1.314.242 Datensätze von 6.083 Wertpapieren.

```
100001298|K|GOOG            |00005445500|0000001|00005451600|0000001|R    KK|0000000003655587| 2 N      A
Zeitpunkt| |Symbol          |Geldkurs   |G-Menge|Briefkurs  |B-Menge|       |Sequenznummer   |
```

Quelltext 6.1: Beispieldatensatz aus „EQY_US_ALL_BBO_20141030“ mit ausgewählten Beschriftungen (senkrechte Stiche nachträglich eingefügt)

BenchmarkDriver

Vor der Implementierung eines eigenen Zuspielers wurden vorhandene Lösungen untersucht. Mendes, Bizarro und Marques haben bereits das Werkzeug FINCoS [MBM13] vorgestellt. Die letzte Version wurde im März 2013 veröffentlicht. Es ist aber weniger auf verteilte Frameworks ausgelegt, daher wäre ein größerer Aufwand nötig, um FINCoS für diese Arbeit anzupassen.

Der BenchmarkDriver wurde im Rahmen dieser Arbeit implementiert. Er ist in Java geschrieben und wird für das Einspielen der Testdaten in das Framework verwendet. Die Testdaten werden initial als String aus einer Datei eingelesen, in ein `Quote`-Objekt umgewandelt, mit Kryo [Eso15] in ein Bytearray serialisiert und im Arbeitsspeicher gehalten. Dadurch entfällt die Umwandlung beim Senden der Objekte an Kafka.

Die Daten werden bei einem Testlauf in einer Endlosschleife abgespielt. Die Geschwindigkeit, mit der die Daten abgespielt werden, wird in 10-Sekunden-Schritten erhöht und folgt der Formel: Tupel pro Sekunde $= 500 \cdot e^{0{,}04 \cdot \text{Schritt}}$. Abbildung 6.3 visualisiert die Formel. Sie gewährleistet einen großen Testbereich und gleichbleibende relative Abstände zwischen den Schritten. Die langsame Anlaufphase baut die Statistik aus der Aufgabe in Abschnitt 5.5.2 vollständig auf. Dies ist notwendig, um eine Verfälschung der Messergebnisse bei relevanten Geschwindigkeiten auszuschließen.

Jede Sekunde wird Testtupel in den Datenstrom eingebettet, das für die spätere Leistungsmessung genutzt wird. Es nutzt ein spezielles Symbol und hat eine eigene, aufsteigende Sequenznummer zur späteren Zuordnung. Der Versandzeitpunkt des Tupels wird in einer Logdatei protokolliert. Der BenchmarkDriver wird auf Harrenhal betrieben.

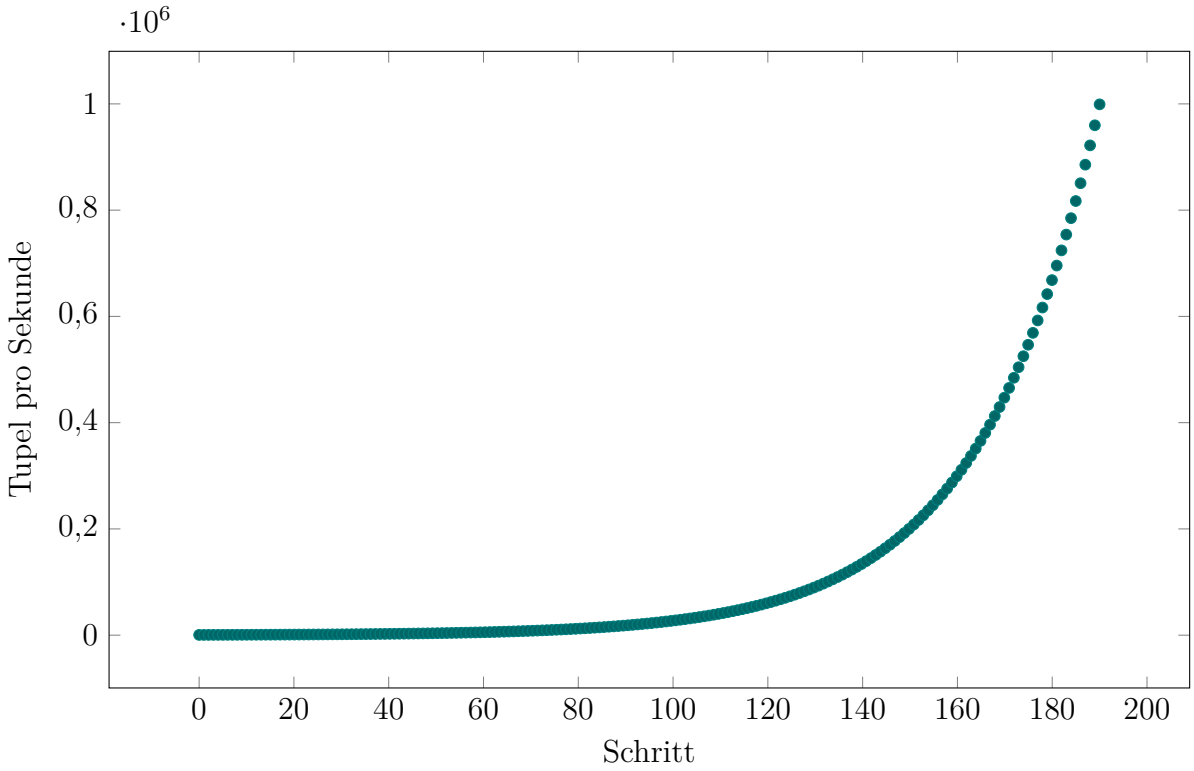


Abbildung 6.3: Geschwindigkeitsverlauf beim Abspielen der Tupel

Apache Kafka

Apache Kafka ist ein verteiltes Publish-Subscribe-System. Es wurde ursprünglich von LinkedIn [Lin15] entwickelt. Es ist auf einen hohen Durchsatz ausgelegt und speichert die Daten persistent auf der Festplatte. Ein zusammenhängender Datenstrom wird *Topic* genannt. Der Datenstrom wird zur Steigerung des Durchsatzes und optional zur Steigerung der Fehlertoleranz in mehrere Partitionen aufgeteilt. Sie können auf mehrere Kafka-Instanzen, genannt *Broker*, verteilt werden. *Producer* emittieren die Daten und schreiben sie in die Partitionen. Die Reihenfolge der Daten bleibt nur innerhalb einer Partition erhalten. Lesende Clients, namens *Consumer*, können mehrere Partitionen gleichzeitig lesen. Eine Partition kann von mehreren verschiedenen *Consumer Groups* gleichzeitig gelesen werden, aber jeweils nur ein Gruppenmitglied zur selben Zeit. Dadurch bleibt die Reihenfolge auch bei mehreren lesenden Clients erhalten. Die Architektur von Kafka wird in Abbildung 6.4 visualisiert. [Apa15x; KNR11]

Kafka wird in der Architektur dieser Arbeit zur Verteilung des Datenstroms und als Upstream Backup genutzt. Da die Daten dort persistiert werden, können sie im Fehlerfall wieder abgerufen werden. Der BenchmarkDriver fungiert als Producer, der die Datensätze

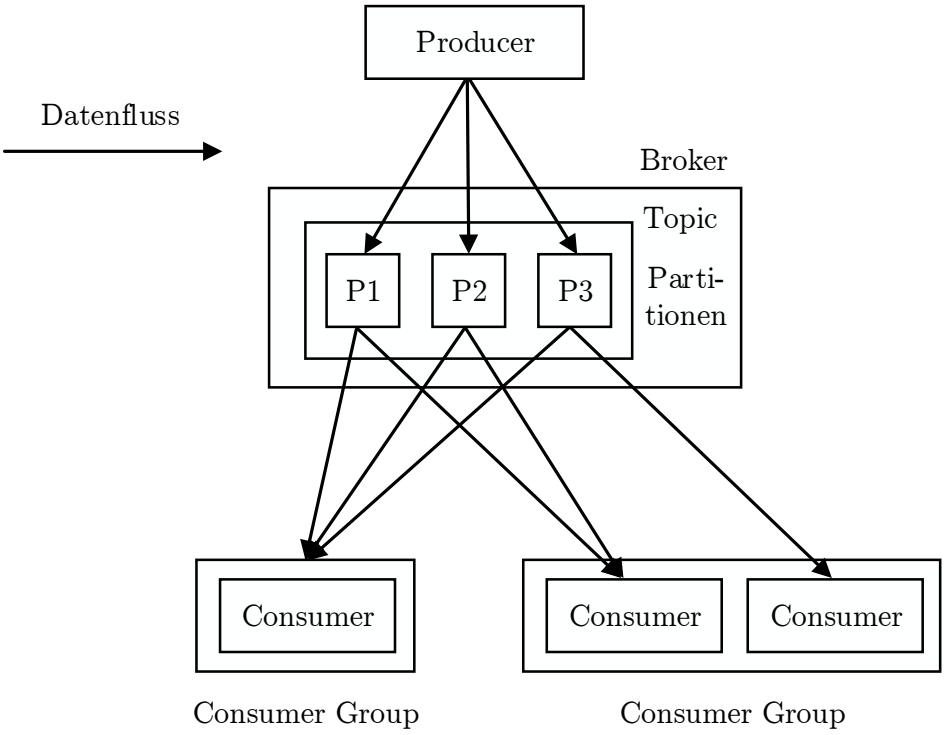


Abbildung 6.4: Architektur von Apache Kafka nach [Apa15x]

einspeist. Das Framework bildet eine Consumer Group. Es kann mehrere Consumer starten, die die einzelnen Partitionen abrufen. Bei der Implementierung werden 32 Partitionen verwendet, da Samza die Parallelität der Aufgabe anhand der Partitionen steuert und die 32 Partitionen gut auf die vier Hosts verteilt werden können. Die Broker sind auf den gleichen vier Hosts installiert, wie die Worker der Frameworks.

6.2.2 Zielsysteme

Die Zielsysteme dienen der Ablage der Daten und der Auswertung der Leistungsmessung. Das Framework legt die Ergebnisse in Redis ab. Zur Messung der Latenz und des Durchsatzes wird der *BenchmarkTimer* genutzt. Durch Messungen wurde sichergestellt, dass mindestens 80.000 Tupel pro Sekunde verarbeitet werden können. Da die eingehenden Daten aggregiert werden, ist diese Kapazität ausreichend für das Zielsystem.

Redis

Redis ist eine In-Memory NoSQL-Datenbank, die verschiedene Datenstrukturen zur Verfügung stellt. Im einfachsten Fall fungiert Redis als *Key-Value-Store*, es unterstützt aber auch Listen, Mengen, sortierte Mengen und weitere Datenstrukturen. Optional kann die Datenbank persistiert werden. Darüber hinaus stellt Redis einen Publish-Subscribe-Mechanismus zur Verfügung. Dieser kann unabhängig von der Datenbank genutzt werden. Redis nutzt ein einfaches Netzwerkprotokoll zur Kommunikation und bietet für viele verschiedene Programmiersprachen Bibliotheken. Da Redis nur einen Thread nutzt, wird zur Leistungssteigerung ein Cluster eingerichtet. Im Clustermodus können nur Befehle ausgeführt werden, die sich auf einen einzelnen Schlüssel beziehen. Jeder der Knoten im Cluster ist für einen bestimmten Schlüssel-Bereich zuständig. Clients müssen ihre Anfrage bereits an den richtigen Knoten richten, sonst werden sie an den zuständigen Knoten verwiesen. [San15a]

Redis wird wegen seinen vielseitigen Funktionen und der einfachen Anbindung als Zielsystem ausgewählt. Es wird in dieser Architektur primär genutzt, um die verarbeiteten Daten abzulegen. Je nach Framework werden auch Datenstrukturen von Redis zur Verarbeitung genutzt. Durch den Publish-Subscribe-Mechnanismus können Drittanwendungen über aktualisierte Daten informiert werden. Dies nutzt der BenchmarkTimer zur Messung der Leistung. Redis wird im Cluster Modus mit acht Master Instanzen auf Pinkmaiden betrieben. Um Redis aus der Anwendung heraus anzusprechen, wird die Bibliothek Jedis [Lei15] genutzt, da es von den Entwicklern von Redis empfohlen wird [San15b].

BenchmarkTimer

Der BenchmarkTimer ist das Gegenstück zum BenchmarkDriver. Er wurde im Rahmen dieser Arbeit implementiert. Der BenchmarkTimer abonniert die Änderungen, die in Redis geschrieben werden. Sobald Datensätze des Testtupels gespeichert werden, wird dies in einer Logdatei mit Zeitstempel und Sequenznummer protokolliert. Bei der Auswertung wird die Logdatei vom BenchmarkDriver und BenchmarkTimer verglichen und dadurch die Ende-zu-Ende-Latenz der Verarbeitung ermittelt. Auf Harrenhal wird ein Zeitserver betrieben, mit dem die Hosts synchronisiert werden. Die Abweichung der Zeit wird damit unter einer Millisekunde gehalten [Mil06, S. 10]. Der BenchmarkTimer ist in Java implementiert und wird auf Pinkmaiden ausgeführt.

6.2.3 Unterstützende Systeme

Neben den Systemen, die direkt der Datenverarbeitung dienen, werden weitere Systeme gebraucht, die die verarbeitenden Systeme dabei unterstützen. Sie werden im Folgenden kurz vorgestellt.

Apache ZooKeeper

Apache ZooKeper ist ein zentraler Dienst zur Koordination von Systemen. Dies schließt Aufgaben wie Konfigurationsmanagement, Namensverwaltung, Synchronisation und *Leader Election*, also die Wahl eines Koordinators, ein. ZooKeeper bietet dafür einen hierarchischen Namensraum, ähnlich wie ein Dateisystem, an. Ein ZooKeeper-Dienst besteht aus Gründen der Fehlertoleranz meist aus mehreren Servern. Einer davon ist der Anführer, über den alle Schreibanfragen abgewickelt werden. Jede Änderung wird erst mit allen Servern abgestimmt und dann bestätigt, daher können alle Server Leseanfragen beantworten. Alle Schreibanfragen sind geordnet und werden atomar ausgeführt. [Hun+10; Apa15bn]

ZooKeeper wird bei dieser Architektur für Kafka genutzt. Kafka hält alle Konfigurationsdaten in ZooKeeper, etwa Informationen über Topics, Partitionen, Broker und Consumer. Wie in Unterkapitel 5.2 dargestellt, nutzen auch die untersuchten Stream Processing Frameworks ZooKeeper für ihre Koordination. Bei der Implementierung wird nur ein ZooKeeper-Server genutzt, da dessen Fehlertoleranz nicht Gegenstand der Untersuchung ist. Der Server wird auf Harrenhal betrieben.

PostgreSQL

PostgreSQL [Pos15] ist ein freies, ANSI-SQL:2008-konformes [ISO08] Datenbankmanagementsystem. In der Datenbank sind die Depotdaten für die in Abschnitt 5.5.4 beschriebene Aufgabe abgelegt. Es sind 10 Depots mit je 50 Positionen angelegt. Zusätzlich existiert ein 11. Depot, in dem nur das Testtupel als Position hinterlegt ist. Es wird zur Überprüfung der Funktion genutzt. Auszüge aus den Daten werden in Tabelle 6.3 und Tabelle 6.4 dargestellt.

PostgreSQL ist auf Harrenhal installiert. Es wird zusammen mit der Abfragebibliothek sql2o [Aab15] und dem *Connection Pool* HikariCP [Woo15] verwendet.

id	name
0	Name 0
1	Name 1

Tabelle 6.3: Auszug aus Tabelle „depot“

id	depot_id	symbol	quantity
0	0	KOPN	464
1	0	DYAX	46
50	1	CVG	309

Tabelle 6.4: Auszug aus Tabelle „depot_position“

Apache Hadoop Distributed Filesystem

HDFS ist ein verteiltes Dateisystem. Es ist für die Speicherung von großen Datenmengen auf gewöhnlicher Hardware ausgelegt. Es trifft daher einige Maßnahmen zur Steigerung der Fehlertoleranz, wie die redundante Speicherung von Daten. Es hat zwei Hauptkomponenten: Auf dem *NameNode* sind die Metadaten hinterlegt, also wo welcher Teil einer Datei gespeichert ist. Auf dem *DataNode* liegen die Daten vor. Der Client fragt die Metadaten beim NameNode ab und arbeitet dann direkt auf den DataNodes. [Apa15s]

In dieser Architektur wird es zur Ablage von temporären Daten und zur Speicherung von Checkpoints genutzt. Der NameNode ist auf Harrenhal installiert. Die vier Worker-Hosts betreiben jeweils einen DataNode.

6.2.4 Aufbau der Anwendung

Die Anwendung wird anhand der vorgegebenen Aufgabenstellung aufgebaut. Ihre logische Struktur wurde bereits in Abbildung 6.2 dargestellt. Diese Struktur findet sich in jeder konkreten Implementierung mit einem Framework, daher wird zunächst die allgemeine Funktionsweise dargelegt.

Der Statistikknoten ist das zentrale Element, das die eingehenden Daten aggregiert. Entsprechend den Anforderungen wird jede Sekunde eine gleitende Statistik mit einer Dauer von fünf Minuten gebildet. Bei Frameworks, die einen Window-Mechanismus anbieten, kann diese Funktion einfach realisiert werden. Wegen der großen Datenmenge von beispielsweise zwei Gigabyte pro Minute bei einer Geschwindigkeit von 200.000 Datensätzen pro Sekunde ist eine Voraggregierung sinnvoll. Die Daten werden eine Sekunde lang gesammelt, aggregiert und dann im Fünf-Minuten-Window gesammelt. Die dort gesammelten Daten werden aggregiert und die Fünf-Minuten-Statistik wird weitergegeben. Ohne vom Framework bereitgestellten Window-Mechanismus, muss dieser

manuell implementiert werden. Die Speicherung der aggregierten Daten ist abhängig vom Framework, muss aber fehlertolerant erfolgen.

Der Alertknoten hat zwei Datenströme als Eingabe: die Kursdaten und die Statistikdaten. Das Verknüpfen beider Datenströme ist abhängig vom Framework. Es muss sichergestellt sein, dass zu jedem eingehenden Kursdatensatz die entsprechende Statistik vorliegt, um die Abweichung zu prüfen. Die Statistiken können im Knoten selbst oder extern vorgehalten werden. Die resultierende Meldung wird in Redis gespeichert und per Publish veröffentlicht. Wie bereits erwähnt, wird auf diesem Pfad die Latenz der Anwendung gemessen.

Die Aufgabe zur Bewertung von Depots ist in zwei Verarbeitungsknoten aufgeteilt. Der erste Verarbeitungsknoten nutzt die eingehenden Statistiken, um die entsprechenden Depotpositionen zu bewerten. Der zweite Verarbeitungsknoten aggregiert den Strom an Depotpositionen nach Depotnummer und speichert sie in Redis. Die Aufteilung in zwei Schritte wird vorgenommen, da so die Tupel besser auf die Verarbeitungsknoten verteilt werden. Die Kurse eines Wertpapiers werden am gleichen Knoten verarbeitet und die Summierung der Positionen für ein Depot findet am gleichen Knoten statt. Zur Beschleunigung des Zugriffs auf PostgreSQL wird Google Guava [Goo15b] als Cache genutzt.

6.3 Implementierung der Aufgaben

Im Folgenden werden die Aufgaben aus Unterkapitel 5.5 für jedes Framework abgearbeitet und die Vorgehensweise beschrieben. Da sich die Storm Trident API bei der Implementierung stark von der gewöhnlichen Storm API unterscheidet, wird sie in einem eigenen Abschnitt behandelt. Die Implementierung wird bei allen Frameworks mit Java 8 vorgenommen. Bei jeder Implementierung wird etwa der gleiche Aufwand aufgewendet, daher wurde auch keine Implementierung speziell auf hohe Leistungsfähigkeit optimiert.

Aus Gründen der Lesbarkeit und des besseren Verständnisses werden die Quelltext-Beispiele gegenüber dem Original modifiziert. Der vollständige Quelltext kann auf dem beiliegenden Datenträger eingesehen werden.

6.3.1 Apache Flink

Installation des Frameworks

Flink stellt eine sehr ausführliche Anleitung bereit [Apa15d]. Sie deckt auch die Voraussetzungen, wie die Einrichtung von ssh und HDFS ab, die bei dieser Arbeit bereits im Vorfeld vorgenommen wurde. Daher beginnt die Durchführung der Aufgabe beim Abschnitt „Flink Setup“ in der Anleitung. Dort wird ein Download-Link angeboten, dessen Ziel nicht vorhanden ist. Daher muss man manuell zur Download-Seite navigieren. Dort werden Binärpakete für die verschiedenen Hadoop Versionen angeboten. Das Paket wird heruntergeladen und entpackt. Im nächsten Abschnitt der Anleitung sind die notwendigen Schritte zur Konfiguration angegeben. Diese bestehen im Wesentlichen aus der Anpassung einer Konfigurationsdatei und einer Datei, die die Hostnamen aller Worker enthält. Die anzupassenden Einstellungen sind sowohl in der Anleitung als auch in der Konfigurationsdatei selbst erklärt. Notwendige Einstellungen sind etwa der Hostname des Masters, die Menge an Arbeitsspeicher und die gewünschte Parallelität. Wenn die Konfiguration angepasst ist, kann Flink mit einem Befehl alle notwendigen Komponenten starten.

In das Projekt in der Entwicklungsumgebung muss die auf der Download-Seite angegebene Maven-Abhängigkeit eingetragen werden. Um die resultierende Anwendung auf dem Cluster auszuführen, müssen alle Abhängigkeiten in ein *Java Archive (JAR)* gepackt werden. Die Dokumentation von Flink zeigt dazu zwei Methoden auf [Apa15c].

Gleitende Statistik

Flink unterstützt die Umsetzung der Aufgabe mit seiner umfangreichen Schnittstelle. Die Anforderungen können größtenteils mit einer Reihe von Funktionsaufrufen umgesetzt werden, die sich an das Prinzip eines *Fluent Interface* [Fow05] anlehnen. Quelltext 6.2 zeigt, wie die Aufgabe in Flink umgesetzt wird. Der erste Block gruppiert den Datenstrom nach dem Symbol, bildet ein Window mit einem Sekundenintervall und nutzt die *Fold*-Operation, um das Window zu einer Statistik zu aggregieren. Gruppierungen können bei Flink nach Wert eines Feldes oder Rückgabewert einer Methode erfolgen. Die Window-Operation arbeitet standardmäßig mit Zeit- oder Mengenangaben, lässt sich aber für andere Anforderungen erweitern [Apa15g]. Die Fold-Operation ist eine Verallgemeinerung der Reduce-Operation [Läm08, S. 4]. Reduce aggregiert zwei Werte des gleichen Typs zu einem Wert dieses Typs. Fold aggregiert alle Werte eines Typs mithilfe eines Initialwerts zu einem

```
DataStream<Tuple2<String, Statistics>> oneSecondStatistics =
quotes
  .groupBy("f0").window(Time.of(1, TimeUnit.SECONDS))
  .foldWindow(new Tuple2<String, Statistics>("symbol", new Statistics()),
    new StatisticFolder()).flatten();

DataStream<Tuple2<String, StatisticsSum>> fiveMinuteStatistics=
oneSecondStatistics
  .groupBy("f0").window(Time.of(5, TimeUnit.MINUTES))
  .every(Time.of(1, TimeUnit.SECONDS))
  .reduceWindow(new StatisticReducer()).flatten();
```

Quelltext 6.2: Gleitende Statistik mit Flink

```
public class StatisticFolder
  implements FoldFunction<StringDoubleTuple, Tuple2<String, Statistics>> {

  public Tuple2<String, Statistics> fold(
      Tuple2<String, Statistics> initial, StringDoubleTuple value) {
    initial.f0 = value.f0;
    initial.f1.addValue(value.f1);
    return initial;
  }
}
```

Quelltext 6.3: Fold-Funktion der gleitenden Statistik in Flink

Wert des Initialwert-Typs. Die konkrete Fold-Funktion ist in Quelltext 6.3 zu sehen. Sie ist in eine eigene Klasse ausgelagert, da der JDK-Compiler die notwendigen Typinformationen für Lambda-Funktionen mit generischen Typen bisher nicht berücksichtigt.

Der zweite Block in Quelltext 6.2 erstellt die gleitende Statistik, indem die voraggregierten Daten der letzten fünf Minuten jede Sekunde mithilfe der Reduce-Operation zusammengefasst werden. Die konkrete Reduce-Funktion wird in Quelltext 6.4 dargestellt.

Meldung bei großen Kursabweichungen

Flink erlaubt die Verknüpfung von mehreren verschiedenartigen Datenströmen mit Co-Operatoren. Quelltext 6.5 zeigt, wie der Kurs-Datenstrom und der Statistik-Datenstrom verbunden, nach Symbol gruppiert und dann in einen `FlatMap`-Operator geleitet werden.

```
public class StatisticReducer
  implements ReduceFunction<Tuple2<String, StatisticsSum>> {

  public Tuple2<String, StatisticsSum> reduce(
      Tuple2<String, StatisticsSum> value1, Tuple2<String, StatisticsSum> value2) {
    return new Tuple2<>(value1.f0, StatisticsSum.aggregate(value1.f1, value2.f1));
  }
}
```

Quelltext 6.4: Reduce-Funktion der gleitenden Statistik in Flink

```
quotes
  .map(quote ->
    new QuotePriceSequenceTuple(quote.getSymbol(), quote.getBidPrice(),
      quote.getSequenceNumber()))
  .connect(statisticsStream).groupBy("f0", "f0")
  .flatMap(new BidPriceDeviationAlertFunction());
```

Quelltext 6.5: Verknüpfung von Streams in Flink

In Quelltext 6.6 ist die Funktion zu sehen, die im Operator ausgeführt wird. Die Methode `flatMap1()` nimmt die Kurse entgegen und führt die Vergleichsmethode aus. Die Statistiken werden durch `flatMap2()` entgegen genommen und in einer Map gespeichert. Die Map wird durch die Implementierung des `CheckpointedAsynchronously`-Interface regelmäßig durch den Flink Checkpoint-Mechanismus gesichert.

Bewertung von Wertpapierdepots

Die Bewertung der in PostgreSQL hinterlegten Depots wird in zwei Schritten vorgenommen. Die Implementierung des Ablaufs ist in Quelltext 6.7 dargestellt. Zunächst werden zu jedem Symbol in der Statistik die betreffenden Depotpositionen aus der Datenbank abgefragt und der Wert berechnet. Diese Funktionalität ist in Quelltext 6.8 abgebildet. Sie ist mithilfe der `RichFlatMapFunction`-Klasse implementiert. Dadurch können beim Aktivieren und Deaktivieren des Operators Funktionen, wie beispielsweise das Verbinden zur Datenbank, ausgeführt werden. Nachdem die Depotpositionen ermittelt wurden, werden sie nach Depot-ID gruppiert und aufsummiert.

```
public class BidPriceDeviationAlertFunction
    extends RichCoFlatMapFunction<QuotePriceSequenceTuple,
      Tuple2<String, StatisticsSum>, Tuple2<String, String>>
    implements CheckpointedAsynchronously<HashMap<String, StatisticsSum>>
{
  // ...

  @Override
  public void flatMap1(QuotePriceSequenceTuple quote,
      Collector<Tuple2<String, String>> out) throws Exception {
    checkDeviation(quote, out);
  }

  @Override
  public void flatMap2(Tuple2<String, StatisticsSum> value,
      Collector<Tuple2<String, String>> out) throws Exception {
    statistics.put(value.f0, value.f1);
  }
}
```

Quelltext 6.6: Co-FlatMap-Funktion von Streams in Flink

```
DataStream<Tuple3<Long, String, Double>> depotPositions = fiveMinuteStatistics
    .flatMap(new DepotPositionQuery(dbSettings));
DataStream<Tuple2<String, Double>> depotSum = depotPositions
    .groupBy("f0").map(new DepotSum());
```

Quelltext 6.7: Bewertung von Wertpapierdepots in Flink

6.3.2 Apache Samza

Installation des Frameworks

Apache Samza führt seine Laufzeitumgebung vollständig auf Hadoop YARN aus, daher muss Samza nicht installiert werden [Apa15am]. Samza bietet ein Skript an, um schnell eine Umgebung aus YARN, Kafka und ZooKeeper aufzubauen. Dieses Skript wird nicht genutzt, da Kafka und ZooKeeper bereits vorbereitet sind. Stattdessen wird YARN laut Anleitung installiert [Apa15t]. Da HDFS schon installiert ist, beschränkt sich dies auf wenige Schritte.

```
public class DepotPositionQuery extends RichFlatMapFunction<
    Tuple2<String, StatisticsSum>, Tuple3<Long, String, Double>> {
  // ...
  @Override
  public void open(Configuration parameters) throws Exception {
    depot = new DepotPositionManager(depotSettings);
  }

  @Override
  public void flatMap(Tuple2<String, StatisticsSum> tuple,
      Collector<Tuple3<Long, String, Double>> out) {
    String symbol = tuple.f0;
    Double avg = tuple.f1.getMean();
    List<DepotPosition> positions = depot.getBySymbol(symbol);
    positions.forEach(position -> {
      out.collect(new Tuple3<>(position.depot_id,
        symbol, avg * position.quantity));
    });
  }
}
```

Quelltext 6.8: Nutzung von PostgreSQL zur Bewertung der Depot Positionen in Flink

Neben der Maven-Abhängigkeit muss die Maven-Konfiguration modifiziert werden, damit ein Archiv gebaut wird, das auf dem Cluster ausführbar ist. Der Vorgang dazu ist nicht explizit beschrieben, sondern es wird das Beispiel-Projekt „hello-samza“ referenziert [Apa15ac]. Das Maven-Plugin *maven-assembly-plugin* muss eingebunden werden. Anhand des Beispiels und der Maven-Dokumentation [Apa15y] wird ein eigener *Assembly Descriptor* erstellt, der beschreibt, welche Inhalte in das resultierende Archiv exportiert werden.

Gleitende Statistik

Samza orientiert sich am reinen Stream Processing Model. Ein Verarbeitungsknoten wird durch das Interface `StreamTask` abgebildet. Tasks können durch weitere Interfaces erweitert werden. Einzelne StreamTasks werden unabhängig voneinander entwickelt, bereitgestellt und mit einer umfangreichen Konfigurationsdatei angepasst. Das Interface besteht aus einer einzelnen `process()`-Methode, die bei eingehenden Daten aufgerufen wird. Die Daten werden nicht typisiert übertragen und müssen per Hand in den richtigen Typ umgewandelt werden. Quelltext 6.9 zeigt die `process()`-Methode der Anwendung.

```
public void process(IncomingMessageEnvelope input,
    MessageCollector collector,
    TaskCoordinator coordinator) {
  Quote quote = (Quote) input.getMessage();
  sendQuotePrice(collector, quote);
  Statistics statistics = getCurrentStatistic(quote);
  statistics.addValue(quote.getBidPrice() / 10000.0);
  currentStatistics.put(quote.getSymbol(), statistics);
}
```

Quelltext 6.9: `process()`-Methode des Statistik-Task in Samza

Sie empfängt den Datensatz und leitet den Geldkurs über Kafka weiter. Dies ist für die spätere Verknüpfung der Kurs- und Statistik-Daten notwendig, damit beide Partitionen vom selben Task bearbeitet werden. Anschließend wird die aktuelle Statistik aus RocksDB geladen, der aktuelle Kurs hinzugefügt und wieder in RocksDB gespeichert.

Für Window-Operationen wird das Interface `WindowableTask` angeboten. Die `window()`-Methode des Interface wird im konfigurierten Intervall aufgerufen. Die Implementierung der Anwendung ist in Quelltext 6.10 dargestellt. Es werden alle aktuellen Statistiken aus RocksDB geladen und verarbeitet. Bei jeder wird die Historie, die in einer separaten RocksDB-Instanz gespeichert ist, aktualisiert, versendet und in Redis gespeichert.

```
public void window(MessageCollector collector, TaskCoordinator coordinator) {
  KeyValueIterator<String, Statistics> i = currentStatistics.all();
  while (i.hasNext()) {
    Entry<String, Statistics> stock = i.next();
    ArrayDeque<StatisticsSum> history = updateHistory(stock);
    StatisticsSum summary = StatisticsSum.aggregate(history);
    sendStatistics(collector, stock, summary);
    redis.set(stock.getKey(), summary);
    currentStatistics.delete(stock.getKey());
  }
  i.close();
}
```

Quelltext 6.10: `window()`-Methode des Statistik-Task in Samza

Meldung bei großen Kursabweichungen

Zur Generierung einer Meldung bei großen Kursabweichungen muss eine Verbundoperation über die Statistik und den aktuellen Kurs gebildet werden. Bei Samza wird dies durch die Nutzung von Kafka-Partitionen sichergestellt. Beim Senden von Tupeln kann ein *Parition Key* mitgegeben werden. Anhand von ihm wird entschieden, in welche Partition das Tupel geschrieben wird. Da ein Task von jedem Stream die gleichen Partitionsnummern liest, werden die Tupel von verschiedenen Streams in einem Task zusammengeführt [Apa15ab]. In Quelltext 6.11 wird dargestellt, wie zwischen den beiden Datenströmen unterschieden

```
public void process(IncomingMessageEnvelope envelope,
    MessageCollector collector,
    TaskCoordinator coordinator) {
  String stream = envelope.getSystemStreamPartition().getStream();
  if ("statistics".equals(stream)) {
    statistics.put((String) envelope.getKey(),
      (StatisticsSum) envelope.getMessage());
  } else if ("quotePrice".equals(stream)) {
    checkDeviation(envelope);
  }
}
```

Quelltext 6.11: `process()`-Methode des Meldungs-Task in Samza

wird. Die Statistiken werden in eine Map eingefügt und die Kurse in einer Methode auf Abweichungen gegenüber der Statistik geprüft.

Bewertung von Wertpapierdepots

Samza bietet für Initialisierungsroutinen das Interface `InitableTask` an. Wie in Quelltext 6.12 zu sehen, ist in der `init()`-Methode der Zugriff auf die Task-Konfiguration möglich. Es wird wieder ein Partition Key genutzt, damit alle Positionen eines Depots in die gleiche Partition geschrieben werden. Da Samza keine generischen Tupel-Klassen bereitstellt, werden eigene Klassen zur Übertragung genutzt, in diesem Fall `DepotPositionMessage`. Samza leitet die Positionen an einen weiteren Task weiter, der sie getrennt nach Depot aufsummiert.

```
public class DepotPositionTask implements StreamTask, InitableTask {
  private DepotPositionManager depot;

  @Override
  public void init(Config config, TaskContext context) {
    // ...
    depot = new DepotPositionManager(dbSettings);
  }

  @Override
  public void process(IncomingMessageEnvelope envelope,
      MessageCollector collector, TaskCoordinator coordinator) {
    String symbol = (String) envelope.getKey();
    StatisticalSummaryValues stats = (StatisticsSum) envelope.getMessage();
    double average = stats.getMean();
    List<DepotPosition> positions = depot.getBySymbol(symbol);
    for (DepotPosition position : positions) {
      collector.send(new OutgoingMessageEnvelope(OUTPUT_STREAM, position.depot_id,
        position.depot_id,
        new DepotPositionMessage(position.symbol, average * position.quantity)));
    }
  }
}
```

Quelltext 6.12: Nutzung von PostgreSQL zur Bewertung der Depot Positionen in Samza

6.3.3 Apache Spark

Installation des Frameworks

Die Installation von Spark gestaltet sich ähnlich wie die von Flink. Nach der Wahl der gewünschten Hadoop-Version kann man das Binärpaket herunterladen. Es steht eine ausführliche Installationsanleitung zur Verfügung [Apa15bc]. Alle Worker werden in eine Datei eingetragen, damit sie automatisch gestartet werden. Nach Angabe des Master-Hostnamens in der Konfigurationsdatei kann der Cluster mit einem Befehl gestartet werden. Alle anderen Einstellungen sind optional und werden in der Dokumentation umfassend erklärt.

Im Entwicklungsprojekt der Anwendung wird die Maven-Abhängigkeit hinzugefügt. Die Dokumentation von Spark enthält eine Anleitung, wie die Anwendung vorzubereiten ist, damit sie auf dem Cluster ausgeführt werden kann [Apa15bg].

Gleitende Statistik

Spark arbeitet auf der gleichen Abstraktionsebene wie Flink. Durch die Arbeitsweise mit Micro-Batches wird der Datenstrom prinzipiell in diskrete Abschnitte unterteilt. Im ersten Block in Quelltext 6.13 werden die Kurse nach Symbol gruppiert und mit allen Kursen im aktuellen Batch eine Statistik erstellt. Die API ist eine Mischung aus Fluent Interface und Mehrzweck-Methoden, wie im Beispiel zu sehen. Gruppieren ist nur nach dem Key einer `PairDStream`-Instanz möglich. Im zweiten Block werden die Statistiken in einem Window gesammelt und jede Sekunde aggregiert.

```
JavaPairDStream<String, Statistics> batchStatistics =
quotes.groupByKey().mapValues(prices -> {
       Statistics statistics = new Statistics();
       for (Double price : prices)
         statistics.addValue(price);
       return statistics.getSummary();
     });

JavaPairDStream<String, StatisticsSum> fiveMinuteStatistics =
batchStatistics.reduceByKeyAndWindow(
       (stat1, stat2) -> StatisticsSum.aggregate(stat1, stat2),
       Minutes.apply(5), Seconds.apply(1));
```

Quelltext 6.13: Gleitende Statistik mit Spark

Meldung bei großen Kursabweichungen

Die Verknüpfung von Datenströmen ist, wie das Gruppieren, nur anhand des Keys möglich. Die Werte der beiden Streams werden in einem neuen `Tuple2`-Objekt zusammengefasst. In Quelltext 6.14 wird die Implementierung der Anwendung vorgestellt. Kurs- und Statistik-Datenstrom werden verknüpft, der neu entstandene Datenstrom wird auf Abweichungen untersucht und die Ergebnisse in Redis gespeichert.

Bewertung von Wertpapierdepots

Spark hat, abgesehen von Window-Operationen, nur eine eingeschränkte Zustandsunterstützung. Nur die Methode `updateStateByKey()` steht zum Speichern eines Zustands zur

```
public void showBidPriceDeviationAlert(JavaPairDStream<String, Quote> quotes,
      JavaPairDStream<String, IStatistics> statisticsStream) {
  quotes.join(statisticsStream).flatMapToPair(tuple -> {
      return checkDeviation(tuple);
    }).foreach(new StringRedisUpdater(redisPool, "spark.alert"));
}
```

Quelltext 6.14: Verknüpfung von Streams in Spark

Verfügung. Bei der Bewertung von Depots wird ein Zustand gebraucht, um den aktuellen Kurs jeder Position zu halten. Die Implementierung ist in Quelltext 6.15 dargestellt. Anhand des Statistik-Datenstroms werden die betroffenen Positionen mit der Funktion `fetchPositionsFromDB` aus der Datenbank geladen. Die Datenbankverbindung muss bei Spark mit einem *Connection Pool* [LM07] arbeiten, da es keine Methoden zum Initialisieren oder Finalisieren eines Operators gibt. Nachdem die Positionen geladen wurden, werden sie nach Depot-ID gruppiert und in Maps zusammengefasst. Im mittleren Block wird der Zustand aktualisiert. Für jedes Depot wird eine Map vorgehalten, in der die aktuellen Einträge hinterlegt sind. Abschließend werden im letzten Block die Einträge aufsummiert und in Redis abgelegt.

6.3.4 Apache Storm

Installation des Frameworks

Apache Storm stellt eine Anleitung bereit, in der die notwendigen Schritte der Installation zusammengefasst sind [Apa14l]. Dort sind die Softwarevoraussetzungen aufgelistet und der Installationsprozess beschrieben. Es wird ein Link zur Download-Seite angegeben, von der aus die Version ausgewählt werden kann. Das Binärpaket muss auf jedem Host entpackt werden, da die Logdateien in der Standardkonfiguration nicht den Hostnamen enthalten. Daher kann Storm nicht für alle Hosts gemeinsam in einem Netzwerkverzeichnis bereitgestellt werden. Neben der Angabe des Masters müssen der ZooKeeper-Server und ein lokales Verzeichnis für temporäre Daten angegeben werden. Außerdem wird die Anzahl der Worker-Prozesse durch Vergabe der TCP-Ports geregelt. Der Cluster wird nicht durch einen einzelnen Befehl gestartet, sondern die Komponenten müssen einzeln auf jedem Host gestartet werden. Die Komponenten nutzen das *fail-fast*-Prinzip, das heißt, bei einem schwerwiegenden Fehler wird die Komponente komplett beendet [Sho04]. Es wird daher

```
JavaPairDStream<Long, Tuple2<String, Double>> depotPositions = statisticsStream
    .flatMapToPair(fetchPositionsFromDB).groupByKey().mapValues(iterable -> {
      Map<String, Double> map = new HashMap<>();
      for (Tuple2<String, Double> value : iterable) {
        map.put(value._1, value._2);
      }
      return map;
    });

JavaPairDStream<Long, Map<String, Double>> depots = depotPositions
    .updateStateByKey((values, state) -> {
      Map<String, Double> map = state.or(new HashMap<>());
      for (Map<String, Double> value : values) {
        for (Map.Entry<String, Double> entry : value.entrySet()) {
          map.put(entry.getKey(), entry.getValue());
        }
      }
      return Optional.of(map);
    });
}

depots.mapToPair(tuple -> {
    Map<String, Double> map = tuple._2;
    double sum = 0;
    for (Double value : map.values()) {
      sum += value;
    }
    return new Tuple2<String, Double>(tuple._1.toString(), sum);
  }).foreachRDD(new DoubleRedisUpdater(redisPool, "spark.depot_sum"));
```

Quelltext 6.15: Bewertung von Wertpapierdepots in Spark

in der Anleitung empfohlen, ein Werkzeug zur Prozessüberwachung, wie etwa *supervisord* [Age15], einzusetzen.

Das Aufsetzen eines Projekts mit Storm ist in einem separaten Dokument beschrieben [Apa14e]. Die Maven-Abhängigkeit von Storm wird im Entwicklungsprojekt hinzugefügt und das Projekt entsprechend dem verknüpften Beispiel angepasst, sodass die nötigen Abhängigkeiten im JAR enthalten sind.

Gleitende Statistik

Verarbeitungsknoten werden bei Storm *Bolt* genannt. Bolts verarbeiten Tupel durch die `execute()`-Methode. Die Tupel sind in den meisten Fällen Teil des Datenstroms, sie können aber auch von Storm erzeugt werden, beispielsweise *Tick-Tuple*, um zeitliche Abläufe zu steuern. Diese Funktion wird zur Implementierung des Window-Mechanismus genutzt. Da Storm keine Mittel hat, um einen Zustand fehlertolerant zu speichern, wird Redis als externe Ablage genutzt. Quelltext 6.16 zeigt die `execute()`-Methode der Anwendung. Im Falle eines Datentupels wird der obere Block abgearbeitet. Die aktuelle Statistik wird aus einer Map geholt und aktualisiert. Das Tupel wird für die spätere Bestätigung beim Acker in einer Liste gespeichert. Es kann noch nicht bestätigt werden, da der Zustand noch nicht persistiert ist.

Der zweite Block wird im Falle eines Tick-Tupels ausgeführt. Für jede Statistik aus der internen Map wird die Historie aus Redis geladen, die Statistiken aggregiert und weitergeleitet. Anschließend wird die aktuelle Statistik in Redis gespeichert. Erst danach werden die bisher eingegangenen Tupel bestätigt.

Bei Storm sind die Tupel nicht typisiert. Werte in den Tupeln können über ihren Index oder Namen angesprochen werden. Jeder Bolt deklariert in der `declareOutputFields()`-Methode, welche Datenströme und Werte er emittiert. Beim Verknüpfen der Bolts werden die Namen angegeben und beim Starten auf Korrektheit geprüft.

Meldung bei großen Kursabweichungen

Streams in Storm werden verknüpft, indem sie in den gleichen Bolt geleitet werden [Apa14b]. Storm kann die Verbindungen zwischen Bolts auf verschiedene Arten partitionieren. Sie werden *Stream Groupings* genannt. Damit die Kurse und Statistiken vom selben Wertpapier bei der gleichen Instanz ankommen, wird das *Fields Grouping* verwendet. Es trifft die Entscheidung, welche Partition genutzt wird, auf Basis eines Feldes. Der erste Block in Quelltext 6.17 zeigt, wie die Topologie aufgebaut wird. Von beiden Datenströmen wird das Feld `symbol` zur Partitionierung herangezogen. Innerhalb des Bolts wird, ähnlich wie bei Samza, die Herkunft des Tupels geprüft und die entsprechende Aktion ausgeführt. Wenn es sich um eine Statistik handelt, wird sie in einer Map abgelegt. Ein Tupel aus dem Kurs-Datenstrom wird auf Abweichungen geprüft und gegebenenfalls eine Meldung ausgegeben.

```
public void execute(Tuple tuple) {
  if (!Util.isTickTuple(tuple)) {
    Quote quote = (Quote) tuple.getValue(1);
    Statistics statistic = getCurrentStatistic(quote);
    statistic.addValue(quote.getBidPrice() / 10000.0);
    ackList.add(tuple);
  } else {
    for (Map.Entry<String, Statistics> stock : currentStatistic.entrySet()) {
      List<StatisticsSum> history = redis.lrange(stock.getKey(), 0, size,
          StatisticsSum.class);
      StatisticsSum currenctStatistic = stock.getValue().getSummary();
      history.add(currenctStatistic);
      StatisticsSum summary = StatisticsSum.aggregate(history);
      collector.emit("statisticsStream", Arrays.asList(stock.getKey(), summary));
      redis.lpush(stock.getKey(), currenctStatistic);
      redis.ltrim(stock.getKey(), 0, size);
    }
    currentStatistic.clear();
    for (Tuple dataTuple : ackList)
      collector.ack(dataTuple);
    ackList.clear();
    collector.ack(tuple);
  }
}

public void declareOutputFields(OutputFieldsDeclarer declarer) {
  declarer.declareStream("statisticsStream", new Fields("symbol", "statistics"));
}
```

Quelltext 6.16: Gleitende Statistik mit Storm

Bewertung von Wertpapierdepots

Storm bietet Methoden zum Initialisieren und Finalisieren des Bolts. Dadurch können Ressourcen einfach verwaltet werden. Die Implementierung ist in Quelltext 6.18 dargestellt. Dort werden die entsprechenden Positionen aus der Datenbank geladen und zur Summation an den nächsten Bolt weitergegeben.

```
builder.setBolt("bidPriceDeviationAlertBolt", new BidPriceDeviationAlertBolt(),
    parallelism)
        .fieldsGrouping("statisticsBolt", "statisticsStream", new Fields("symbol"))
        .fieldsGrouping("quotes", new Fields("symbol"));

public class BidPriceDeviationAlertBolt extends BaseBasicBolt {
// ...
  @Override
  public void execute(Tuple input, BasicOutputCollector collector) {
    String stream = input.getSourceStreamId();
    if ("statisticsStream".equals(stream)) {
      statistics.put(input.getString(0), (StatisticsSum) input.getValue(1));
    } else if ("default".equals(stream)) {
      checkDeviation(input, collector);
    }
  }
}
```

Quelltext 6.17: Verknüpfung von Streams in Storm

6.3.5 Apache Storm Trident

Installation des Frameworks

Die Installation von Storm Trident entspricht der von Storm, sie wird in Abschnitt 6.3.4 erläutert.

Gleitende Statistik

Storm Trident bietet Hilfsmittel für Aggregationen oder die Zustandsverwaltung, sie sind allerdings nicht sehr umfangreich. Window-Operationen werden nicht angeboten, stattdessen wird in der Dokumentation vorgeschlagen, die eingehenden Daten in Zeitraster einzuordnen [Apa14f]. Der Zustand wird von Storm Trident über das `State`-Interface abstrahiert. Standardmäßig werden nur Implementierungen bereitgestellt, die wie ein Key-Value-Store arbeiten. Um eine effizientere Implementierung mit Redis-Datenstrukturen zu ermöglichen, wird eine eigene `State`-Implementierung verwendet. Quelltext 6.19 zeigt einen Teil der Implementierung. Die Aufrufe von Funktionen folgen dem Muster Eingangswerte - Funktion - Ausgangswerte. Im ersten Block werden die eingehenden Tupel mit einem Zeitstempel versehen, in das Zeitraster eingeordnet, nach Zeitraster gruppiert und

```
public class DepotPositionBolt extends BaseBasicBolt {
  private DepotPositionManager depot;
  // ...

  @Override
  public void prepare(Map stormConf, TopologyContext context) {
    depot = new DepotPositionManager(dbSettings);
  }

  @Override
  public void execute(Tuple tuple, BasicOutputCollector collector) {
    String symbol = tuple.getString(0);
    StatisticalSummary stats = (StatisticalSummary) tuple.getValue(1);
    List<DepotPosition> positions = depot.getBySymbol(symbol);
    for (DepotPosition position : positions) {
      collector.emit(Arrays.asList(position.depot_id,
          position.symbol, stats.getMean() * position.quantity));
    }
  }
}
```

Quelltext 6.18: Bewertung von Wertpapierdepots in Storm

aggregiert. Anschließend wird mit den Ergebnissen der Zustand `oneSecondStatistics` aktualisiert.

Der zweite Block ruft aus dem Zustand die Statistiken der letzten fünf Minuten ab und aggregiert diese. Danach wird das Resultat in das Zeitraster eingeteilt und im Zustand `fiveMinuteStatistics` gespeichert. Dieses Vorgehen erreicht zwar die gewünschte Funktionalität, ist aber sehr aufwendig.

Meldung bei großen Kursabweichungen

Um den aktuellen Kurs mit der Statistik zu vergleichen, muss die aktuelle Statistik aus dem Zustand abgefragt werden. Der erste Block in Quelltext 6.20 nutzt eine `stateQuery`, um die aktuellste Statistik abzufragen. Anschließend wird im zweiten Block der Vergleich durchgeführt und gegebenenfalls eine Meldung in Redis gespeichert.

```
TridentState oneSecondStatistics =
quoteStream
  .each(new Fields("symbol"), new TimestampQuote(), new Fields("timestamp"))
  .each(new Fields("timestamp"), new TimeBucket(1, "timestamp"),
    new Fields("timestampGroup"))
  .groupBy(new Fields("symbol", "timestampGroup"))
  .aggregate(new Fields("symbol", "bidPrice", "timestampGroup"),
    new BuildStatistics(), new Fields("statistic"))
  .partitionPersist(getStateFactory("oneSecondStatistics"), new Fields("statistic"),
    new StatisticsUpdater(), new Fields("symbol"));

TridentState fiveMinuteStatistics =
oneSecondStatistics
  .newValuesStream()
  .stateQuery(oneSecondStatistics, new Fields("symbol"),
    new QueryStatistics(5 * 60), new Fields("stats"))
  .each(new Fields("symbol", "stats"), new SumStatistics(),
    new Fields("statisticSum"))
  .each(new Fields("symbol"), new TimestampQuote(), new Fields("timestamp"))
  .each(new Fields("timestamp"), new TimeBucket(1, "timestamp"),
    new Fields("timeframe"))
  .partitionPersist(getStateFactory("fiveMinuteStatistics"),
    new Fields("symbol", "timeframe", "statisticSum"), new StatisticsSumUpdater(),
    new Fields("symbol", "statisticSum"));
```

Quelltext 6.19: Gleitende Statistik mit Storm Trident

Bewertung von Wertpapierdepots

Das Aussehen der Verarbeitungsfunktion von Storm Trident in Quelltext 6.21 entspricht fast der von Storm in Quelltext 6.18. Die Klassenhierarchie von Storm und Storm Trident ist aber trotz der semantischen und syntaktischen Gemeinsamkeiten komplett getrennt, daher sind beide Implementierungen inkompatibel zueinander.

```
Stream statisticsStream =
quoteStream
  .each(new Fields("symbol", "quote"), new ExtractBidPrice(),
    new Fields("bidPrice", "sequenceNr"));
  .stateQuery(fiveMinuteStatistics, new Fields("symbol"),
    new QueryNewestStatisticSum(), new Fields("statisticSum"))
  .each(new Fields("statisticSum"), new StatisticExtract(),
    new Fields("avg", "std"))

statisticsStream
  .each(new Fields("bidPrice", "avg", "std", "symbol", "sequenceNr"),
    new BidPriceDeviationAlertFunction(), new Fields("newSymbol", "alert"))
  .each(new Fields("newSymbol", "alert"),
    new RedisStringUpdater(redisPool, "storm_trident.alert"), new Fields());
```

Quelltext 6.20: Verknüpfung von Streams in Storm Trident

```
public class DepotPositionFunction extends BaseFunction {
  private DepotPositionManager depot;
  // ...

  @Override
  public void prepare(Map conf, TridentOperationContext context) {
    depot = new DepotPositionManager(dbSettings);
  }

  @Override
  public void execute(TridentTuple tuple, TridentCollector collector) {
    String symbol = tuple.getString(0);
    Double average = tuple.getDouble(1);
    List<DepotPosition> positions = depot.getBySymbol(symbol);
    for (DepotPosition position : positions) {
      collector.emit(Arrays.asList(position.depot_id, average * position.quantity));
    }
  }
}
```

Quelltext 6.21: Bewertung von Wertpapierdepots in Storm Trident

7 Ergebnisse

Im Folgenden werden die Kriterien aus Unterkapitel 5.4 auf die in Unterkapitel 5.2 untersuchten Frameworks angewendet. Dabei werden die im Abschnitt 5.3.1 beschriebenen Mittel zur Informationsgewinnung genutzt. Dieses Kapitel ist analog zum Unterkapitel 5.4 aufgebaut.

Einige Ergebnisse werden in Tabellen zusammengefasst. Die Bedeutung der verwendeten Symbole ist in Tabelle 7.1 erläutert. Eine genauere Beschreibung der Ergebnisse ist im entsprechenden Unterabschnitt enthalten.

-	Nicht vorhanden beziehungsweise keine Unterstützung
(✓)	Teilweise vorhanden beziehungsweise eingeschränkte Unterstützung
✓	Vollständig vorhanden beziehungsweise komplette Unterstützung

Tabelle 7.1: Bedeutung der Symbole in den Ergebnistabellen

7.1 Functional Suitability

7.1.1 Unterstützte Programmiersprachen

Apache Flink

Apache Flink stellt für Java und Scala eine API zur Verfügung. Java wird ab Version 6 unterstützt. Es wird aber auch eine Schnittstelle für Java 8 bereitgestellt, um die neu eingeführten Lambda-Funktionen zu nutzen. Wie bereits in Abschnitt 6.3.1 erwähnt, generieren noch nicht alle Java 8 Compiler ausreichende Typinformationen, daher können Lambda-Funktionen in Flink nur eingeschränkt genutzt werden [Apa15j]. Für Scala werden nur Binärpakete für die Sprachversion 2.10 bereitgestellt. Da Scala-Hauptversionen

zueinander inkompatibel sind, müssen Pakete für die aktuelle Scala-Version 2.11 selbst kompiliert werden [Ale15].

Das Mischen von Java und Scala innerhalb eines Projekts wird in der Dokumentation von Flink nicht erwähnt. Da Java und Scala, wie innerhalb von Flink, grundsätzlich gemischt werden können, ist auch hier davon auszugehen, dass es möglich ist.

Apache Samza

Apache Samza ist zwar mit einer Mischung aus Java und Scala implementiert, stellt aber nur eine Java-API zur Verfügung. Java wird ab Version 6 unterstützt, ab der nächsten Version ist Java 7 nötig [Hom15]. Da offiziell keine anderen Sprachen unterstützt werden, kann innerhalb eines Projekts nur Java verwendet werden.

Apache Spark

Apache Spark bietet Schnittstellen in den Sprachen Java, Scala, Python, R [R F15] und SQL an. R kann nur im Batch-Modus benutzt werden. SQL wird über die *DataFrame API* bereitgestellt. Sie kann nur auf eine diskrete Datenmenge, etwa einen Mini-Batch oder ein Window angewendet werden. Java wird ab Version 6 unterstützt. Für Scala wird ein Binärpaket für Version 2.10 bereitgestellt, Version 2.11 muss manuell kompiliert werden [Apa15aw]. Die Python-Implementierung ist noch unvollständig, insbesondere Datenquellen wie Kafka werden noch nicht unterstützt. [Apa15be]

Die Ausführungsumgebung wird in der Schnittstelle durch den SparkContext repräsentiert. Da verschiedene Anwendungen den SparkContext nicht teilen können, lassen sich Java und Python nicht mischen [Apa15ax]. Auch wenn in der offiziellen Dokumentation das Mischen von Java und Scala nicht erwähnt wird, ist dies möglich [Ole15].

Apache Storm

Apache Storm stellt native APIs in Java und Clojure zur Verfügung. Java wird ab Version 6 unterstützt. Zusätzlich bietet Storm das *Multi-Lang Protocol* [Apa14j]. Das Protokoll basiert auf *JavaScript Object Notation (JSON)* und erlaubt es, beliebige Sprachen anzubinden. Storm unterstützt offiziell JavaScript, Python und Ruby. In Java wird nur ein `ShellBolt` erstellt, der über die Standard-Datenströme mit dem externen Programm,

das in der entsprechenden Sprache implementiert ist, kommuniziert. Außerdem gibt es inoffizielle Projekte, die APIs in Scala und JRuby bereitstellen [Apa14m]. Die Sprachen können innerhalb eines Projekts beliebig gemischt werden.

In Storm Trident kann nur die Java-API genutzt werden [Xu13a].

Zusammenfassung

Sprache	Flink	Samza	Spark	Storm	Storm Trident
Java	✓	✓	✓	✓	✓
Scala	✓	-	✓	(✓)	-
Python	-	-	(✓)	✓	-
Clojure	-	-	-	✓	-
Beliebig	-	-	-	✓	-
Mischbar	(✓)	-	✓	✓	-

Tabelle 7.2: Unterstützte Programmiersprachen

7.1.2 Mächtigkeit der angebotenen API

Apache Flink

Flink bietet zahlreiche Operationen für Datenströme an. Map, Reduce und Fold sind die typischen Big-Data-Operationen. Durch *FlatMap* können aus einem Eingabetupel kein oder mehrere Ausgabetupel erzeugt werden. *Filter* sortiert Tupel aus dem Datenstrom aus und *Project* selektiert einzelne Felder aus Tupeln. Split kann einen Datenstrom nach einem bestimmten Kriterium auf mehrere Datenströme aufteilen. Datenströme mit dem gleichen Typ können über den *Union*-Operator in einen Strom zusammengefasst werden. Alle Funktionen haben Methoden, um sie zu initialisieren und zu finalisieren. Flink liefert auch vordefinierte Aggregationen, wie Min, Max oder Count, mit. [Apa15g]

Datenströme lassen sich nach einem oder mehreren Feldern beziehungsweise einem benutzerdefinierten Kriterium gruppieren. Auf gruppierte Datenströme können alle oben genannten Operationen angewendet werden. [Apa15g]

Window-Operatoren lassen sich auf der Basis von Zeit, Anzahl, einem Wertunterschied oder einer benutzerdefinierten Funktion erstellen. Für die Länge des Window und den Zeitpunkt, wann das Window Tupel emittiert, können unterschiedliche Kriterien genutzt werden. Auf Windows können Map-, Reduce- und Fold-Operationen angewendet werden. Der Window-Operator kann einzelne Tupel oder das ganze Window auf einmal emittieren. Window-Operationen und gruppierte Datenströme lassen sich kombinieren. Über ein zeitbasiertes Window können verschiedene Datenströme durch die Verbund-Operation oder das Kreuzprodukt verknüpft werden. [Apa15g]

Mehrere verschiedene Datenströme können über Co-Operatoren in einer Funktion zusammengeführt und mit einer benutzerdefinierten Logik verbunden werden. Iterative Prozesse, also Zyklen im Graph, werden ebenfalls unterstützt. [Apa15g]

Apache Samza

Samza bietet als einzige Abstraktion zur Stream-Verarbeitung Tasks an. Es lassen sich Methoden zum Initialisieren und Finalisieren von Ressourcen hinzufügen. Außerdem werden Window-Operationen durch das periodische Aufrufen einer speziellen Methode unterstützt. Durch einen Task lassen sich mehrere verschiedene Datenströme ausgeben. [Apa15at; Apa15ar]

Die Verknüpfung von mehreren Datenströmen wird gewährleistet, indem mehrere Kafka-Datenströme in einen Task geleitet werden. Durch den Partition Key lässt sich steuern, auf welcher Partition ein Tupel abgelegt wird. Entweder direkt durch die Angabe einer Partitionsnummer oder indirekt durch die Angabe eines Objekts, dessen *HashCode* [Ora15d] gebildet wird. [Apa15ar]

Apache Spark

Spark bietet zahlreiche Operationen für Datenströme an. Es werden Map, Reduce, FlatMap, Union und Filter unterstützt. *Transform* ist der allgemeinste Operator, der Operationen auf den zugrundeliegenden RDDs erlaubt. So können Batch- und Streaming-Modus verbunden werden, indem auf RDDs aus dem Batch-Bereich zugegriffen wird. Als vordefinierte Aggregation wird nur Count angeboten. Initialisierungs- oder Finalisierungsmethoden sind nicht vorgesehen. [Apa15be]

Die API unterscheidet strikt zwischen gewöhnlichen Datenströmen und solchen, die nach Schlüssel und Wert aufgeteilt sind. Nur letztere können nach dem Schlüssel gruppiert werden. Der gruppierte Datenstrom kann mit der Reduce-Operation weiterverarbeitet werden oder ein Zustand mit seiner Hilfe aktualisiert werden. Außerdem lassen sich die oben genannten Operationen anwenden. Dabei ist zu beachten, dass sich die Argumente im Vergleich zu einem nicht-gruppierten Datenstrom unterscheiden. [Apa15be]

Window-Operationen können zeitbasiert genutzt werden. Sowohl die Länge, als auch der Zeitpunkt, wann das Window emittiert wird, lassen sich einstellen. Auf ein Window lassen sich die oben genannten Operationen anwenden. Verschiedene Datenströme lassen sich anhand des Schlüssels durch die Verbund-Operation verknüpfen. [Apa15be]

Apache Storm

Storm arbeitet ebenso wie Samza nur mit Verarbeitungsknoten. Sie stellen Methoden zur Initialisierung und Finalisierung bereit. Durch vom System generierte Tick-Tupel können Window-Operationen durchgeführt werden. Ein Bolt kann mehrere verschiedene Datenströme ausgeben. [Apa14d]

Es werden verschiedene Verbindungen der Bolts (Groupings) unterstützt. Durch *Shuffle* werden die Tupel für eine bessere Auslastung zufällig verteilt. Optional kann *Local or Shuffle* verwendet werden, damit weniger Tupel über das Netzwerk verschickt werden müssen. Das *None*-Grouping ist aktuell identisch zum Shuffle-Grouping. Fields- und *Partial-Key*-Grouping verteilen die Daten anhand der angegebenen Felder auf eine beziehungsweise zwei Instanzen eines Bolts. Damit können Verbund-Operationen realisiert werden, indem zwei Datenströme anhand des selben Feldes in eine Instanz geleitet werden. Durch *All* können die Tupel an alle Instanzen eines Bolts weitergeleitet werden. Das Gegenteil ist das *Global*-Grouping, bei dem alle Tupel an eine Instanz weitergeleitet werden. Das *Direct*-Grouping lässt den produzierenden Bolt entscheiden, an wen das Tupel weitergeleitet wird. [Apa14d]

Apache Storm Trident

Storm Trident bietet Funktionen, Filter und Projektionen als grundlegende Operationen auf Datenströmen an. Funktionen sind äquivalent zur FlatMap-Operation. Sie unterstützen Methoden zum Initialisieren und Finalisieren von Ressourcen. Aggregationen werden

mit der *Combiner*-, *Reducer*- oder *Aggregator*-Operation durchgeführt. Combiner sind bei anderen Frameworks als Reduce-Funktionen bekannt. Reducer werden bei anderen Frameworks Fold-Funktion genannt. Aggregator sind die allgemeine Form von Aggregationen, mit ihnen lassen sich benutzerspezifische Aggregationen umsetzen. Zwischen den Operatoren können die gleichen Verbindungen, wie bei Storm genutzt werden. [Apa14o]

Datenströme lassen sich nach mehreren Feldern gruppieren. Auf gruppierte Datenströme lassen sich Aggregationen und Funktionen anwenden. [Apa14o]

Window-Operationen sind nicht implementiert. Mehrere gleiche Datenströme können per *Merge* in einem Datenstrom zusammengeführt werden. Verschiedene Datenströme können im gleichen Batch-Intervall per Verbund-Operator verknüpft werden. [Apa14o]

Zusammenfassung

	Flink	Spark	Storm Trident
Map/FlatMap	✓	✓	✓
Reduce	✓	✓	✓
Filter	✓	✓	✓
Fold	✓	-	✓
Union	✓	✓	✓
Project	✓	-	✓
Initialisierung und Finalisierung von Operatoren	✓	-	✓
Weitere	Split, Co-Operatoren, Iterativ	-	Verbindungstypen
Gruppierung - ein Feld	✓	✓	✓
Gruppierung - mehrere Felder	✓	-	✓
Window-Operator	✓	✓	-
Window Kriterium	Zeit, Anzahl, Abweichung, Benutzerdefiniert	Zeit	-
Zugriff auf komplettes Window/Batch	✓	✓	-
Verbund-Operator	✓	✓	✓

Tabelle 7.3: Mächtigkeit der angebotenen API mit hoher Abstraktion

	Samza	Storm
Emittierung mehrerer Datenströme	✓	✓
Window-Hilfsmittel	✓	✓
Initialisierung und Finalisierung von Operatoren	✓	✓
Initialisierung und Finalisierung von Operatoren	✓	✓
Verbund über Feld	✓	✓
Direkte Wahl der Partition	✓	✓
Weitere Verbindungstypen	-	Shuffle, All, Global

Tabelle 7.4: Mächtigkeit der angebotenen API mit niedriger Abstraktion

7.1.3 Unterstützung von zustandsbehafteten Operationen

Apache Flink

Bei Flink kann jede Funktion einen Zustand durch Objektattribute halten. Damit der Zustand im Fehlerfall erhalten bleibt, wird das `Checkpointed`-Interface bereitgestellt. Es ist in Abbildung 7.1 dargestellt. Der in Abschnitt 7.5.2 dargestellte Fehlertoleranzmechanismus von Flink ruft beim Erstellen eines Checkpoints die Methode `snapshotState()` auf. Der Rückgabewert der Methode besteht aus einem Objekt, das den Zustand der Funktion abbildet. Dieses Objekt wird von Flink in den Checkpoint einbezogen. Im Fehlerfall wird die `restoreState()`-Methode mit dem zuvor gespeicherten Objekt aufgerufen. Dadurch kann die Funktion den internen Zustand wiederherstellen. [Apa15g]

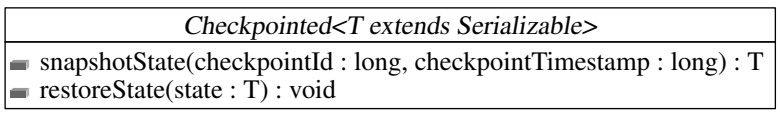


Abbildung 7.1: `Checkpointed`-Interface von Flink

Während der Erstellung des Checkpoints wird der Operator angehalten, um einen konsistenten Zustand sicherzustellen. Das `CheckpointedAsynchronously`-Interface erfordert keine Unterbrechung, dafür muss die Funktion dafür sorgen, dass der Rückgabewert von `snapshotState()` nicht geändert wird, etwa durch eine Kopie des Zustands. [Apa15g]

Apache Samza

Samza stellt für zustandsbehaftete Operationen jedem Task einen lokalen Speicher bereit. Um Fehlertoleranz zu gewährleisten, wird jede Änderung in einem Änderungsprotokoll verzeichnet. Es wird von Samza auf Kafka abgelegt und im Fehlerfall an den Speicher zur Wiederherstellung weitergeleitet. Der Speicher implementiert dafür das `StorageEngine`-Interface, das in Abbildung 7.2 dargestellt ist. Es ist auf die minimalen Methoden für die Wiederherstellung beschränkt und überlässt beispielsweise die Abfrage-Methoden der konkreten Implementierung. [Apa15ab]

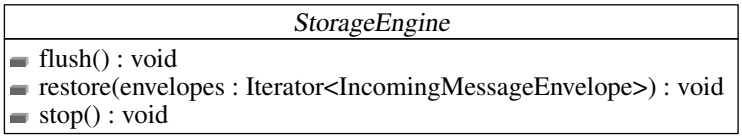


Abbildung 7.2: `StorageEngine`-Interface von Samza

Zu Samza gehört die Implementierung eines Key-Value-Stores auf der Basis von RocksDB. In RocksDB werden die Daten persistent auf dem Host gespeichert und im Fehlerfall aus Kafka wiederhergestellt. Wenn ein Task einen lokalen Speicher benötigt, wird dieser in der Konfigurationsdatei definiert und im Task angefordert. Wie in Abbildung 7.3 dargestellt, werden die typischen Methoden von Key-Value-Stores [Cat11, S. 15] angeboten. [Apa15ab]

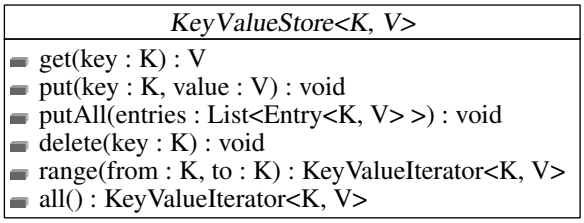


Abbildung 7.3: `KeyValueStore`-Interface von Samza

Apache Spark

Zum Speichern eines Zustands wird die `updateStateByKey()`-Operation bereitgestellt. Für jeden Key kann ein beliebiger Wert als fehlertoleranter Zustand gespeichert werden. Die Funktion zum Aktualisieren des Zustands wird in Abbildung 7.4 dargestellt. Die Parameter der `call()`-Methode sind die neuen Werte und der aktuelle Zustand. Daraus

bildet die Methode einen neuen Zustand als Rückgabewert. Der Zustand wird durch die in Abschnitt 7.5.2 beschriebenen Fehlertoleranzmechanismen geschützt. [Apa15be]

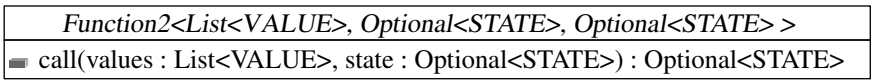


Abbildung 7.4: Funktion zum Aktualisieren des Zustands in Spark

Im Batch-Betrieb werden auch Akkumulatoren verwendet, diese sind allerdings nicht fehlertolerant, da bei einem Ausfall die Exactly-Once-Semantik nicht eingehalten wird. [Ras15]

Apache Storm

Storm bietet keine Unterstützung von zustandsbehafteten Operationen an. Innerhalb eines Bolts können zwar beliebige Daten im Speicher gehalten werden, deren Fehlertoleranz muss aber manuell, etwa mithilfe von externen Systemen, sichergestellt werden.

Apache Storm Trident

Storm Trident bindet für zustandsbehaftete Operationen externe Systeme ein. Dafür wird das in Abbildung 7.5 dargestellte Interface `State` genutzt. Die vorgegebenen Methoden dienen nur der Fehlertoleranz. Die Implementierung der Abfragen ist abhängig vom konkreten Speichersystem. Der Zustand wird mithilfe der `partitionPersist()`-Operation verändert und mithilfe der `stateQuery()`-Operation gelesen. Die Methoden benötigen für die konkrete Operation einen `StateUpdater` beziehungsweise eine `QueryFunction`. Sie werden in Abbildung 7.6 und Abbildung 7.7 dargestellt. Wie in Abschnitt 7.5.2 ausgeführt wird, kennt Storm Trident verschiedene Fehlertoleranz-Modi. Die Implementierung dieser drei Interfaces muss auf den genutzten Fehlertoleranzmodus abgestimmt sein. [Apa14q]

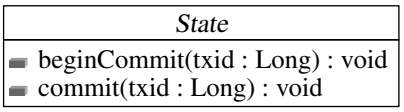


Abbildung 7.5: `State`-Interface von Storm Trident

Um die Implementierung zu vereinfachen gibt es das `MapState`-Interface mit zugehörigen Klassen zum Lesen und Schreiben von Werten. Damit wird eine Art Key-Value-Store

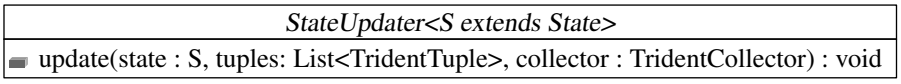


Abbildung 7.6: `StateUpdater`-Interface von Storm Trident

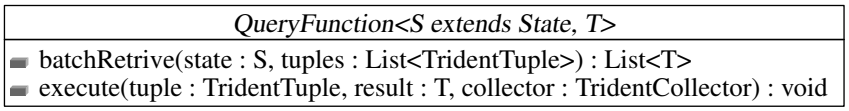


Abbildung 7.7: `QueryFunction`-Interface von Storm Trident

abgebildet. Die unterstützten externen Systeme sind in Abschnitt 7.3.2 in Tabelle 7.9 aufgelistet. [Apa14q]

Zusammenfassung

	Flink	Samza	Spark	Storm	Storm Trident
Methode	Checkpointing von Funktionen	Externes System mit Änderungs-protokoll	Key-Value-Operator	-	Externes System mit Transaktio-nen
Standardim-plementier-ung	Ablage auf HDFS	Key-Value-Store (RocksDB)	Ablage auf HDFS	-	Map-Interface (diverse Systeme)

Tabelle 7.5: Unterstützung von zustandsbehafteten Operationen

7.2 Performance Efficiency

In den folgenden Abschnitten werden die Ergebnisse der Latenzmessungen in Diagrammen dargestellt. Die Abszisse stellt die Geschwindigkeit der eingehenden Daten in Tupel pro Sekunden dar. Auf der Ordinate ist die mittlere Latenz der Verarbeitung in Millisekunden aufgetragen. Verschiedene Diagramme nutzen gegebenenfalls unterschiedliche Skalen, um die Darstellung zu verbessern. Die Ordinate ist logarithmisch skaliert. Alle Messprotokolle sind auf dem beiliegenden Datenträger enthalten.

7.2.1 Mittlere Verarbeitungszeit von Tupeln

Die mittlere Verarbeitungszeit von Tupeln aller Frameworks ist in Abbildung 7.8 abgebildet. Einige der zugrundeliegenden Werte sind in Tabelle 7.6 dargestellt. Die Verarbeitungszeiten werden mit vier Rechenknoten gemessen. Die Nutzung von einer Sekunde als Batch-Intervall bei Apache Spark ist nicht möglich, da bereits bei 500 Tupeln die Verarbeitungszeit bei etwa einer Sekunde liegt. Daher wird die Messung mit einem Batch-Intervall von fünf Sekunden durchgeführt.

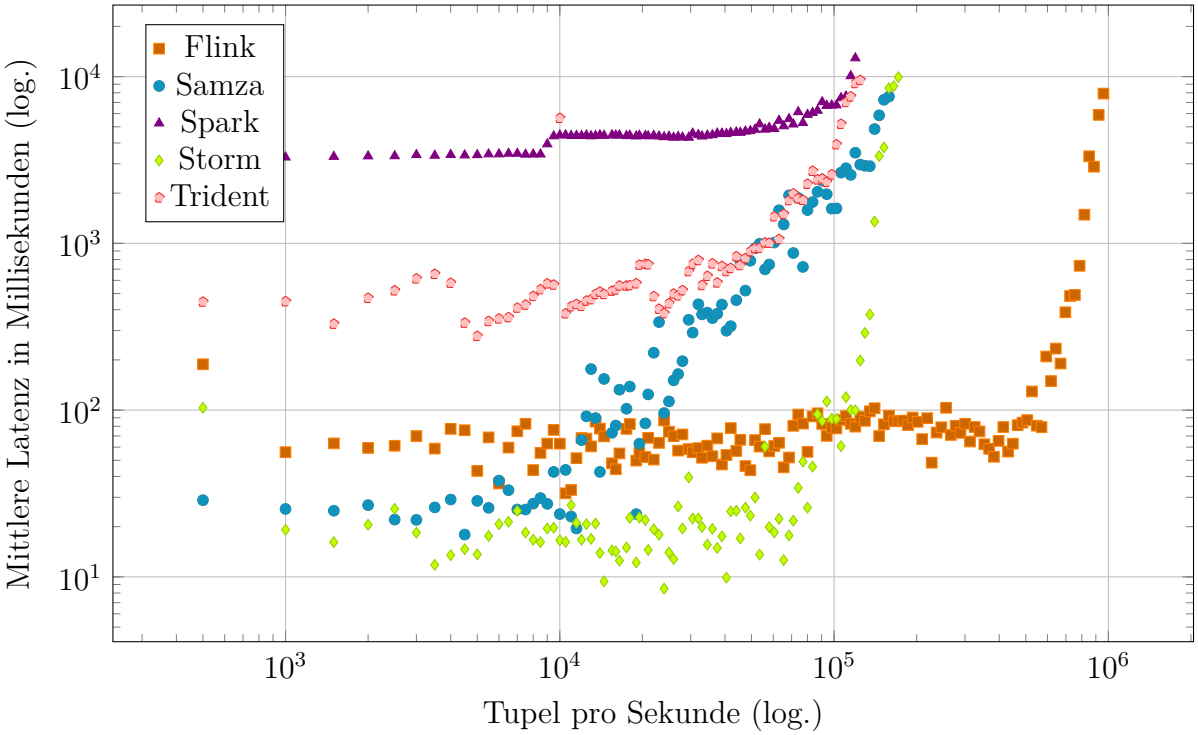


Abbildung 7.8: Mittlere Latenzen aller Frameworks im Vergleich

Der erste Messwert der Latenz bei 500 Tupeln pro Sekunde ist durch die Initialisierung der Operatoren und der externen Verbindungen zu Ressourcen erhöht. Flink und Storm zeigen über einen weiten Geschwindigkeitsbereich eine Latenz unter 100 Millisekunden. Erst bei einem höherem Durchsatz steigt die Latenz auf eine Sekunde und darüber. Bei Flink beginnt der Anstieg bei etwa 600.000 Tupeln pro Sekunde, bei Storm schon deutlich früher bei etwa 120.000 Tupeln pro Sekunde. Samza kann nur bis etwa 13.000 Tupeln pro Sekunde eine Latenz von unter 100 Millisekunden halten. Danach steigt die Latenz überproportional. Spark und Storm Trident haben, bedingt durch ihre Batch-orientierte Arbeitsweise, grundsätzlich eine höhere Latenz. Sie bewegt sich bei Storm Trident bis

etwa 60.000 Tupel pro Sekunde im Bereich zwischen 400 und 1000 Millisekunden. Danach steigt sie stark an. Das lange Batch-Intervall führt zu einer minimalen Latenz von über drei Sekunden, die sich aus dem Batch-Intervall von durchschnittlich 2,5 Sekunden und der Verarbeitungszeit ergibt. Ab 7,5 Sekunden Latenz reicht das Batch-Intervall nicht mehr zum Verarbeiten der Tupel aus und der folgende Batch muss in eine Warteschlange eingereiht werden. Dies wird ab 102.000 Tupeln pro Sekunde erreicht. Daher steigt ab diesem Zeitpunkt die Latenz stark an.

Tupel pro Sekunde	Flink	Samza	Spark	Storm	Storm Trident
500	188	28	3.170	103	445
1000	56	26	3.281	19	448
10500	32	44	4.458	16	379
25000	74	113	4.373	14	435
49500	44	787	4.722	23	894
77000	83	722	5.281	49	1.814
102000	78	1.621	6.745	89	3.906
119500	80	3.495	12.908	100	9.103
152000	82	7.261	-	3.762	-
201500	85	-	-	-	-
506000	87	-	-	-	-
817500	1.486	-	-	-	-
959500	7.912	-	-	-	-

Tabelle 7.6: Mittlere Latenzen in Millisekunden aller Frameworks im Vergleich

7.2.2 Skalierung auf mehrere Worker

Apache Flink

Der Ablauf der Anwendung wird durch die API in einen *Operator DAG* übersetzt. Durch einen Optimierer wird der Graph in einen *JobGraph* umgewandelt. Beim Ausführen einer Anwendung wird der JobGraph an den JobManager übergeben. Der JobManager erstellt daraus den *ExecutionGraph*, der in *Pipelines* organisiert ist. Entsprechend der gewünschten

Parallelität wird eine Menge an Pipelines erstellt. Sie werden in *TaskSlots* ausgeführt, die durch die TaskManager bereitgestellt werden. Jedem TaskSlot kann genau eine Pipeline zugeordnet werden. Die resultierenden Pipelines werden an die TaskManager verteilt und ausgeführt. Da die Aufgabenstellung eher Ein-/Ausgabe-intensiv ist, werden vier Taskslots pro CPU verwendet. [Apa15h; Apa15k]

Abbildung 7.9 stellt die Messergebnisse dar. Die Latenz bleibt über einen weiten Geschwindigkeitsbereich auf einem niedrigen Niveau. Bei einer Sekunde Latenz können etwa 110.000, 340.000 beziehungsweise 785.000 Tupel pro Sekunde mit einem, zwei beziehungsweise vier Knoten erreicht werden.

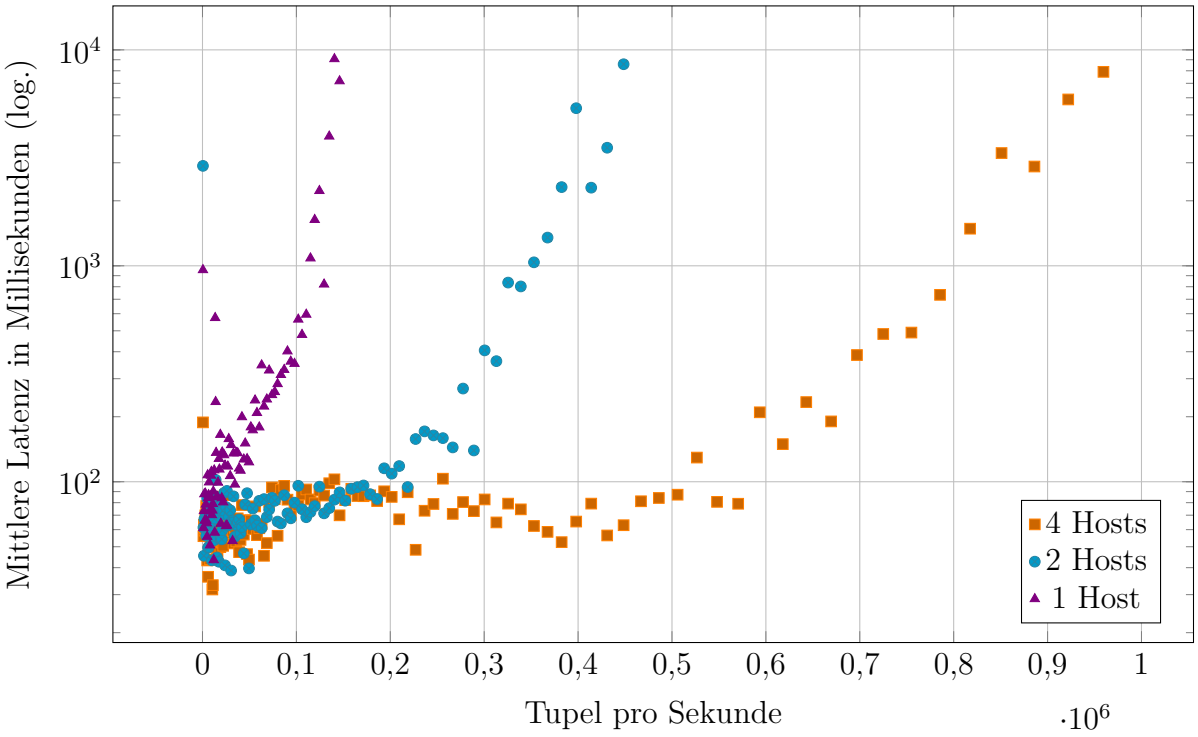


Abbildung 7.9: Skalierung von Flink

Apache Samza

Samza fordert bei YARN entsprechend der Konfiguration Container zur Ausführung von SamzaContainern an. Sie enthalten jeweils einen TaskRunner und die Tasks. Innerhalb des Containers wird zur Bearbeitung der Tasks nur ein Thread genutzt. Daher skaliert Samza über die Anzahl der Container. Die Anzahl der Tasks wird anhand der Kafka-Partitionen

bestimmt. Zur Messung wurden 32 Partitionen und 18 Container pro Rechenknoten genutzt. [Apa15aj]

Abbildung 7.10 stellt die Messergebnisse dar. Die Konfiguration mit einem Rechenknoten konnte nicht getestet werden, da der Samza ApplicationMaster aufgrund von Ressourcenbeschränkungen nicht startet. Die Latenz erhöht sich mit steigender Geschwindigkeit zunächst stark, der Anstieg flacht im weiteren Verlauf aber ab. Bei einer Latenz von einer Sekunde werden 20.000 beziehungsweise 60.000 Tupel pro Sekunde Durchsatz erreicht. Bei einer höheren Latenz ist noch mehr Durchsatz möglich. Mit 2,5 Sekunden Latenz werden etwa 31.000 beziehungsweise 110.000 Tupel pro Sekunde mit zwei beziehungsweise vier Rechenknoten verarbeitet.

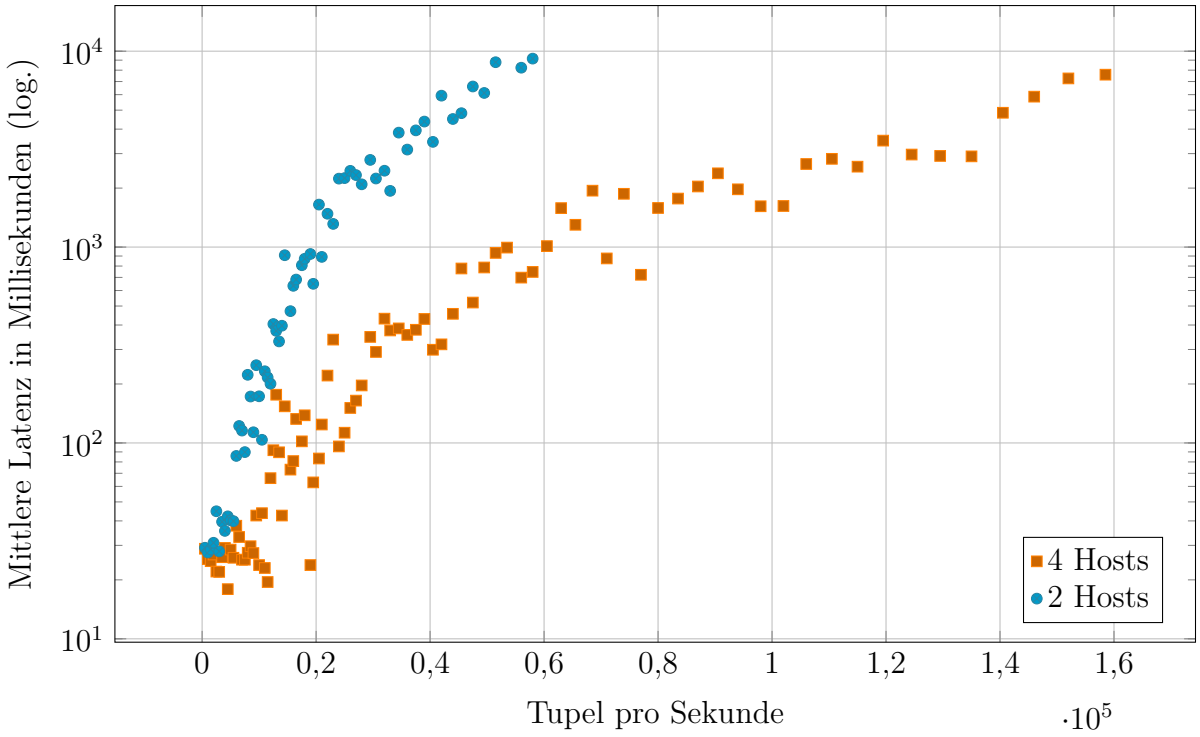


Abbildung 7.10: Skalierung von Samza

Apache Spark

Spark arbeitet mit Micro-Batches, deren Verarbeitung bei jedem Batch neu geplant und gestartet wird. Die Rechenknoten puffern die eingehenden Tupel von Kafka und verarbeiten sie erst, wenn der Batch komplett ist. Dieses Verfahren erhöht die Latenz stark, da der Aufwand für das Starten der Verarbeitung prinzipiell bei jedem Batch

anfällt. Aktuell wird durch Spark immer nur ein Batch gleichzeitig verarbeitet. Wenn die Verarbeitung eines Batches länger dauert als das Batch-Intervall, wird der neue Batch in eine Warteschlange eingereiht. Zur Messung werden die Standardeinstellungen von Spark genutzt, sie sehen pro CPU-Kern einen Thread vor. [Apa15be; Das15]

Wie im vorherigen Abschnitt wurde ein Batch-Intervall von fünf Sekunden gewählt. Die Ergebnisse sind in Abbildung 7.11 dargestellt. Mit einer Latenz von 7,5 Sekunden können etwa 65.000, 77.000 beziehungsweise 102.000 Tupel pro Sekunde mit einem, zwei beziehungsweise vier Rechenknoten verarbeitet werden.

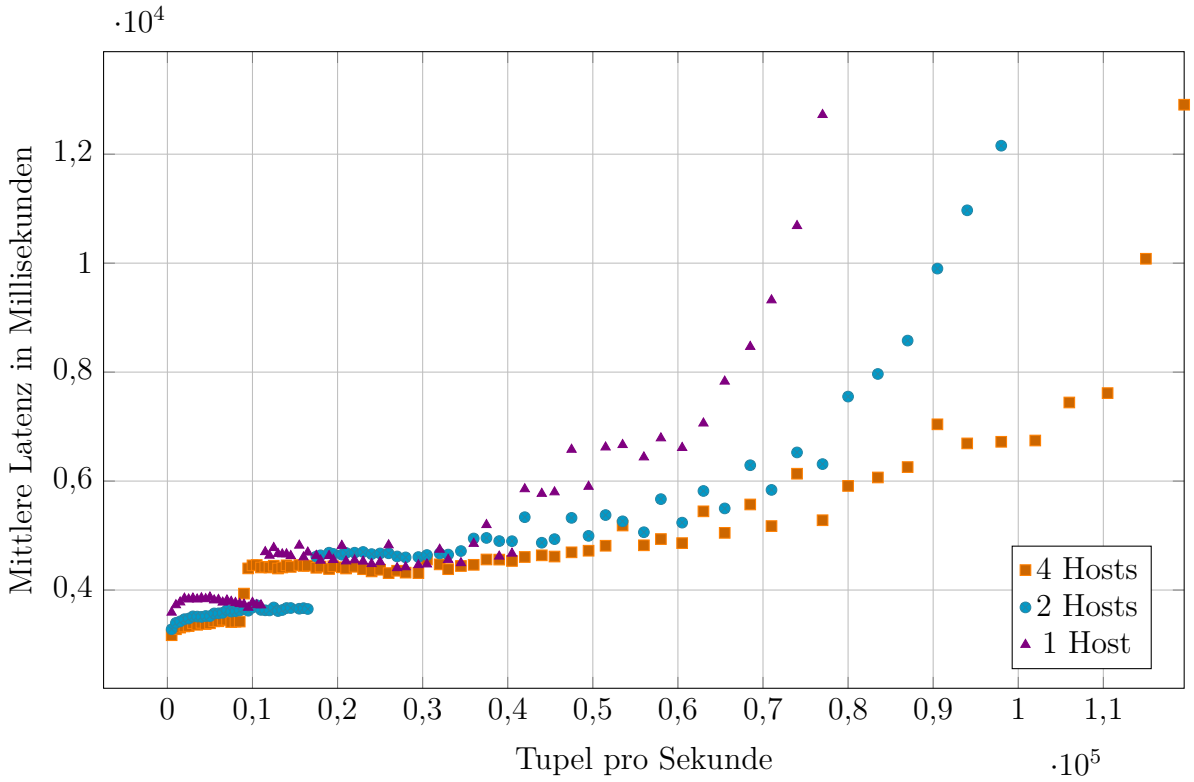


Abbildung 7.11: Skalierung von Spark

Apache Storm

An der Verarbeitung sind vier Komponenten beteiligt. Die Supervisor starten Worker-Prozesse, in denen *Executor*-Threads bereitgestellt werden. Innerhalb dieser Threads wird die eigentliche Verarbeitung in Tasks ausgeführt. Ein Task ist ein Verarbeitungsknoten oder eine Datenquelle. Die Parallelität wird über die Anzahl der Executors gesteuert. Sie wird initial beim Erstellen der Topologie festgelegt, kann aber auch zur Laufzeit

geändert werden. Die Anzahl der Tasks ist über die gesamte Laufzeit konstant, um zur Laufzeit skalieren zu können, müssen mehr Tasks als Executors genutzt werden. Zur Messung werden je nach Verarbeitungsknoten ein bis vier Threads pro CPU-Kern genutzt. [Apa14r]

Abbildung 7.12 stellt die Messergebnisse dar. Die Latenz bleibt über einen weiten Geschwindigkeitsbereich auf niedrigem Niveau. Bei einer Sekunde Latenz können etwa 52.000, 84.000 beziehungsweise 135.000 Tupel pro Sekunde mit einem, zwei beziehungsweise vier Knoten verarbeitet werden.

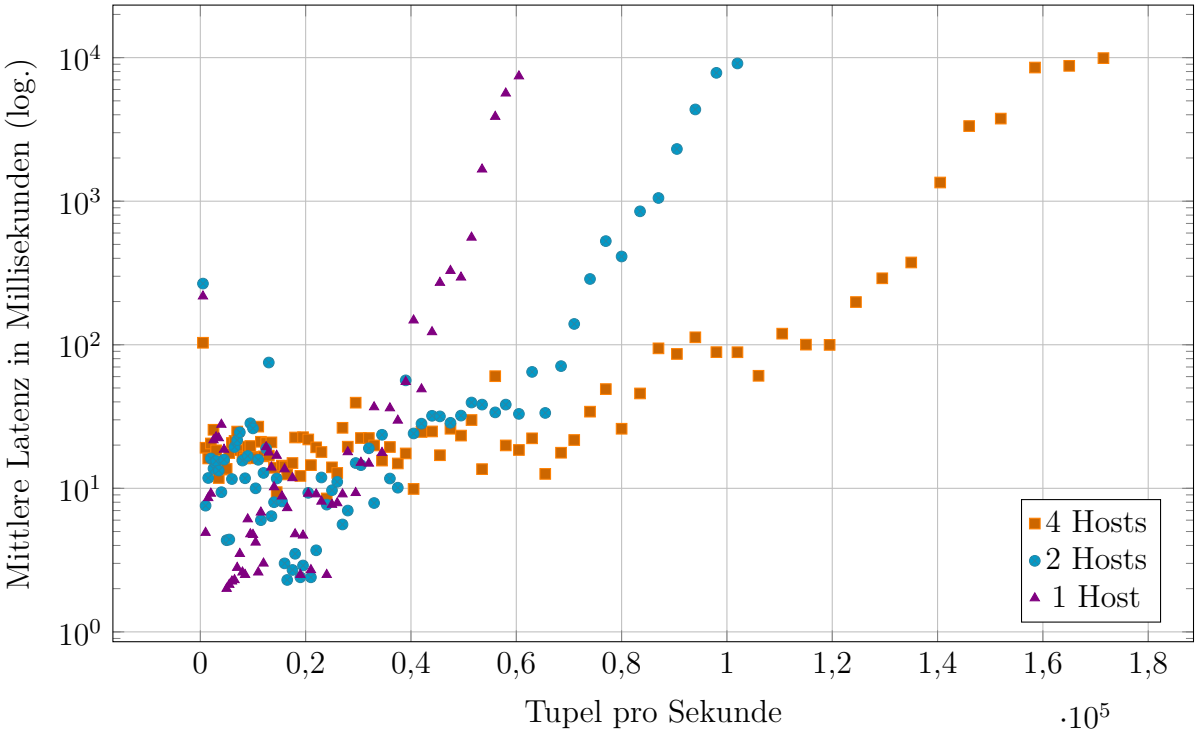


Abbildung 7.12: Skalierung von Storm

Apache Storm Trident

Die Parallelität von Storm Trident wird wie bei der gewöhnlichen Storm API konfiguriert. Es werden die selben Einstellungen genutzt.

Die Messergebnisse sind in Abbildung 7.13 dargestellt. Bedingt durch die Batch-orientierte Arbeitsweise ist die Latenz prinzipiell höher, als die von Storm. Der Anstieg der Latenz flacht mit größerer Rechenknotenanzahl bei steigender Geschwindigkeit ab. Bei einer

Latenz von einer Sekunde werden 41.000, 50.000 beziehungsweise 58.000 Tupel pro Sekunde erreicht. Ein höherer Durchsatz geht mit stark steigender Latenz einher. Bei 2,5 Sekunden Latenz können etwa 46.000, 61.000 beziehungsweise 94.000 Tupel pro Sekunde mit einem, zwei beziehungsweise vier Knoten verarbeitet werden.

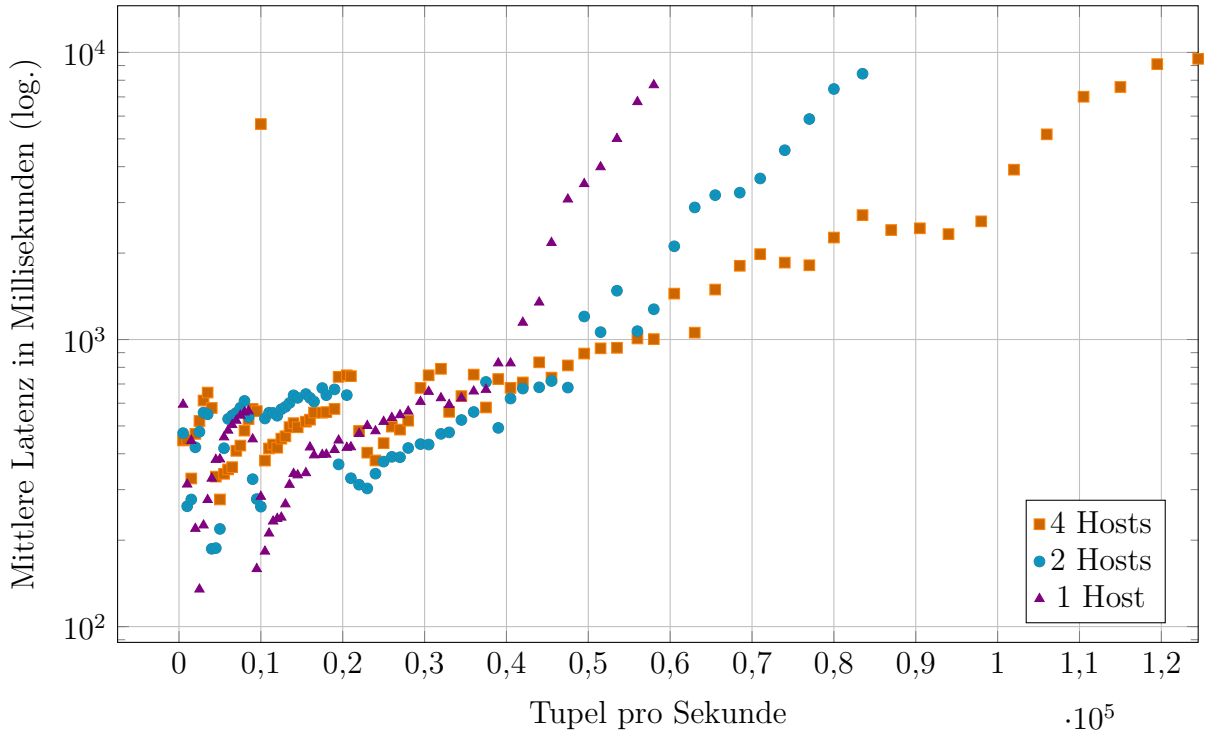


Abbildung 7.13: Skalierung von Storm Trident

Zusammenfassung

Die Tabelle 7.7 zeigt die Skalierung der Frameworks. Es wird das Verhältnis des maximalen Durchsatzes zwischen einem und zwei Hosts und zwei und vier Hosts dargestellt. Die zugrundeliegenden Durchsatzmessungen sind bei Flink und Storm bei einer Sekunde, bei Samza und Storm Trident bei 2,5 Sekunden und bei Spark bei 7,5 Sekunden durchgeführt worden.

Verhältnis	Flink	Samza	Spark	Storm	Storm Trident
1 : 2 Hosts	3,1	-	1,2	1,6	1,3
2 : 4 Hosts	2,3	3,5	1,3	1,6	1,5

Tabelle 7.7: Skalierung auf mehrere Worker

7.3 Compatibility

7.3.1 Integration mit bestehenden Clusterwerkzeugen

Apache Flink

Flink bietet mehrere Möglichkeiten zur Integration mit anderen Clusterwerkzeugen an. Es kann auf der eigenen Cluster-Implementierung, auf Hadoop YARN oder auf Apache Tez [Apa15bm] ausgeführt werden [Apa15o]. Letzeres ist aktuell im Alpha-Stadium und hat noch einige Einschränkungen, unter anderem ist die Streaming-API nicht verfügbar [Apa15m]. Zur Nutzung von Flink mit der Google Compute Engine [Goo15a] ist ein Skript beigelegt, das alle Komponenten automatisch aufsetzt [Apa15i].

Apache Samza

Apache Samza ist grundsätzlich so aufgebaut, dass verschiedene Clusterwerkzeuge genutzt werden können. Bisher besteht jedoch nur eine Implementierung für Apache Hadoop YARN. [Apa15ag]

Apache Spark

Apache Spark unterstützt drei verschiedene Clusterwerkzeuge. Es kann im Standalone-Modus, mit Apache Hadoop YARN oder Apache Mesos [Apa15aa] betrieben werden. Während im Standalone-Modus und mit YARN die Ressourcen fest zugewiesen werden und anderen Anwendungen nicht zur Verfügung stehen, können mit Mesos die Ressourcen dynamisch zugeteilt werden. Spark kann mit einem beigelegten Skript in der Amazon Elastic Compute Cloud (EC2) [Ama15a] automatisch einen Cluster aufbauen. [Apa15ax]

Apache Storm

Apache Storm unterstützt nur den Standalone-Modus. Yahoo stellt eine Erweiterung bereit, um Storm auf YARN zu nutzen. Sie hat aber einige Einschränkungen und wurde schon über 10 Monate nicht mehr aktualisiert [Yah14]. Außerdem existiert ein Projekt, um Storm automatisch in der Amazon Elastic Compute Cloud zu starten, dieses wird aber ebenfalls seit längerer Zeit nicht mehr gepflegt [Mar13]. [Apa14l]

Zusammenfassung

Clusterwerkzeug	Flink	Samza	Spark	Storm
Standalone	✓	-	✓	✓
Apache YARN	✓	✓	✓	(✓)
Weitere	Google Compute Engine, (Apache Tez)	-	Apache Mesos, Amazon EC2	(Amazon EC2)

Tabelle 7.8: Unterstützte Clusterwerkzeuge

7.3.2 Anbindung von Ein- und Ausgabesystemen

Apache Flink

Flink kann TCP-Sockets oder Dateien zur Ein- und Ausgabe verwenden. Es können lokale Dateisysteme, HDFS und Amazon S3 [Ama15b] genutzt werden. Apache Kafka und RabbitMQ [Piv15] werden als externe Nachrichtensysteme unterstützt. Beide Systeme können zur Ein- und Ausgabe benutzt werden. Zusätzlich wird die Twitter Streaming API als Quelle unterstützt. Lediglich für Apache Kafka wird eine fehlertolerante Anbindung bereitgestellt. [Apa15g]

Eigene Datenquellen müssen das `SourceFunction`-Interface implementieren. Über die `run()`-Methode startet Flink die Datenquelle. Die Anbindung übergibt mithilfe eines `SourceContext`-Objekts eingehende Tupel an Flink. Mithilfe der Checkpoint-Infrastruktur können fehlertolerante Quellen erstellt werden. Abbildung 7.14 stellt den Aufbau einer Datenquelle dar.

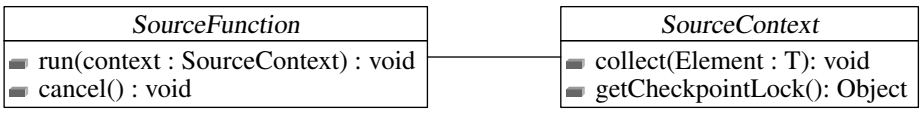


Abbildung 7.14: Datenquelle in Flink

Datensenken werden über das `SinkFunction`-Interface angebunden. Flink ruft die dort definierte `invoke()`-Methode auf, wenn ein neues Tupel ausgegeben werden soll. Das Interface wird in Abbildung 7.15 dargestellt.

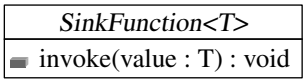


Abbildung 7.15: `SinkFunction<T>`-Interface in Flink

Apache Samza

Samza unterstützt nur Kafka zur Ein- und Ausgabe von Daten. Eine eigene Datenquelle muss immer über Kafka angebunden werden. Eine eigene Datensenke kann mithilfe der `StreamTask`-Klasse erstellt werden. Die Tupel gehen über die `process()`-Methode ein und werden dann gespeichert. Das Interface ist in Abbildung 7.16 dargestellt.

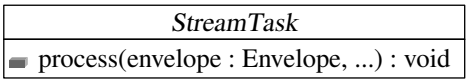


Abbildung 7.16: `StreamTask`-Interface von Samza

Apache Spark

Spark unterstützt als Quelle TCP-Sockets und Dateien. Es können lokale Dateisysteme, HDFS und Amazon S3 genutzt werden. Als externe Datenquellen können Akka-Datenströme, Kafka, Apache Flume [Apa12a], Amazon Kinesis [Ama15c], ZeroMQ [iMa14], MQTT [SN15] und Twitter verwendet werden. Für eigene Anbindungen steht die abstrakte Klasse `Receiver` zur Verfügung. Ein Teil der Klasse wird in Abbildung 7.17 dargestellt. Die `onStart()`- und `onStop()`-Methode muss implementiert werden. Daten werden durch die `store()`-Methoden weitergegeben. Sie bestimmen die Semantik der Datenquelle. Die Methode, die einen einzelnen Datensatz entgegennimmt, ist nicht blockierend, daher können bei einem Ausfall Daten verloren gehen. Die Methode, die mehrere Datensätze entgegennimmt, blockiert, solange die Datensätze repliziert werden. Dadurch können fehlertolerante Datenquellen implementiert werden. [Apa15bd; Apa15be]

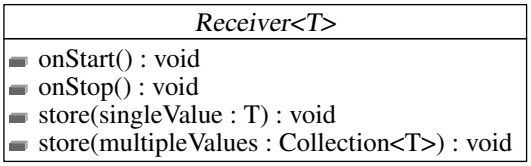


Abbildung 7.17: `Receiver`-Interface von Spark

Spark unterstützt als Datensenken nur Dateien. Externe Systeme müssen durch das `Function`-Interface angebunden werden, das durch die `foreachRDD()`-Operation aufgerufen wird. Der `call()`-Methode wird der aktuelle Batch übergeben, um ihn zu speichern. Die Funktion selbst wird auf dem Driver ausgeführt, die Operationen auf dem Batch, wie etwa ein `foreach()`-Aufruf, werden auf die Worker verteilt. Wie bereits erwähnt sind Methoden zum Initialisieren und Finalisieren nicht vorgesehen. Das Interface ist in Abbildung 7.18 dargestellt. [Apa15be]

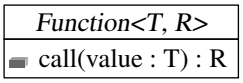


Abbildung 7.18: `Function<T, R>`-Interface von Spark

Apache Storm

Storm stellt Module für Microsoft Azure Event Hubs [Mic15], Apache HBase [Apa15v], HDFS, Apache Hive [Apa15w], *Java Database Connectivity (JDBC)* [Ora15c], Kafka und Redis bereit [Apa15bk]. Alle Module außer das HDFS-Modul unterstützen die Ein- und Ausgabe von Tupeln. Das HDFS-Modul kann nur Tupel ausgeben. Die Module sind sowohl für die gewöhnliche Storm API als auch für Storm Trident geeignet. Die Erstellung von eigenen Anbindungen wird in den beiden folgenden Absätzen getrennt für die Storm und Storm Trident API erläutert. Obwohl sich beide APIs an machen Stellen syntaktisch und semantisch stark ähneln, sind sie völlig getrennt und daher inkompatibel.

Storm API Die Storm API stellt zur Anbindung von eigenen Datenquellen die abstrakte Klasse `BaseRichSpout` bereit. Sie wird in Abbildung 7.19 dargestellt. Durch Storm wird periodisch die `nextTuple()`-Methode aufgerufen, um Daten abzufragen. Wenn ein Tupel erfolgreich verarbeitet wurde, wird die `ack()`-Methode aufgerufen. Danach kann das Tupel aus dem Quellsystem gelöscht werden. Bei einem Fehler wird die `fail()`-Methode aufgerufen, damit das Tupel aus dem Quellsystem nochmals abgerufen werden kann. Das heißt, für eine fehlertolerante Verarbeitung muss das Quellsystem die Daten so lange vorhalten, bis sie bestätigt werden. [Apa15bh]

Für die Anbindung von Datensenken werden Verarbeitungsknoten in Form der Klasse `BaseBasicBolt` genutzt. Sie ist in Abbildung 7.20 dargestellt. Die `execute()`-Methode

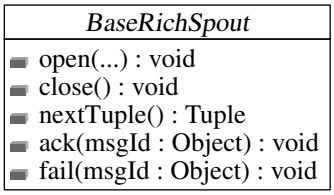


Abbildung 7.19: `BaseRichSpout`-Interface von Storm

wird aufgerufen, wenn neue Tupel zur Verfügung stehen. Nach dieser Methode wird das Tupel automatisch beim Acker bestätigt. [Apa15bh]

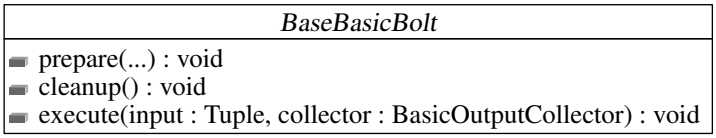


Abbildung 7.20: `BaseBasicBolt`-Interface von Storm

Storm Trident API Storm Trident stellt, je nach gewünschter Fehlertoleranz-Semantik, verschiedene Schnittstellen für Datenquellen zur Verfügung. Es wird exemplarisch die transaktionale Schnittstelle erklärt, bei der sich eine einmal ausgegebene Transaktionsnummer immer auf die gleichen Tupel bezieht (siehe Abschnitt 7.5.2). Sie besteht aus den zwei Interfaces `Coordinator` und `Emitter`. Die Interfaces sind in Abbildung 7.21 dargestellt. Der `Coordinator` verwaltet Metadaten, wie die Zuordnung von Partitionen zu den Hosts der Datenquelle. Der `Emitter` liest die Daten vom Quellsystem und emittiert sie als Tupel. Durch Metadaten, die als Parameter an `emitPartitionBatch()` übergeben werden, kann ein Batch im Fehlerfall wiederhergestellt werden. Die Metadaten werden als Rückgabewert der Funktion ausgegeben und durch Storm in ZooKeeper gespeichert. [Apa14p]

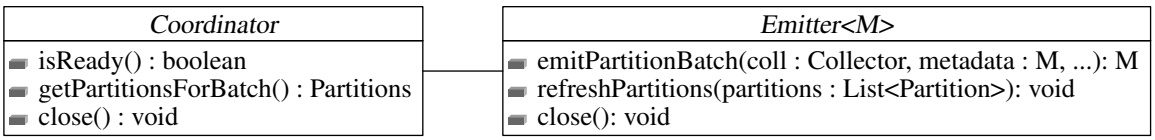


Abbildung 7.21: Datenquelle in Storm Trident

Eigene Datensenken werden durch die abstrakte Klasse `BaseFunction` abgebildet. Sie ist in Abbildung 7.22 dargestellt. Wenn neue Tupel zur Verfügung stehen, wird die `execute()`-Methode aufgerufen. [Apa14o]

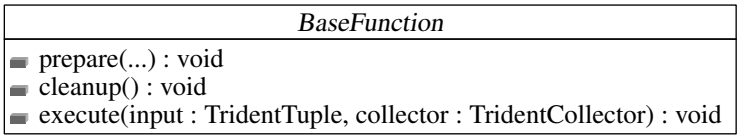


Abbildung 7.22: `BaseFunction`-Interface von Storm Trident

Zusammenfassung

Anbindung	Flink	Samza	Spark	Storm
Dateien	✓	-	✓	(✓)
TCP-Sockets	✓	-	✓	-
Apache Kafka	✓	✓	✓	✓
Weitere	RabbitMQ	-	Apache Flume, Akka, Amazon Kinesis, MQTT, ZeroMQ	Microsoft EventHubs, Apache HBase, Apache Hive, JDBC, Redis
Eingabe erweiterbar	✓	-	✓	✓
Ausgabe erweiterbar	✓	✓	✓	✓

Tabelle 7.9: Unterstützte Ein- und Ausgabesysteme

7.4 Usability

7.4.1 Dokumentation des Frameworks

Apache Flink

Flink bietet eine umfangreiche Online-Dokumentation [Apa15g], die sich in die Bereiche Installation/Konfiguration, Programmierung, Bibliotheken und Interna gliedert. Für jeden Bereich gibt wiederum einige Unterseiten. Die für den Streaming-Modus relevante Dokumentation enthält eine Anleitung zur Maven-Konfiguration, ein Beispielprogramm

und eine ausführliche Beschreibung der Operatoren. Es fehlt eine Beschreibung der Fehlertoleranzmechanismen. Außerdem ist das gezeigte Beispiel der fehlertoleranten Kafka Anbindung nicht korrekt, es bezieht sich auf eine veraltete Version. Die Optionen der Konfigurationsdatei werden ausführlich und mithilfe von Hintergrundinformationen über die Architektur von Flink erklärt. Es liegen sowohl für Java als auch für Scala mehrere Beispielprogramme vor, die an entscheidenden Stellen mit Kommentaren versehen sind. Flink stellt eine mit *JavaDoc* beziehungsweise *ScalaDoc* generierte API-Dokumention zur Verfügung, die insbesondere die nutzerseitige Schnittstelle abdeckt.

Apache Samza

Die Dokumentation von Samza [Apa15al] enthält eine Einführung in das Thema Stream Processing, Vergleiche mit anderen Frameworks, eine Beschreibung der API und Informationen zu den großen Themenbereichen Laufzeitumgebung, Jobs, YARN und Betrieb der Anwendung. Anhand von Beispielen werden typische Anwendungsfälle, wie das Verbinden von verschiedenen Datenströmen, beschrieben. Bei Samza werden Tasks durch eine Konfigurationsdatei umfangreich angepasst, daher steht eine eigene Anleitung mit detaillierten Erklärungen bereit. Das Beispiel „hello-samza“ [Apa15am] wird genutzt, um die Einrichtung der Betriebsumgebung schrittweise zu erklären. Das Beispiel ist auch eine Demonstration der wesentlichen Eigenschaften der API. Samza stellt eine mit JavaDoc generierte API-Dokumentation zur Verfügung.

Apache Spark

Spark hat eine Hauptseite zum Thema Streaming [Apa15be], die zunächst einen Überblick gibt und Schritt für Schritt ein Beispiel vorstellt. Anschließend werden die grundlegenden Funktionen von Spark Streaming erklärt. Dies umfasst die Operatoren, die Datenquellen, die Ausgabe, das Bereitstellen der Anwendung und die Überwachung im Betrieb. Zur Verdeutlichung sind Quelltext-Beispiele in allen drei Programmiersprachen eingebunden. Die einzelnen Aspekte der Hauptseite verweisen meist für eine genauere Beschreibung auf weitere Unterseiten. Es wird ausführlich auf Fehlertoleranzmechanismen eingegangen. Die Konfiguration von Spark und weitere Interna werden ebenfalls erläutert. Spark liegen Beispielanwendungen und API-Dokumentationen in allen unterstützten Sprachen bei.

Apache Storm

Storm stellt ein Tutorial [Apa15bh] bereit, das die grundlegenden Begriffe anhand eines Quelltext-Beispiels erklärt. Zusätzlich gibt es eine Dokumentation [Apa15bi], die das Konzept und die Implementierung von Storm erläutert. Sie ist in die Abschnitte Grundlagen, Trident, Einrichtung, Fortgeschrittene und Experten gegliedert. Die Dokumentation wirkt stellenweise unstrukturiert und unübersichtlich. Einige Seiten sind bereits veraltet, werden aber trotzdem referenziert. Manche Informationen fehlen gänzlich, etwa zur Ausgabe von Metriken, wie in Abschnitt 7.7.2 dargestellt. Die Online-Dokumentation ist nicht versioniert. Manche Themen, wie Sicherheit und die Web-API, sind nicht in der Online-Dokumentation beschrieben, sondern nur im Projektarchiv [Apa15bl]. Storm liefert einige Quelltext-Beispiele für verschiedene Sprachen mit, von denen manche vollständig unkommentiert sind. Eine mit JavaDoc generierte Dokumentation steht zur Verfügung, ein Großteil der Klassen ist aber nicht näher beschrieben.

Zusammenfassung

Die folgende Tabelle enthält eine Zusammenfassung der Ergebnisse. Zusätzlich wird der Umfang der Anleitung in Wörtern gemessen. Dabei wird nur der für das Streaming relevante Anteil herangezogen. Die Anzahl wurde durch das Werkzeug *wc* [OI13] ermittelt. Der Anteil an Kommentaren am Quelltext wird angegeben, um die Menge an API-Dokumentation abschätzen zu können. Er wurde mithilfe des Werkzeugs *cloc* [Dan15] ermittelt.

Art der Dokumentation	Flink	Samza	Spark	Storm
Tutorial	-	(✓)	✓	✓
Anleitung	✓	✓	✓	(✓)
Beispiele	✓	✓	✓	(✓)
API-Dokumentation	✓	✓	✓	(✓)
Umfang der Anleitung in Wörtern	56.000	38.000	52.000	51.000
Anteil von Kommentaren am Quelltext	29 %	36 %	30 %	18 %

Tabelle 7.10: Dokumentation der Frameworks

7.4.2 Ersteinrichtung des Frameworks

Die genaue Beschreibung der Ersteinrichtung findet bereits in Unterkapitel 6.3 statt, daher werden an dieser Stelle nur die Ergebnisse in Tabelle 7.11 zusammengefasst.

Kriterium	Flink	Samza	Spark	Storm
Installationsanleitung	✓	✓	✓	✓
Voraussetzungen	HDFS	YARN, HDFS, Kafka, ZooKeeper	HDFS	ZooKeeper, (supervisord)
Zentrale Bereitstellung in einem Verzeichnis	✓	✓	✓	-
Start/Stop des Clusters per Skript	✓	✓	✓	-
Anleitung zur Projekteinrichtung	✓	(✓)	✓	✓

Tabelle 7.11: Ersteinrichtung der Frameworks

7.4.3 Bereitstellung von Anwendungen

Apache Flink

Flink bietet eine CLI und eine Web-GUI zur Bereitstellung von Anwendungen. Über beide Methoden können Anwendungen gestartet, gestoppt und aufgelistet werden. Die Web-GUI besteht aus zwei Komponenten: einer Oberfläche zum Starten der Anwendung und zur Darstellung des Ausführungsplans und dem Dashboard, um laufende Anwendungen zu verwalten. Beide Oberflächen sind in Abbildung 7.23 und Abbildung 7.24 dargestellt. Anwendungen werden mit ihren Abhängigkeiten in eine JAR gepackt und über eines der beiden Werkzeuge an den Cluster gesendet. Die Aktualisierung von Anwendungen ist nicht vorgesehen. Eine Möglichkeit ist, die neue Version bereitzustellen und anschließend die alte Version zu beenden. [Apa15e; Apa15n]

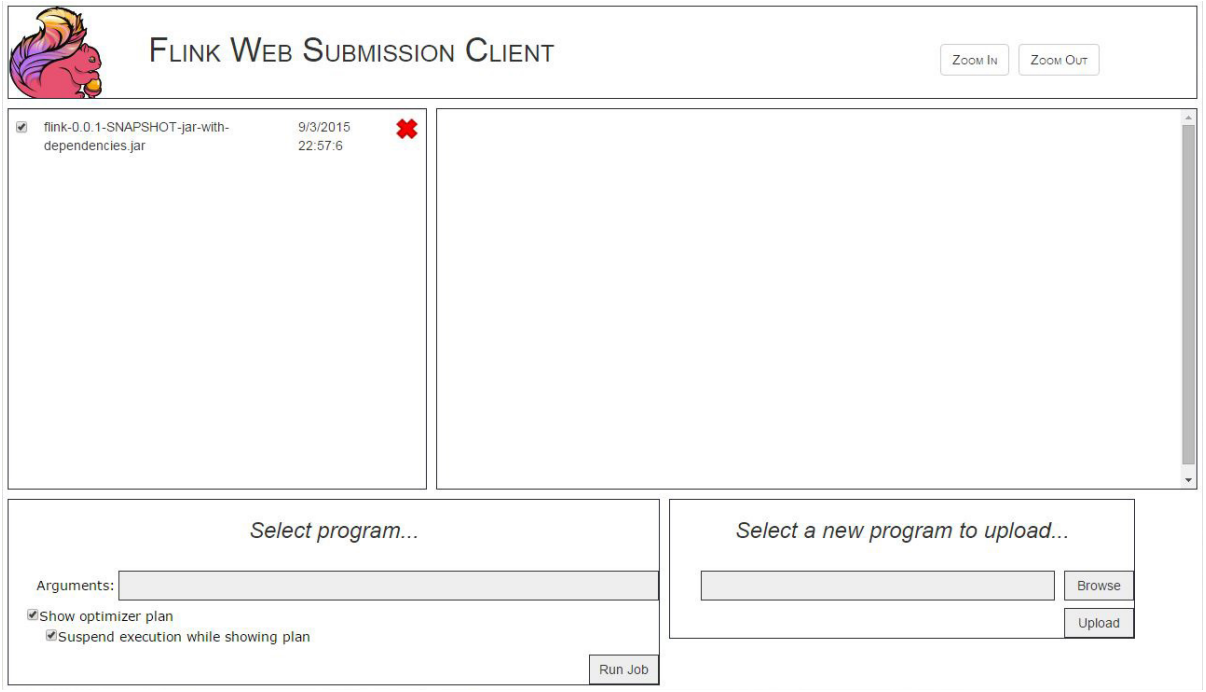


Abbildung 7.23: Flink Web-GUI zum Starten von Anwendungen

Apache Samza

Bei Samza wird die JAR so gepackt, dass sie alle zum Ausführen notwendigen Komponenten enthält. Ein in der JAR enthaltenes Skript startet die Anwendung auf YARN. Dazu wird dem Skript der Pfad zur Konfigurationsdatei der Anwendung übergeben, in der unter anderem der Pfad zur JAR enthalten ist. Die NodeManager laden sie vom angegebenen Pfad und führen sie aus, daher muss der Pfad für alle Rechenknoten verfügbar sein. Um die laufenden Anwendungen aufzulisten oder Anwendungen zu beenden wird die CLI oder Web-GUI von YARN genutzt. Letztere ist in Abbildung 7.25 dargestellt. Bei Samza lassen sich die einzelnen Tasks, welche die Anwendung bilden, flexibel austauschen, daher ist die Aktualisierung von Anwendungen einfacher möglich. [Apa15ap]

Apache Spark

Spark stellt eine CLI zum Bereitstellen und Verwalten von Anwendungen bereit. Zur Verwaltung ist zusätzlich eine Web-GUI vorhanden. Abbildung 7.26 stellt die Web-GUI dar. Die JAR kann auf mehrere Arten zur Verfügung gestellt werden, wie etwa als lokale Datei oder per HDFS. Ein gemeinsames Verzeichnis ist nicht nötig, Spark kann die

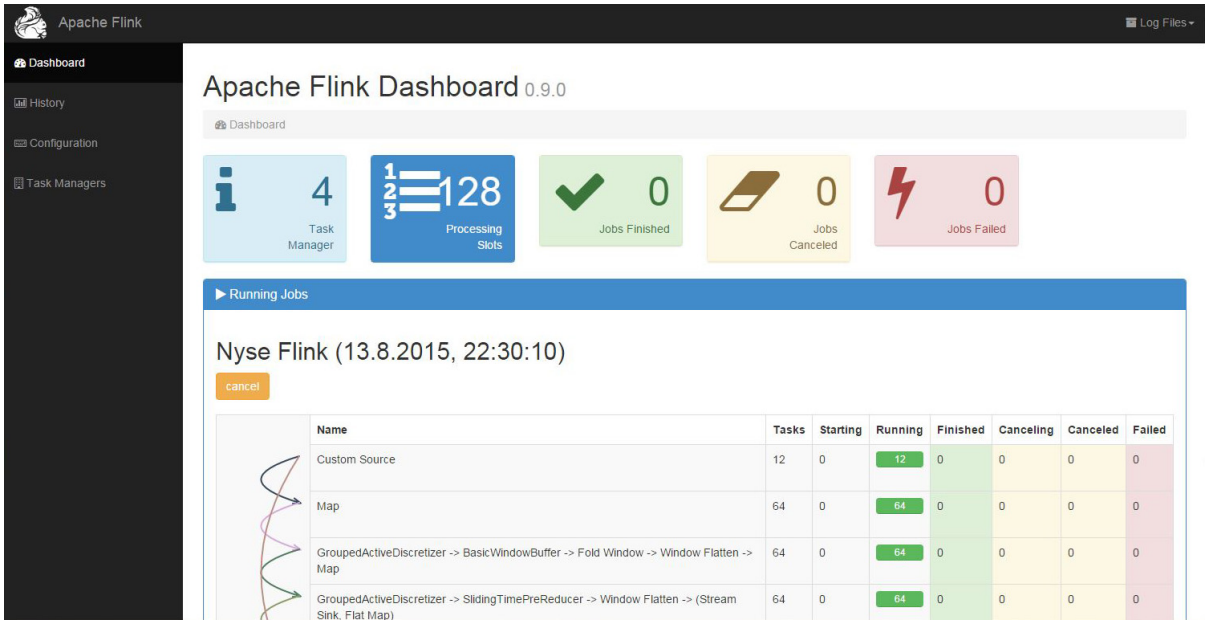


Abbildung 7.24: Flink Web-GUI zur Verwaltung von Anwendungen

JAR selbst verteilen. Bei Spark übernimmt nicht alleine der Master die Ausführung der Anwendung, es gibt zusätzlich noch den Driver, der den SparkContext hält. Der Driver wird standardmäßig auf dem Host ausgeführt, der die Anwendung startet. Bei lang laufenden Streaming-Anwendungen sollte der Driver aber auf einem Worker ausgeführt werden, da nur dort eine Option zur Verfügung steht, um ihn im Fehlerfall neu zu starten. Zur Aktualisierung von Anwendungen gibt es keine Hilfestellung. Für einen Betrieb ohne Unterbrechung müssen die alte und neue Version parallel betrieben werden. [Apa15ax; Apa15bg]

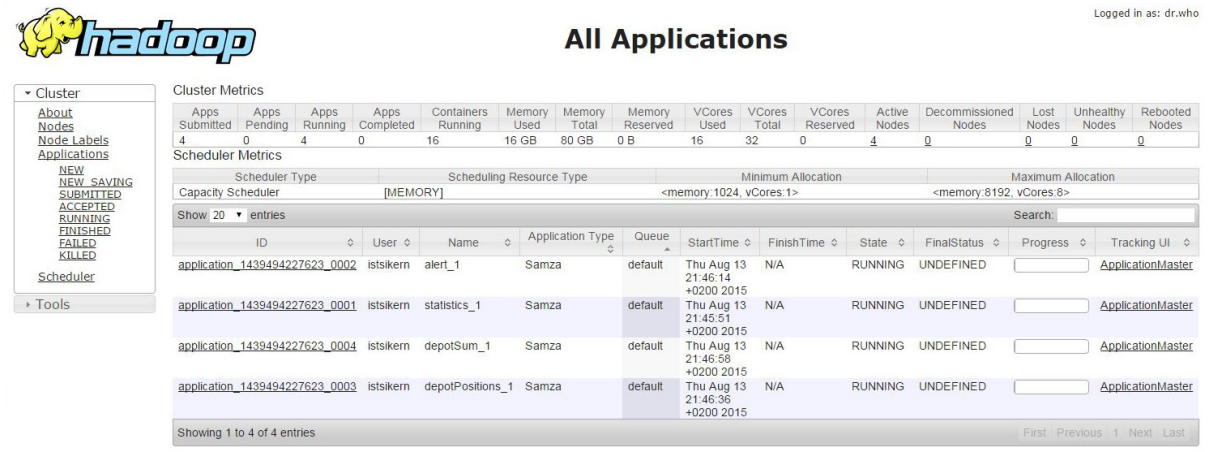
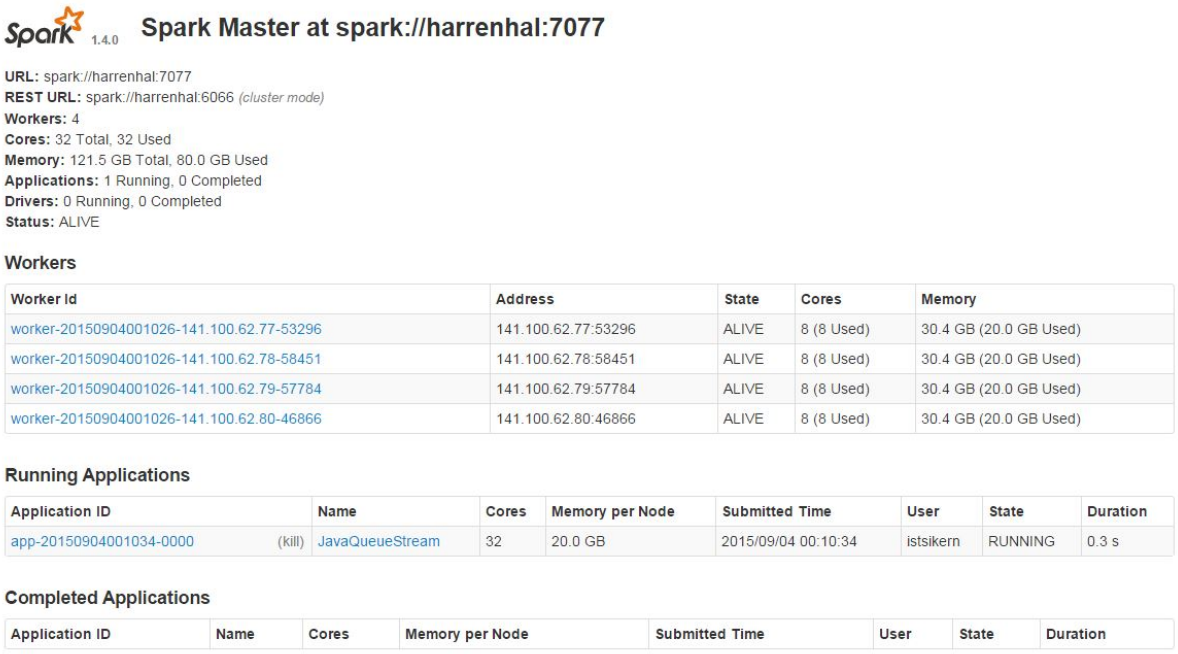


Abbildung 7.25: Auszug aus der Hadoop YARN Web-GUI



Spark 1.4.0 **Spark Master at spark://harrenhal:7077**

URL: spark://harrenhal:7077
REST URL: spark://harrenhal:6066 *(cluster mode)*
Workers: 4
Cores: 32 Total, 32 Used
Memory: 121.5 GB Total, 80.0 GB Used
Applications: 1 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20150904001026-141.100.62.77-53296	141.100.62.77:53296	ALIVE	8 (8 Used)	30.4 GB (20.0 GB Used)
worker-20150904001026-141.100.62.78-58451	141.100.62.78:58451	ALIVE	8 (8 Used)	30.4 GB (20.0 GB Used)
worker-20150904001026-141.100.62.79-57784	141.100.62.79:57784	ALIVE	8 (8 Used)	30.4 GB (20.0 GB Used)
worker-20150904001026-141.100.62.80-46866	141.100.62.80:46866	ALIVE	8 (8 Used)	30.4 GB (20.0 GB Used)

Running Applications

Application ID		Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20150904001034-0000	(kill)	JavaQueueStream	32	20.0 GB	2015/09/04 00:10:34	istsikern	RUNNING	0.3 s

Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Abbildung 7.26: Auszug aus der Spark Master Web-GUI

Apache Storm

Storm bietet eine CLI zur Bereitstellung und Verwaltung von Anwendungen. Die fertig gepackte JAR wird durch das Werkzeug in den Cluster geladen und gestartet. Außerdem kann die Anzahl der Executors während des Betriebs geändert werden. In der zusätzlichen Web-GUI können ebenfalls die Anwendungen aufgelistet, aktiviert, deaktiviert und beendet werden. Sie ist in Abbildung 7.27 abgebildet. Das Aktualisieren der Anwendung muss aktuell manuell vorgenommen werden. Zur Zeit wird eine Funktion entwickelt, um eine Anwendung ohne das komplette Unterbrechen der Verarbeitung zu aktualisieren. [Apa14k; Xu13b]

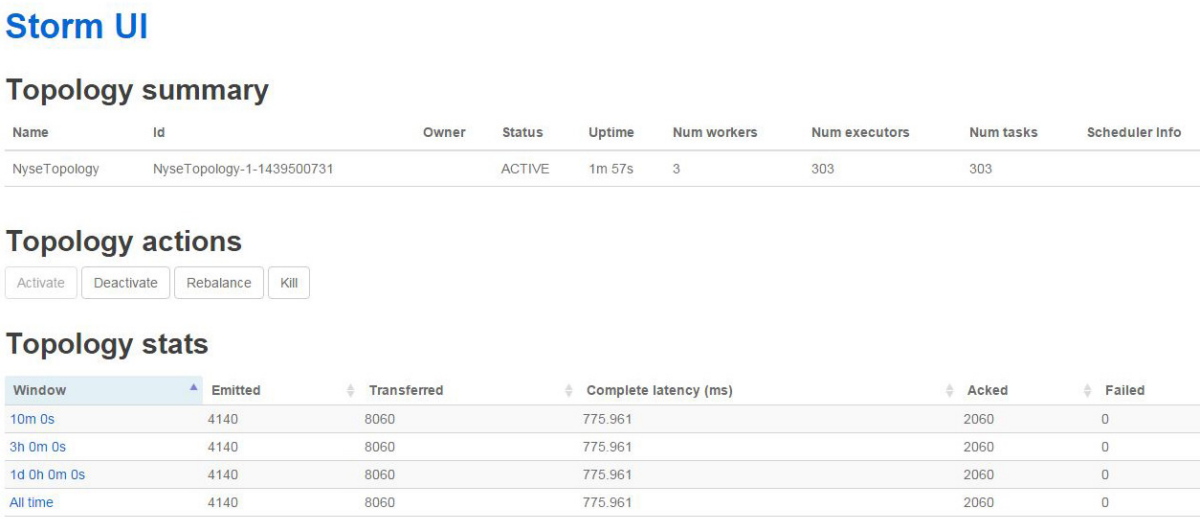


Abbildung 7.27: Auszug aus der Storm Web-GUI

Zusammenfassung

Funktion	Flink	Samza	Spark	Storm
Bereitstellung per CLI	✓	✓	✓	✓
Bereitstellung per Web-GUI	✓	-	-	-
Bereitstellung ohne gemeinsames Verzeichnis	✓	-	✓	✓
Verwaltung per CLI	✓	(✓)	✓	✓
Verwaltung per Web-GUI	✓	(✓)	✓	✓
Prozess zur Aktualisierung von Anwendungen	-	(✓)	-	-

Tabelle 7.12: Bereitstellung von Anwendungen

7.4.4 Benutzerfreundlichkeit der API

Apache Flink

Bei Flink ist das Working Framework durch die Typisierung und die Möglichkeit Objektattribute für den Zustand zu nutzen auf die aktuelle Aufgabe beschränkt.

Kenntnis über Implementierungsdetails ist nur in wenigen Fällen notwendig, etwa wenn durch die mangelnde Compilerunterstützung von Java 8 Lambda-Funktionen nicht nutzbar sind. Sonst ist die API durch das Fluent Interface und die sinnvolle Wahl von Parameter-

und Funktionsnamen leicht erfassbar. Durch diese Maßnahmen lässt sich die API gut ohne weitere Kenntnisse nutzen.

Die API von Flink ist durchgehend konsistent gehalten. Die selben Operatoren werden bei gewöhnlichen Datenströmen, gruppierten Datenströmen und Windows angeboten. Für alle Typen können die selben Funktionen genutzt werden.

Apache Samza

Bei der Implementierung der Aufgabe in Samza müssen durch die nicht-typisierte Schnittstelle stets die anderen Tasks in den Kontext einbezogen werden. Eine große Rolle spielt außerdem die externe Konfigurationsdatei, in der Einstellungen, wie die Eingabe-Datenströme und die Klassen zur Serialisierung, angegeben werden müssen. Bei einem neuen Datentyp muss ein neuer Serialisierer implementiert und konfiguriert werden. Dadurch ist als Working Framework beim Lösen einer Aufgabe die ganze Anwendung zu betrachten.

Die grundlegende Samza-API beschränkt sich auf wenige, klar strukturierte Interfaces, die jeweils nur ein bis zwei Methoden bereitstellen. Die Methoden sind sinnvoll benannt, beschränken sich auf wenige Parameter und sind per JavaDoc dokumentiert. Das Verbinden von Datenströmen durch die Konfiguration und das anschließende Auswerten des Eingabe-Datenstroms ist nur anhand der API nicht sofort ersichtlich. Auch die doppelte Bedeutung des Partition Keys erschließt sich erst beim Blick in die Dokumentation. Daher ist für manche Aufgaben etwas mehr Kontext, als nur die API, nötig.

Die Konsistenz der API ist vollständig gegeben. So sind die Methoden der eingehenden und ausgehenden Nachrichten analog aufgebaut. Die Methode, die bei eingehenden Nachrichten und beim Berechnen des Windows aufgerufen wird, ist ebenfalls gleich.

Apache Spark

Spark bietet typisierte Datenströme und eine umfangreiche API an. Dadurch können die Aufgaben meist in einem lokalen Kontext gelöst werden. Lediglich Einstellungen, wie das Batch-Intervall, werden auf Anwendungsebene getroffen.

Spark abstrahiert die meisten Implementierungsdetails in sinnvollem Maße. Die API ist teilweise als Fluent Interface aufgebaut, hat aber durch kombinierte Methoden etwas

Redundanz. Die Methoden und Parameter sind sinnvoll benannt. Lediglich der Gültigkeitsbereich der Variablen im Zusammenspiel mit Driver und Workern ist ohne die Dokumentation nicht zu erfassen. Daher werden in manchen Kontexten Kenntnisse über die konkrete Implementierung benötigt.

Die Konsistenz der API ist bei Spark nur teilweise gegeben. So können etwa bei gruppierten Datenströmen zwar auch Map- und Reduce-Operationen ausgeführt werden, sie nutzen aber andere Argumente als nicht gruppierte Datenströme.

Apache Storm

Bei Storm muss durch die nicht typisierte Schnittstelle immer auch der produzierende Bolt mit einbezogen werden. Durch die Verknüpfung der Verarbeitungsknoten auf Namensbasis können aber Namenskonventionen genutzt werden, die den Aufwand verringern. Trotzdem muss der Kontext der Anwendung in die Lösung der Aufgabe einbezogen werden.

Besonders die Nutzung der vielfältigen Groupings erschließt sich durch die API nicht sofort. Manche Methoden haben nur eine unzureichende Benennung der Parameter. Auch die vielen Einstellungsmöglichkeiten der Parallelität erfordern Hintergrundwissen über die Implementierung. In bestimmten Kontexten ist daher eine genauere Kenntnis von Storm nötig.

Konsistenz ist bei Storm kaum gegeben. Besonders die komplett getrennte Klassenhierarchie von Storm Trident ist unverständlich. Teilweise unterscheiden sich semantisch gleiche Konzepte in ihrer API. Auch innerhalb von Storm gibt es getrennte Klassenhierarchien und unterschiedliche Methoden- und Parameternamen für semantisch gleiche Konzepte, etwa bei Verarbeitungsknoten, die manuell Tupel bestätigen und solchen, die dies automatisch vornehmen.

Zusammenfassung

Kriterium	Flink	Samza	Spark	Storm
Working Framework	lokal	Anwendung	lokal	Anwendung
Penetrability	gut	kontextabh.	kontextabh.	kontextabh.
Consistency	konsistent	konsistent	neutral	inkonsistent

Tabelle 7.13: Benutzerfreundlichkeit der API

7.5 Reliability

7.5.1 Aktive Community

Die Ergebnisse dieses Kriteriums werden direkt in Tabelle 7.14 erfasst. Die Quellen dieser Tabelle sind in Tabelle 7.15 aufgelistet. Storm und Storm Trident werden nicht getrennt erfasst, da sie im gleichen Projekt entwickelt werden. Da Flink und Spark sowohl Batch- als auch Stream-orientiert arbeiten, werden für beide Frameworks die Kennzahlen einmal auf das gesamte Projekt und einmal nur auf den Streaming-Bereich bezogen dargestellt. Bei der Auswertung der Commits wird ein Commit dem Streaming-Bereich zugeordnet, wenn ein geänderter Dateipfad oder die Commit-Nachricht das Wort „streaming“ enthält. Als veröffentlichte Version wird jede Version erfasst, die auf „The Central Repository“ [Son11] veröffentlicht wurde. Unterschiedliche Ausprägungen einer Version, wie verschiedene Scala-Versionen, werden nicht mitgezählt. Alle Kennzahlen umfassen den Zeitrahmen vom 1. August 2014 bis zum 31. Juli 2015.

Kennzahl	Flink	Flink Streaming	Samza	Spark	Spark Streaming	Storm
Versionen	6	6	3	9	9	5
Commits	2.920	1.234	263	4.656	416	1.913
Commiter	110	47	38	597	102	106
Mailinglisteneinträge Nutzer	2.295	250	-	12.669	1.981	4.684
Mailinglisteneinträge Entwickler	5.977	1.590	2956	5.850	662	10.425[1]

Tabelle 7.14: Aktive Community

[1] Die Storm Entwicklermailingliste enthält auch automatische Emails von der Versionsverwaltung und dem Bugtracker

Flink	Versionskontrolle	https://github.com/apache/flink
	Mailingliste Nutzer	https://mail-archives.apache.org/mod_mbox/flink-user
	Mailingliste Entwickler	https://mail-archives.apache.org/mod_mbox/flink-dev
Samza	Versionskontrolle	https://github.com/apache/samza
	Mailingliste Nutzer	https://mail-archives.apache.org/mod_mbox/samza-dev
	Mailingliste Entwickler	-
Spark	Versionskontrolle	https://github.com/apache/spark
	Mailingliste Nutzer	https://mail-archives.apache.org/mod_mbox/spark-user
	Mailingliste Entwickler	https://mail-archives.apache.org/mod_mbox/spark-dev
Storm	Versionskontrolle	https://github.com/apache/storm
	Mailingliste Nutzer	https://mail-archives.apache.org/mod_mbox/storm-user
	Mailingliste Entwickler	https://mail-archives.apache.org/mod_mbox/storm-dev

Tabelle 7.15: Quellen der Tabelle 7.14

7.5.2 Verarbeitungsgarantien und Fehlertoleranz

Apache Flink

Apache Flink toleriert aktuell nur den Ausfall von TaskManagern. Durch den Ausfall des Masters brechen alle laufenden Anwendungen ab und neue lassen sich nicht mehr starten. Der Master bildet daher einen *Single Point of Failure (SPOF)*. Diese Beschränkung soll in der nächsten Version behoben werden [Apa15b]. Da die Fehlertoleranzmechanismen erst in der aktuellen Version eingeführt wurden, ist die Dokumentation noch sehr rudimentär [Apa15p].

Bei Ausfällen der TaskManager bietet Flink die Exactly-Once-Semantik an. Auch im Fehlerfall werden alle Tupel nur genau einmal verarbeitet und der Zustand der Operatoren entsprechend wiederhergestellt. Flink nutzt dazu *Asynchronous Barrier Snapshotting* [Car+15]. Es wird vom *Checkpoint Coordinator* im JobManager überwacht. In die Datenquellen werden periodisch *Stream Barriers* eingebracht. Sie fließen mit dem Datenstrom durch den gesamten Graph. Alle Tupel, die vor der Stream Barrier verarbeitet wurden, sind Teil des Snapshots, alle danach Teil des nächsten Snapshots. Sobald an einem Eingang eines Verarbeitungsknotens eine Stream Barrier eingegangen ist, werden an diesem Eingang keine weiteren Tupel mehr angenommen. Wenn an jedem Eingang eine Stream Barrier eingegangen ist, wird der Zustand des Operators gesichert und die Stream Barrier an alle nachfolgenden Verarbeitungsknoten weitergegeben. Benutzerdefinierte Funktionen können über eine Schnittstelle über Snapshots informiert werden und gegebenenfalls den eigenen Zustand sichern. Sobald die Stream Barrier an den Datensenken angekommen ist, ist der Snapshot erfolgreich abgeschlossen. Snapshots werden entweder beim JobManager oder im HDFS abgelegt. Damit die Wiederherstellung möglich ist, muss eine persistente Datenquelle genutzt werden, um ältere Tupel abzurufen. [Car+15]

Im Fehlerfall stoppt der JobManager die Verarbeitung, startet die Operatoren neu und stellt den letzten Snapshot wieder her. Anschließend beginnt die Verarbeitung der Tupel ab dem Punkt, der zuletzt vom Snapshot erfasst wurde. [Car+15]

Der Test der Fehlertoleranz bei Flink war nicht erfolgreich. Nach einem TaskManager-Ausfall wird die Verbindung zu Kafka nicht mehr korrekt hergestellt. Die Schnittstelle zu Kafka wird gerade vollständig überarbeitet, daher sollte der Test mit der neuen Anbindung wiederholt werden. [Met15]

Apache Samza

Samza kann den Ausfall des ApplicationMasters und der SamzaContainer tolerieren. Wenn der aktuelle ApplicationMaster ausfällt, werden durch YARN alle zugehörigen SamzaContainer beendet und ein neuer ApplicationMaster gestartet. Dieser startet die Container neu, welche ihrerseits mit der Wiederherstellung beginnen. [Apa15af]

Samza stellt beim Ausfall von Containern eine At-Least-Once-Semantik bereit. Dies wird erreicht, indem Checkpoints erstellt werden. Samza liest die persistierten Tupel von den Kafka-Partitionen. An Checkpoints werden die aktuellen Lesepositionen der Partitionen gesichert und in Kafka abgelegt. Wenn ein Container ausfällt wird vom ApplicationMaster

bei YARN ein neuer Container angefordert und die Tasks dort gestartet. Die Tasks lesen den letzten Checkpoint aus Kafka und beginnen ab der gespeicherten Position mit der Verarbeitung der Partitionen. Die Tupel, die seit dem letzten Checkpoint bis zum Ausfall verarbeitet wurden, werden dabei ein zweites Mal verarbeitet. Dies muss in der Anwendung berücksichtigt werden. Die Benutzung von Checkpoints ist optional, sie können ausgeschaltet beziehungsweise vorhandene Checkpoints ignoriert werden. [Apa15ah]

Die Ergebnisse des praktischen Tests sind in Abbildung 7.28 dargestellt. Beim Ausfall eines Workers dauert es etwa 15 Schritte beziehungsweise 15 Sekunden bis die Latenz wieder das ursprüngliche Niveau erreicht hat. Die verlorenen Tupel werden innerhalb von 18 Sekunden abgearbeitet. Durch Anpassung des Zeitlimits lässt sich die Latenz gegebenenfalls senken.

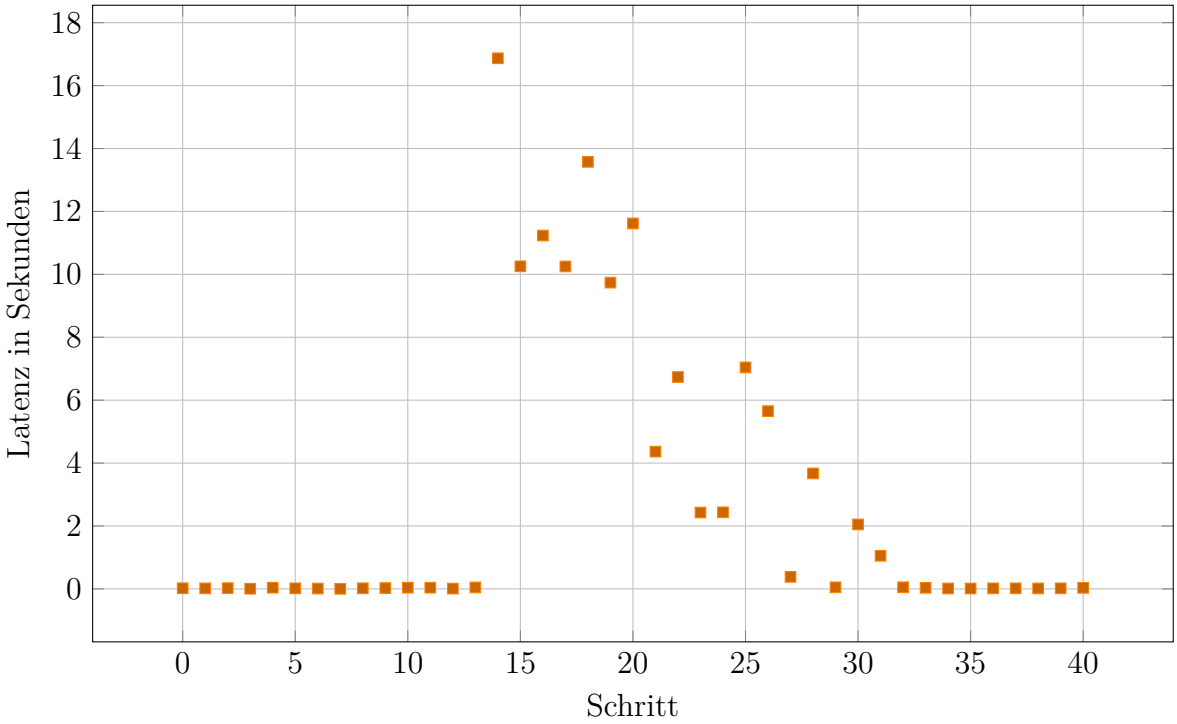


Abbildung 7.28: Fehlertoleranz von Samza

Apache Spark

Spark kann Ausfälle bei den Workern, beim Master und beim Driver kompensieren. Durch den Ausfall des Masters können keine neuen Anwendungen mehr gestartet werden. Laufende Anwendungen sind nicht vom Ausfall des Masters betroffen. Es werden mehrere

Master gestartet und mithilfe von ZooKeeper ein primärer Master gewählt. Beim Ausfall des Anführers übernimmt ein Master in Bereitschaft die Führung. [Apa15be]

Der Driver hält den SparkContext, daher werden bei einem Fehler alle Worker beendet und der im Speicher gehaltene Zustand geht verloren. Wie in Abschnitt 7.4.3 erwähnt, kann er automatisch neu gestartet werden. Durch Checkpoints kann er seinen ursprünglichen Zustand wiederherstellen. Anschließend werden die Worker neu gestartet. Sie durchlaufen ebenfalls die Wiederherstellung aus den Checkpoints. [Apa15bc]

Ausfälle von Workern werden durch eine Kombination von Neuberechnung der RDDs und Checkpoints kompensiert. RDDs haben die Eigenschaft, dass sie sich durch die Quell-RDDs und den Ausführungsplan wiederherstellen lassen. Da bei zustandsbehafteten Operationen viele Quell-RDDs vorgehalten werden müssten, werden Checkpoints eingesetzt, um Zwischenstände zu sichern. Sie werden auf einem fehlertoleranten Speicher, wie HDFS, abgelegt. Ein anderer Worker kann bei einem Ausfall die Verarbeitung übernehmen. Er stellt das RDD aus den Quell-RDDs und gegebenenfalls einem Checkpoint wiederher. [Zah+13]

Die angebotene Fehlertoleranz-Semantik unterscheidet sich je nach Quell- und Zielsystem. Die Verarbeitung selbst hat grundsätzlich eine Exactly-Once-Semantik. Bei Datenquellen, welche die Daten nicht persistieren, können bei einem Ausfall Daten verloren gehen. Datenquellen, welche die Daten persistieren und erst bei Bestätigung des Empfangs löschen, können mithilfe von *Write-Ahead-Logs* die At-Least-Once-Semantik erreichen. Bei der Nutzung von Kafka wird die Exactly-Once-Semantik erreicht. Ausgabeoperationen haben grundsätzlich At-Least-Once-Semantik. Wenn das Ziel idempotente Änderungen oder Transaktionen unterstützt, kann die Exactly-Once-Semantik erreicht werden. [Apa15be]

Abbildung 7.29 stellt die Ergebnisse des Fehlertoleranz-Tests in Spark dar. Vom Ausfall sind die Tupel vom Schritt 15 bis 19 betroffen. Sie haben eine leicht erhöhte Latenz von etwa 180 Millisekunden.

Apache Storm

Storm toleriert den Ausfall des Nimbus, der Supervisor und der Worker. Der Nimbus bildet einen SPOF, da nur eine Instanz davon gleichzeitig gestartet werden kann. Da er nur am Starten der Anwendung beteiligt ist und nicht für aktive Anwendungen gebraucht wird, ist dies aber keine große Einschränkung. Zustandsbehaftete Daten werden im ZooKeeper

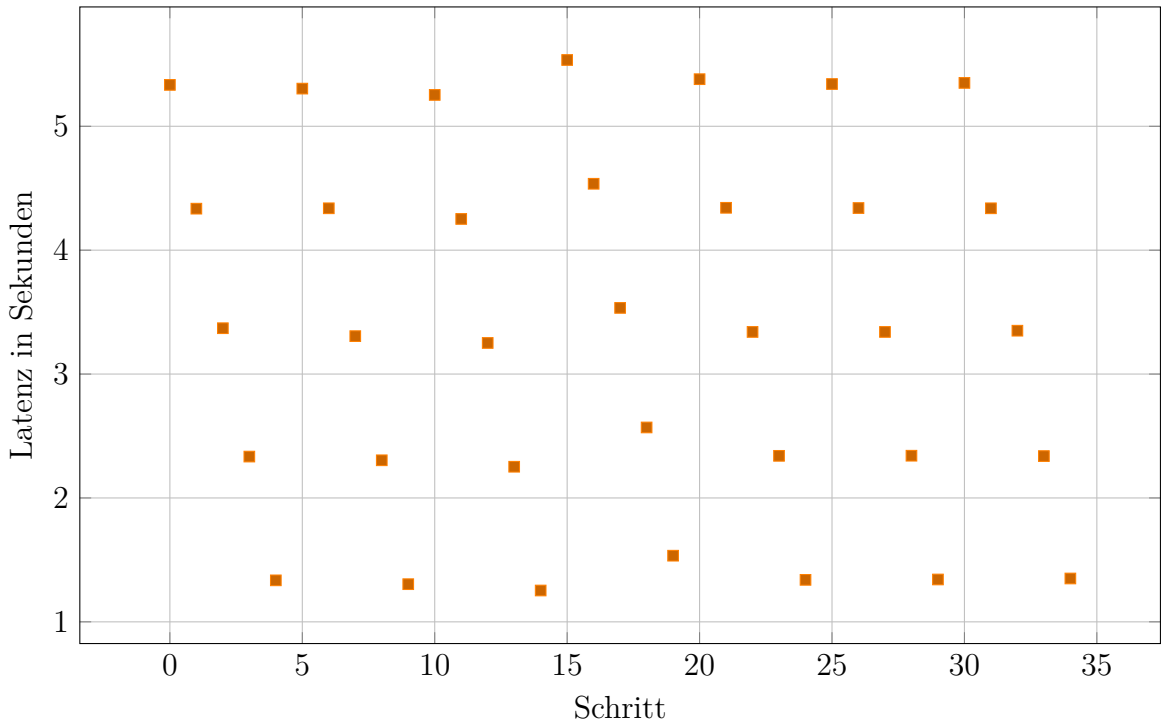


Abbildung 7.29: Fehlertoleranz von Spark

abgelegt. Dadurch kann der Nimbus auf einem anderen Host gestartet werden. Die Supervisor sind für das Starten und Überwachen der Worker zuständig. Worker sind von einem Ausfall des Supervisors nicht betroffen. Nach dem Neustart stellt der Supervisor seinen Zustand aus ZooKeeper wiederher. Wenn ein Worker ausfällt, wird er duch den Supervisor neu gestartet. [Apa14g]

Die vollständige Verarbeitung jedes Tupels wird vom Acker überwacht. Acker werden als Bolts innerhalb der Topologie ausgeführt. Jedes Tupel erhält eine zufällige, 64 Bit lange ID (Identifier). Für jedes von einer Quelle emittierte Tupel wird beim Acker ein Eintrag mit seiner Identifier (ID) und einer Checksumme erfasst. Als Checksumme wird initial die ID benutzt. Jedes Tupel enthält die ID des Tupels, durch das es ursprünglich generiert wurde, im Folgenden Quell-ID gennant. Jeder Bolt, der ein Tupel verarbeitet, bildet die Checksumme durch eine XOR-Verknüpfung der ID des eingehenden und des ausgehenden Tupels und schickt sie zusammen mit der Quell-ID an den Acker. Er bildet ein XOR mit der empfangenen Checksumme und der gespeicherten Checksumme. Wenn die Checksumme auf null steht, wurde die Topologie erfolgreich durchlaufen und der Acker löscht den Eintrag. Andernfalls wird nach einer Zeitüberschreitung das Tupel erneut aus der Quelle geladen. Daher ist Storm auf eine persistente Quelle angewiesen. Storm

verwirklicht damit die At-Least-Once-Semantik. Topologien können auch ohne Acker betrieben werden, dann wird die At-Most-Once-Semantik angeboten. [Apa14h]

Das Ergebnis des Fehlertoleranz-Tests wird in Abbildung 7.30 dargestellt. Die Schritte 14 bis 38 sind hauptsächlich vom Ausfall betroffen. Dies entspricht einer Zeitspanne von 24 Sekunden. Die wiederhergestellten Tupel haben eine Latenz von bis zu 40 Sekunden. Durch Konfigurationsänderungen lässt sich diese Zeitspanne gegebenenfalls senken.

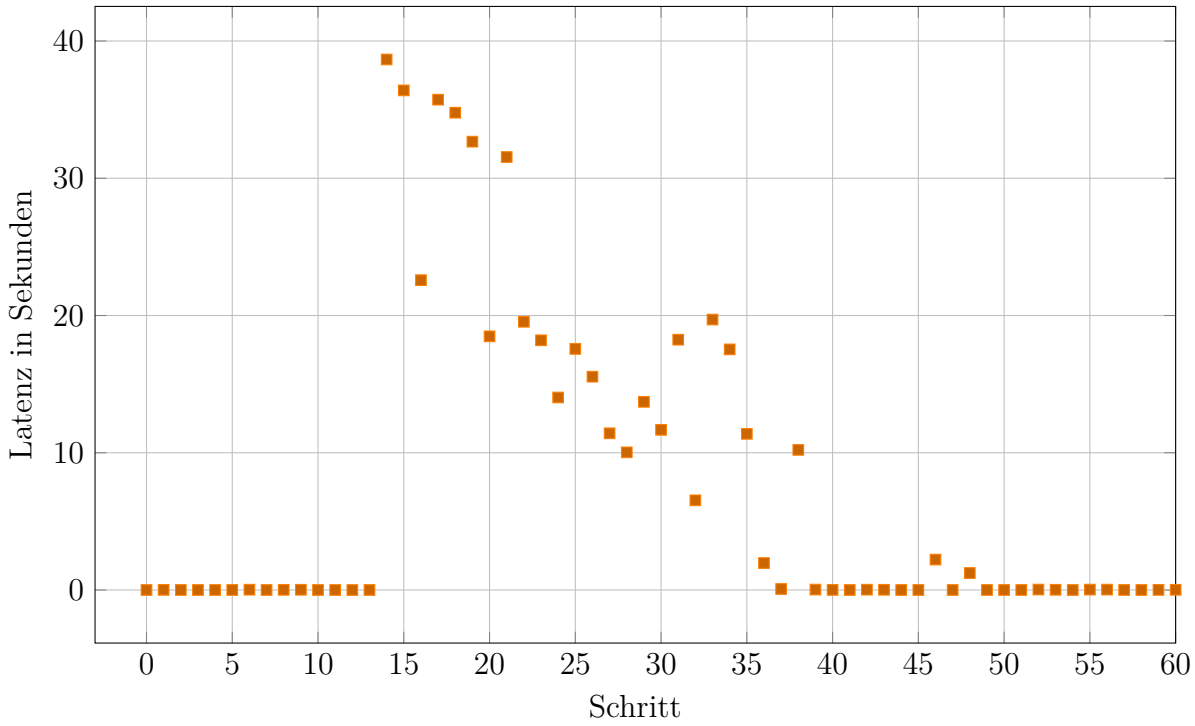


Abbildung 7.30: Fehlertoleranz von Storm

Apache Storm Trident

Storm Trident wird mit den gewöhnlichen Storm-Komponenten ausgeführt, daher findet der erste Absatz des Unterabschnitts von Storm hier auch Anwendung.

Trident bietet drei Modi: einen nicht-transaktionsorientierten, einen transaktionsorientierten und einen opak-transaktionsorientierten Modus. Der erste Modus bietet keinerlei Verarbeitungsgarantien, er ist nur für die Ausführung im Batch-Modus gedacht. Der transaktionsorientierte Modus garantiert, dass ein einmal zusammengestellter Batch mit einer festen Transaktions-ID immer die gleichen Tupel enthält. Das heißt im Fehlerfall wird exakt der gleiche Batch noch einmal abgespielt. Falls dieser Batch nicht

wiederhergestellt werden kann, etwa weil ein Teil des Quellsystems ausfällt, kann die Anwendung nicht weiter arbeiten. Im Gegenzug ist es für die Anwendung einfach damit umzugehen, da eine Transaktions-ID sich immer auf den gleichen Batch bezieht. Der opak-transaktionsorientierte Modus garantiert, dass ein Tupel genau einmal komplett in einem Batch verarbeitet wird. Die Transaktions-ID bezieht sich aber nicht auf eine eindeutige Menge von Tupeln. Das heißt, wenn ein Batch erneut abgespielt werden muss, können neue Tupel hinzu gekommen sein. Daher müssen zustandsorientierte Anwendungen diese Besonderheit beachten. [Apa14q]

Die Abbildung 7.31 stellt das Ergebnis des Fehlertoleranz-Tests dar. Die Schritte 7 bis 73 sind vom Ausfall betroffen. Dabei tritt eine maximale Latenz von über einer Minute auf. Wie bei Storm lässt sich durch eine Konfigurationsänderung die Latenz gegebenenfalls verringern.

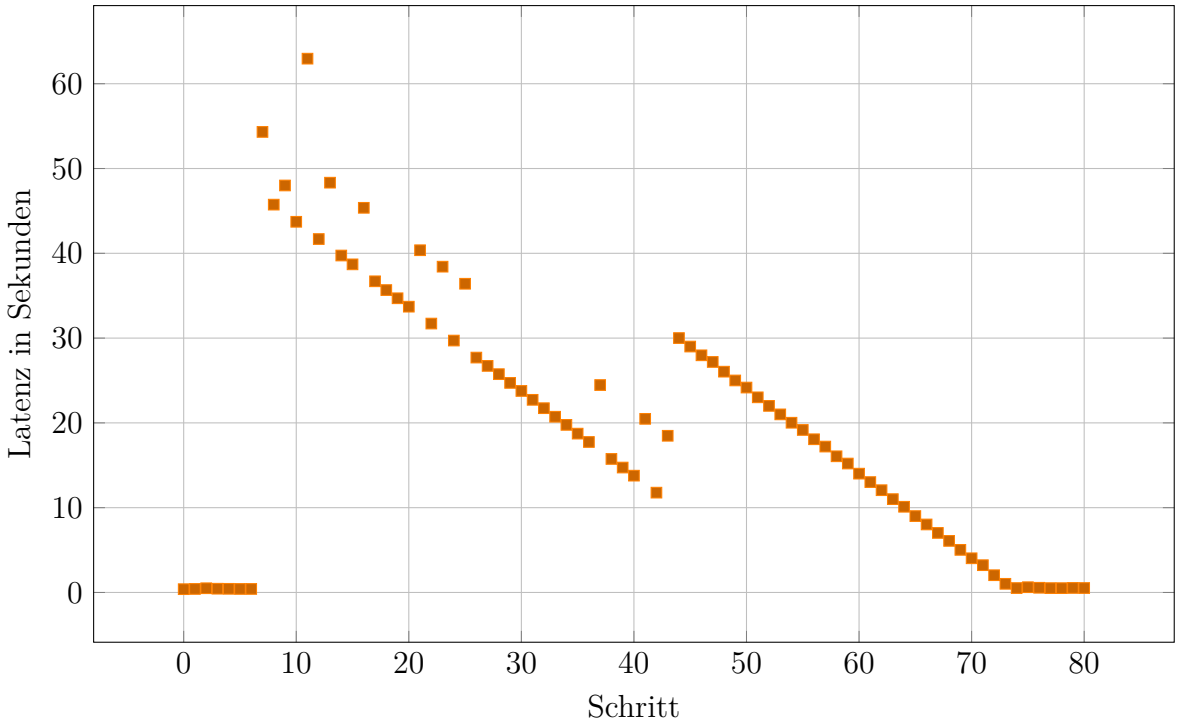


Abbildung 7.31: Fehlertoleranz von Storm Trident

Zusammenfassung

Kriterium	Flink	Samza	Spark	Storm	Storm Trident
Hochverfügbarer Master	-	-	✓	-	-
Verarbeitung unabhängig von Master	-	-	✓	✓	✓
Exactly-Once-Semantik	✓	-	✓	-	✓
At-Least-Once-Semantik	-	✓	✓	✓	-
At-Most-Once-Semantik	-	✓	-	✓	✓
Workerausfall-Test	-	✓	✓	✓	✓

Tabelle 7.16: Verarbeitungsgarantien und Fehlertoleranz

7.6 Security

7.6.1 Nutzerbasierte Zugriffskontrolle

Apache Flink

Flink hat keine Unterstützung für Zugriffskontrollen. Lediglich gesicherte HDFS-Cluster können angesprochen werden. [Apa15p]

Apache Samza

Samza stellt selbst keine Sicherheitsmerkmale zur Verfügung [Apa15as]. Da es auf Kafka und YARN basiert, muss auf ihre Sicherheitsfunktionen zurückgegriffen werden. Kafka unterstützt in der aktuellen Version keine Authentifizierung. Zurzeit wird an einer *Kerberos* [Mas15] und *Simple Authentication and Security Layer (SASL)* [Int06] Implementierung gearbeitet [Kre14]. YARN unterstützt Kerberos zur Authentifizierung und umfangreiche *Access Control Lists (ACLs)* zur Autorisierung. Außerdem kann YARN die Ressourcen zwischen verschiedenen Nutzern aufteilen. [Apa15t]

Apache Spark

Spark kann durch ACLs Berechtigungen zum Starten und Ändern von Anwendungen vergeben. Die Web-GUI benötigt einen *Servlet Filter* [Ora15e], um die Authentifizierung durchzuführen. In der CLI wird der aktuelle Benutzer des Betriebssystems übernommen. Bei der Ausführung auf YARN können Kerberos und alle YARN-spezifischen Sicherheitsfunktionen zur Absicherung und Aufteilung des Clusters auf mehrere Nutzer benutzt werden [Apa15bb]. Im Standalone-Modus lässt sich die Ressourcennutzung nur pro Applikation und nicht pro Benutzer beschränken. [Apa15az]

Apache Storm

Storm unterstützt Kerberos zur Authentifizierung und eine eigene ACL-Implementierung zur Autorisierung. Beides kann über Servlet Filter oder eine eigene Implementierung des *Java Authentication and Authorization Service (JAAS)* [Ora15a] angepasst werden. Durch einen speziellen Scheduler können die Ressourcen zwischen verschiedenen Benutzern aufgeteilt werden. [Apa15bl]

Zusammenfassung

Techniken	Flink	Samza	Spark	Storm
Kerberos	-	(✓)	✓	✓
ACL	-	(✓)	✓	✓
Ressourcenaufteilung	-	(✓)	✓	✓

Tabelle 7.17: Nutzerbasierte Zugriffskontrolle

7.6.2 Verschlüsselung des Datenverkehrs

Apache Flink

Die Flink-Dokumentation stellt keine Informationen zur Verschlüsselung des Datenverkehrs zur Verfügung. Der Bugtracker [Apa15au] oder die Roadmap [TS15] enthalten keine Informationen dazu. Demnach unterstützt Flink keine Verschlüsselung.

Apache Samza

Wie bereits erwähnt, nutzt Samza die Sicherheitsmerkmale der Systeme Kafka und YARN. Kafka unterstützt in der aktuellen Version keine Verschlüsselung des Datenverkehrs. Für die nächste Version sind umfangreiche Sicherheitsmaßnahmen geplant [Kre14]. Hadoop unterstützt aktuell bereits die Verschlüsselung des Datenverkehrs [Apa15t].

Apache Spark

Apache Spark bietet einige Optionen zur Verschlüsselung des Datenverkehrs. Aktuell können nur die Steuerungsverbindungen verschlüsselt werden, aber nicht die Daten und die Web-GUI. Diese Funktionen werden aktuell erst implementiert [Apa15bf]. Worker können sich gegenüber anderen Workern und dem Master authentifizieren, sodass sich kein fremder Worker anmelden kann. [Apa15bb]

Apache Storm

Apache Storm kann seine Datenströme teilweise verschlüsseln [Apa15bl]. Die Verschlüsselung der Tupel ist durch eine spezielle Serialisierungsklasse möglich [Apa13b]. Der Datenstrom an sich bleibt unverschlüsselt. Die Web-GUI und die Schnittstelle für verteilte Methodenaufrufe lassen sich durch *Transport Layer Security (TLS)* absichern.

Zusammenfassung

Verbindung	Flink	Samza	Spark	Storm
Tupel	-	-	-	✓
Steuerung	-	✓	✓	-
Web-GUI	-	-	-	✓

Tabelle 7.18: Verschlüsselung des Datenverkehrs

7.7 Maintainability

7.7.1 Aktualisieren des Frameworks

Apache Flink

Die Flink-Dokumentation liefert keine speziellen Anweisungen zum Aktualisieren des Frameworks und zum Schema der Versionsnummer. Daher wird angenommen, dass, insbesondere im Standalone-Modus, der Cluster während der Aktualisierung angehalten und neu gestartet werden muss. Je nach Verfügbarkeit der Ressourcen können mehrere Cluster parallel bereitgestellt werden. Die Versionsnummer 0.x signalisiert im Allgemeinen, dass sich die API jederzeit ändern kann und daher Anpassungen an der Anwendung erforderlich werden [Pre14]. Auch dazu gibt es in der Dokumentation keine Aussage.

Apache Samza

Da Samza mit jeder Anwendung komplett ausgeliefert und getrennt auf YARN bereitgestellt wird, können gleichzeitig verschiedene Versionen genutzt werden. Samza benutzt *Semantic Versioning* [Pre14], daher gibt es nur inkompatible Änderungen der API, wenn sich die Hauptversion ändert oder die Hauptversion, wie aktuell, 0 ist [Apa15ai]. Kafka kann innerhalb der Hauptversionen während dem laufenden Betrieb Broker für Broker aktualisiert werden [Apa15x]. Hadoop YARN kann ebenfalls, die entsprechende hochverfügbare Konfiguration vorausgesetzt, im laufenden Betrieb aktualisiert werden [Set13].

Apache Spark

Spark nutzt Semantic Versioning, daher bleibt die API innerhalb einer Hauptversion stabil. Eine Aktualisierung ohne Unterbrechung ist nicht dokumentiert. Ähnlich wie bei Flink können mehrere Versionen parallel betrieben werden. [Apa14a]

Apache Storm

Beginnend mit Version 0.10 unterstützt Storm unterbrechungsfreie Aktualisierungen. Die einzelnen Komponenten können nacheinander gestoppt, aktualisiert und wieder gestartet werden. Die Ausfälle werden durch die üblichen Fehlertoleranzmechanismen ausgeglichen.

Die Dokumentation von Storm enthält keine Aussage zur Semantik der Versionsnummer. [Goe15]

Zusammenfassung

Kriterium	Flink	Samza	Spark	Storm
Feste Versionssemantik	-	✓	✓	-
Unterbrechungsfreie Aktualisierung	-	✓	-	✓

Tabelle 7.19: Aktualisieren des Frameworks

7.7.2 Erfassung von Metriken

Apache Flink

Flink unterstützt im Streaming-Modus bisher keine anwendungsbezogenen Metriken. Lediglich die Ressourcenauslastung kann in der Web-GUI verfolgt werden. Im Batch-Modus werden Zähler und Histogramme unterstützt. Diese Unterstützung wird in der nächsten Version in den Streaming-Modus übernommen. [Ewe15]

Apache Samza

Samza hat eine integrierte Komponente zur Generierung von Metriken. Sie stellt standardmäßig Metriken zum aktuellen Systemzustand zur Verfügung. Zusätzlich können anwendungsspezifische Metriken mithilfe von Zählern und Zustandsanzeigen implementiert werden. Einstellungen der Metriken werden in der Task-Konfiguration angepasst. Sie können im JSON-Format per Kafka veröffentlicht oder über *Java Management Extensions (JMX)* abgerufen werden. Ein Metrik-Eintrag ist in Quelltext 7.1 zu sehen. [Apa15ao]

Apache Spark

Spark nutzt die Bibliothek Metrics [Hal14a] zur Erfassung von Metriken. Es stellt selbst umfangreiche Metriken, wie Job-Statistiken, Latenz und Durchsatz der Anwendung, bereit. Sie werden über eine Web-GUI angezeigt. Ein Auszug davon ist in Abbildung 7.32

```
{
  "header": {
    "job-id": "1",
    "job-name": "statistics",
    "host": "Lannisport.fbi.h-da.de",
    "container-name": "samza-container-1",
    "source": "TaskName-Partition 12",
    "time": 1439495911523,
    // ...
  },
  "metrics": {
    "org.apache.samza.container.TaskInstanceMetrics": {
      "commit-calls": 2,
      "window-calls": 113,
      "flush-calls": 2,
      "send-calls": 2934,
      "process-calls": 824,
      "messages-sent": 2934,
      "kafka-quote-12-offset": "823"
    },
    "de.infomotion.skern.nyse.streaming.samza.task.StatisticsTask": {
      "quote-count": 824
    },
    // ...
  }
}
```

Quelltext 7.1: Auszug aus einem Samza-Metrik-Eintrag mit anwendungsspezifischer Metrik

dargestellt. Durch die Bibliothek lassen sich anwendungsspezifische Metriken, wie Zähler, Zustandsanzeigen, Histogramme, Zähler mit Zeitbezug und Zeitmessung einer Aktion, erfassen [Hal14b]. Die Metriken sind über verschiedene, konfigurierbare Wege zugänglich, etwa per Logdatei, REST-Schnittstelle, JMX oder einer Log-Bibliothek. [Apa15ba]

Apache Storm

Storm bietet eine integrierte Unterstützung von Metriken. Umfangreiche Statistiken zu laufenden Anwendungen können über die in Abbildung 7.33 dargestellte Web-GUI abgerufen werden. Da Storm Trident mehrere Operationen in einen Bolt zusammenfasst, sind dessen Statistiken nicht so detailliert, wie die der gewöhnlichen API. Für die Integration in Anwendungen stehen Zähler, Zustandsanzeigen und aggregationsbasierte

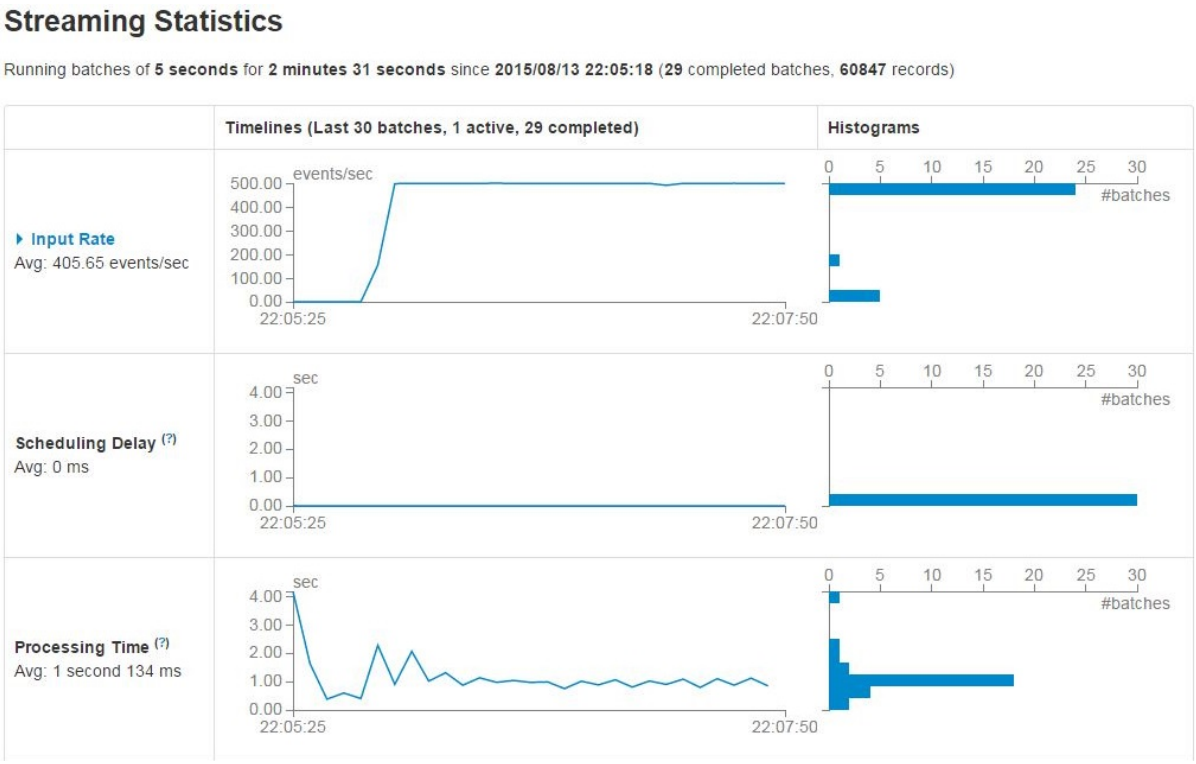
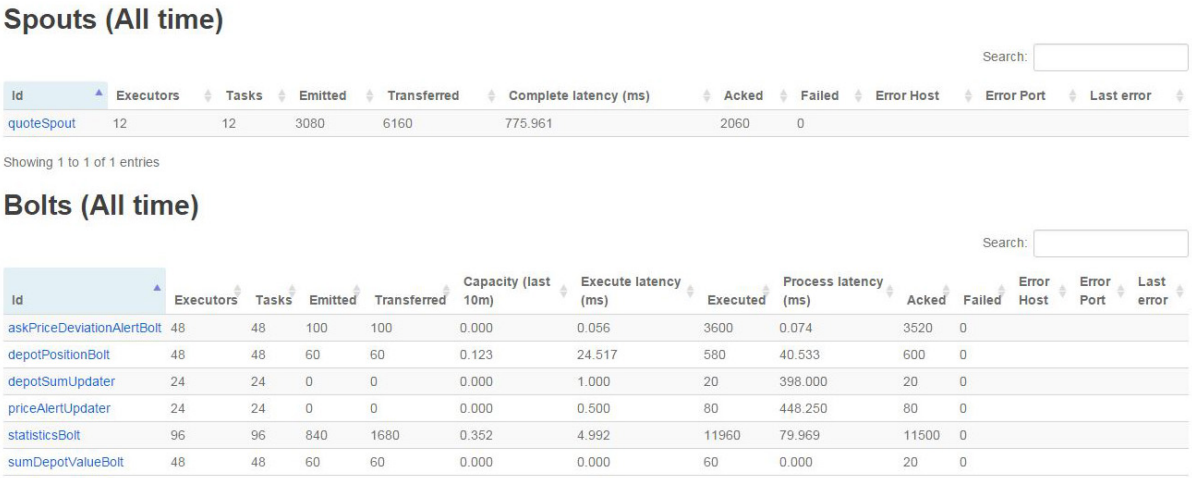


Abbildung 7.32: Auszug aus der Statistik auf der Spark Web-GUI

Metriken bereit, die nach dem Reduce- oder Fold-Prinzip arbeiten [Apa14n]. Die offizielle Dokumentation ist lückenhaft, was die Ausgabe der Metriken betrifft. Es gibt eine `LoggingMetricsConsumer`-Klasse, die als Bolt in die Topologie eingebunden wird und die Metriken in einer Logdatei speichert. [Tro14]



Spouts (All time)

Search:

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Error Host	Error Port	Last error
quoteSpout	12	12	3080	6160	775.961	2060	0			

Showing 1 to 1 of 1 entries

Bolts (All time)

Search:

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Error Host	Error Port	Last error
askPriceDeviationAlertBolt	48	48	100	100	0.000	0.056	3600	0.074	3520	0			
depotPositionBolt	48	48	60	60	0.123	24.517	580	40.533	600	0			
depotSumUpdater	24	24	0	0	0.000	1.000	20	398.000	20	0			
priceAlertUpdater	24	24	0	0	0.000	0.500	80	448.250	80	0			
statisticsBolt	96	96	840	1680	0.352	4.992	11960	79.969	11500	0			
sumDepotValueBolt	48	48	60	60	0.000	0.000	60	0.000	20	0			

Abbildung 7.33: Auszug aus der Statistik auf der Storm Web-GUI

Zusammenfassung

Funktion	Flink	Samza	Spark	Storm
Integrierte Statistiken	-	✓	✓	✓
Ausgabe in Web-GUI	-	-	✓	✓
Zähler	-	✓	✓	✓
Zustandsanzeige	-	✓	✓	✓
Weitere	-	-	Histogramme, Zähler mit Zeitbezug, Zeitmessung	Aggregation per Reduce oder Fold
Ausgabe	-	Kafka, JMX	Log-Bibliothek, CSV, JMX, REST	Logdatei

Tabelle 7.20: Anzeige von Metriken

7.7.3 Aussagekräftige Logdaten

Apache Flink

Flink nutzt Simple Logging Facade for Java (SLF4J) [QOS15b] als Logging-Schnittstelle. Standardmäßig wird log4j [Apa12b] als konkrete Implementierung verwendet. Alternativ lässt sich auch logback [QOS15a] verwenden. Für Anpassungen stehen bei beiden Implementierungen entsprechende Konfigurationsdateien bereit.

Jede Komponente, wie JobManager, TaskManager, Web-GUI und CLI erstellt eine eigene Logdatei, die im Dateinamen den Hostnamen des Rechenknotens enthält. Zunächst werden in den jeweiligen Logdateien Angaben zur Konfiguration angegeben. Insbesondere beim Job- und TaskManager werden Ereignisse, wie das Starten und Beenden von Anwendungen und deren Operatoren sowie Checkpoints, festgehalten. Anwendungsspezifische Meldungen sind in den Logdateien der TaskManager enthalten. Sicherheitsrelevante Vorgänge werden nicht erfasst. Die Dokumentation von Flink zeigt in einem kurzen Beispiel die Verwendung von Logging. Die Logdateien sind auch über die Web-GUI von Flink zugänglich. [Apa15r]

Apache Samza

Samza verwendet ebenfalls SLF4J als Logging-Schnittstelle. Als Implementierung wird log4j in der Anleitung vorgestellt. Eine vorgefertigte Konfigurationsdatei liegt Samza bei. Durch JMX kann während der Laufzeit das Log-Level angepasst werden.

Samza erstellt Logdateien für den ApplicationMaster und die SamzaContainer. Wenn Samza auf YARN ausgeführt wird, werden die Logdateien verstreut in verschiedene Verzeichnisse, aufgeteilt nach Container, abgelegt. YARN beherrscht zwar die Aggregation von Logdateien, aber erst wenn die Anwendung beendet wurde, was bei Streaming-Anwendungen eher selten geschieht. Dadurch ist die Auswertung der Logdateien aufwendig. Um diesen Nachteil auszugleichen, unterstützt Samza auch das Senden der Logdaten an eine Kafka Topic. Die Logdatei des ApplicationMasters enthält die Abbildung der Partitionen auf Tasks und Details zur Ressourcenzuteilung durch YARN. Durch den SamzaContainer werden Informationen, die Checkpoints oder die Kommunikation mit Kafka betreffen, im Log gespeichert. Zusätzlich werden dort die Logdaten der Tasks gespeichert. Da Samza und YARN auf Kafka basieren, müssen deren Logdateien gegebenenfalls untersucht werden, etwa bei sicherheitsrelevanten Vorgängen. [Apa15an]

Apache Spark

Spark nutzt ebenfalls SLF4J als Logging-Schnittstelle und log4j als Logging-Bibliothek. Eine Beispielkonfiguration liegt Spark bei.

Standardmäßig legen der Master und die Worker Logdateien an, die sich hauptsächlich auf den Status des Prozesses und das Starten und Stoppen von Anwendungen beziehen. In ein separates Verzeichnis werden pro Anwendung und Worker Logdaten über die Verarbeitung, wie Task-Informationen und RDD-Bewegungen, und Logdaten der Anwendung gespeichert. Der Driver protokolliert, welche Worker beteiligt sind und welcher Benutzer die Anwendung gestartet hat. Zusätzlich kann ein Ereignisprotokoll erstellt werden, damit die in der Web-GUI angezeigten Daten auch nach dem Beenden der Anwendung verfügbar sind. [Apa15ba; Apa15ay]

Apache Storm

Storm nutzt SLF4J mit log4j2 als Logging-Bibliothek. Die Dokumentation von Storm enthält zum Thema Logging nur wenige, veraltete Informationen [Apa14f]. Daher wurden die Informationen hauptsächlich bei der Implementierung der Aufgaben gesammelt.

Storm legt Logdateien getrennt nach Komponenten an. Die Logdatei des Nimbus enthält Informationen über die Zuordnung der Executors. Die Supervisor-Logdaten geben Auskunft über gestartete Worker. Beide Logdateien sind eher unübersichtlich und enthalten nur wenige relevante Informationen. Wenn Sicherheitsfunktionen aktiviert sind, werden sicherheitsrelevante Daten in einer separaten Datei protokolliert. Die Logdateien der Worker werden mit dem entsprechenden Namen der Anwendung, einem Zeitstempel und dem Worker-Port benannt. Sie enthalten Informationen zu den gestarteten Tasks und die anwendungsspezifischen Logdaten.

Zusammenfassung

Funktion	Flink	Samza	Spark	Storm
Logging-Bibliothek	SLF4J + log4j/logback	SLF4J + log4j	SLF4J + log4j	SLF4J + log4j2
Trennung nach Anwendung	-	✓	✓	✓
Sicherheitsrelevante Daten	-	(✓)	✓	✓

Tabelle 7.21: Aussagekräftige Logdaten

7.7.4 Testen auf dem Entwicklersystem

Apache Flink

Da alle benutzerdefinierten Funktionen nur die vom Benutzer angegebenen Typen und eventuell das `Iterator<T>`- oder `Collector<T>`-Interface nutzen, können sie direkt in Unit-Tests benutzt werden. Dafür verfügt das `Collector<T>`-Interface über eine einfache Implementierung, welche die Ergebnisse in einer Liste speichert. Für die Ausführung des gesamten Frameworks in der Entwicklungsumgebung steht die lokale Ausführungsumgebung zur Verfügung. Durch eine Methode wird sie immer dann automatisch genutzt, wenn die Anwendung nicht in einer Cluster-Umgebung ausgeführt wird. Alle Komponenten werden damit innerhalb eines Prozesses ausgeführt und die Parallelität durch Threads erreicht. Die lokale Umgebung kann, wie jede Anwendung, debuggt werden. Alternativ kann nach Einrichtung der Java-Remote-Debugger genutzt werden. Um Tests ohne echte Datenquellen durchzuführen, können Listen mit vorbereiteten Elementen eine Datenquelle nachahmen. [Apa15f; Apa15l]

Apache Samza

Einzelne Tasks in Samza können nicht direkt mit einem Unit-Test erfasst werden, da die Methoden Samza-spezifische Objekte erwarten. Es ist daher einfacher, die Klasse so zu gestalten, dass die eigenen Methoden direkt für Tests zugänglich sind. Die lokale Ausführung kann durch eine Konfigurationsänderung eingestellt werden. Dabei wird die Parallelität entweder durch Prozesse oder Threads erreicht. In der Entwicklungsumgebung

muss für die Ausführung von Samza eine extra Ausführungs-Konfiguration angelegt werden. Samza liegt ein Tutorial für Remote-Debugging bei. [Apa15aq; Apa15ak]

Apache Spark

Die meisten Funktionen in Spark können direkt in Unit-Tests verwendet werden, da sie nur die vom Benutzer angegebenen Typen enthalten. Spark kann einen lokalen Master nutzen, um eine lokale Ausführungsumgebung zu erstellen. Die Dokumentation weist ausführlich darauf hin, dass sich der Gültigkeitsbereich von Variablen beim lokalen Modus von dem beim Cluster-Modus unterscheidet und daher ein Programm, das lokal funktioniert, auf dem Cluster fehlerhaft sein kann. Im lokalen Modus kann Spark innerhalb der Entwicklungsumgebung debuggt werden. Remote-Debugging ist ebenfalls möglich. Für Testzwecke kann ein Datenstrom durch eine Warteschlange aus RDDs simuliert werden. [Apa15be]

Apache Storm

Bolts können in Storm nicht direkt in einem Unit-Test verwendet werden, denn die Ergebnis-Tupel werden über ein Storm-spezifisches Objekt erfasst. Ganze Topologien können mit einem lokalen Cluster getestet werden. Über Konfigurationsoptionen kann die maximale Anzahl der parallelen Tasks eingestellt werden. Optional werden alle emittierten Tupel in einer Logdatei protokolliert. Der lokale Cluster kann innerhalb der Entwicklungsumgebung zum Debuggen genutzt werden. Remote-Debugging ist ebenfalls möglich. Für Tests ohne Datenquelle kann ein spezieller Spout verwendet werden, der eine feste Liste an Tupeln abspielt. [Apa14i]

Zusammenfassung

Funktion	Flink	Samza	Spark	Storm
Direkte Unit-Tests	✓	-	(✓)	-
Lokale Clusterausführung	✓	✓	✓	✓
Remote-Debugging	✓	✓	✓	✓
Abspielen von Test-Tupeln	✓	-	✓	✓

Tabelle 7.22: Testen auf dem Entwicklersystem

7.8 Assigned Property

7.8.1 Verfügbarkeit externer Unterstützung

Einige Entwickler von Flink haben zwar das Unternehmen Data Artisans [Dat15a] gegründet, ein Angebot für die kommerzielle Unterstützung ist aber nicht ersichtlich. Für Samza wird keine kommerzielle Unterstützung angeboten. Unterstützung für Apache Spark wird durch Databricks, das von Entwicklern von Spark gegründet wurde, angeboten [Dat15c]. Außerdem ist Spark in vielen Distributionen vertreten, wie Cloudera [Clo15], Hortonworks [Hor15], MapR [Map15] und weitere [Dat15d]. Für Apache Storm wird keine kommerzielle Unterstützung angeboten, es wird aber bei den Distributionen von Hortonworks und MapR mitgeliefert.

Die Ergebnisse der Internetrecherche werden in Tabelle 7.23 dargestellt. Die verwendeten Suchbegriffe sind in Tabelle 7.24 angegeben. Wie bei dem in Abschnitt 7.5.1 dargestellten Kriterium, werden die Kennzahlen von Flink und Spark einmal insgesamt und einmal für die Spezialisierung auf Streaming angegeben. Die Kennzahlen wurden am 25. August 2015 erhoben.

Kennzahl	Flink	Flink Streaming	Samza	Spark	Spark Streaming	Storm
Google.com	62.800	14.800	18.700	546.000	196.000	204.000
Amazon.com	3	0	4	62	5	24
Stackoverflow.com	56	11	21	2.222	186	564

Tabelle 7.23: Verfügbarkeit externer Unterstützung

Framework	Suchbegriff
Apache Flink	„Apache Flink“
Apache Flink Streaming	„Apache Flink“ streaming
Apache Samza	„Apache Samza“
Apache Spark	„Apache Spark“
Apache Spark Streaming	„Apache Spark“ streaming
Apache Storm	„Apache Storm“

Tabelle 7.24: Verwendete Suchbegriffe für Tabelle 7.23

8 Zusammenfassung

Ausgangspunkt der Arbeit war die Aufgabenstellung, die Qualität von aktuellen Open Source Stream Processing Frameworks zu evaluieren. Zunächst wurden das Verarbeitungsmodell und die Anforderungen an Stream Processing Frameworks dargestellt (siehe Kapitel 4). Insbesondere wurde eine Referenzarchitektur (siehe Unterkapitel 4.3) vorgestellt, um die verschiedenen Komponenten eines Frameworks übergreifend zuordnen zu können. Anschließend wurden die Frameworks ausgewählt, die evaluiert werden (siehe Unterkapitel 5.2). Danach wurden die für die Evaluierung notwendigen Methoden (siehe Unterkapitel 5.3) zur Informationsgewinnung und Auswahl der Vergleichskriterien erarbeitet. Sie wurden genutzt, um die Kriterien (siehe Unterkapitel 5.4) für die Evaluierung gemäß dem in ISO 25010 dargestellten Qualitätsmodell zu entwickeln. Anschließend wurden praktische Aufgaben für die Durchführung des Cognitive Walkthrough und der Leistungsmessung erstellt (siehe Unterkapitel 5.5). Diese wurden im Rahmen der praktischen Umsetzung (siehe Kapitel 6) realisiert, wodurch der Praxisbezug der Arbeit intensiviert werden konnte. Abschließend wurden die Ergebnisse (siehe Kapitel 7) für die einzelnen Kriterien ausgeführt und in Tabellen zusammengefasst dargestellt.

Die folgenden Unterkapitel erläutern die Resultate dieser Arbeit und geben einen Anstoß für weitere Untersuchungen.

8.1 Ergebnis der Evaluation

Es wurde eine große Bandbreite von Kriterien untersucht. Der Vergleich hat gezeigt, dass keines der evaluierten Frameworks alle Kriterien vollständig abdeckt. Vielmehr ist die Wahl des Frameworks von den konkreten Anforderungen einer Anwendung abhängig. Die Ergebnisse werden für jedes Framework im Folgenden kurz zusammengefasst.

Apache Flink weist eine gute und mächtige API und eine hohe Leistungsfähigkeit auf. Allerdings hat Flink noch einen geringen Reifegrad, der sich durch die fehlende Unterstützung von Sicherheitsfunktionen, Metriken, externer Unterstützung und insbesondere der mangelhaften Fehlertoleranzfunktion ausdrückt. Da die Entwicklung von einer großen Community vorangetrieben wird, werden diese Bereiche in Zukunft wahrscheinlich weiter verbessert.

Apache Samza hat eine übersichtliche API und eine gute Dokumentation. Durch die Architektur von Samza hängt der Betrieb hauptsächlich von Kafka und Hadoop YARN ab. Die mäßige Leistungsfähigkeit, kleine Community und mangelnde externe Unterstützung schränken die praktische Relevanz allerdings ein.

Apache Spark hat eine umfangreiche API, eine große Community und viel externe Unterstützung. Außerdem ist der Reifegrad im Bezug auf betriebs- und entwicklungsrelevante Aspekte hoch. Prinzipbedingt ist das Framework aufgrund seiner hohen Latenz für sehr zeitkritische Anwendungen ungeeignet.

Apache Storm hat eine sehr geringe Latenz und einen hohen Durchsatz. Es hat eine große Community und wird stetig erweitert, insbesondere in betriebsrelevanten Aspekten. Die API und die Dokumentation sind nur von mäßiger Qualität, außerdem fehlt die Unterstützung für zustandsbehaftete Operationen. Storm Trident hat im Gegensatz zu Flink und Spark eine deutlich schlichtere API und kompliziertere zustandsbehaftete Operationen.

8.2 Diskussion des Konzepts

Neben dem Vergleich der vier konkreten Frameworks liefert diese Arbeit die Basis, um in Zukunft weitere Frameworks zu untersuchen. Ein Bestandteil ist die in Unterkapitel 4.3 erarbeitete Referenzarchitektur, in die die Komponenten der Frameworks eingeordnet werden können, um einen besseren Überblick zu gewinnen.

Die Orientierung am Qualitätsmodell von ISO 25010 sorgt für nachvollziehbare und umfassende Kriterien. Sie lassen sich zusammen mit den hier vorgestellten Methoden für weitere Frameworks oder neue Versionen wiederverwenden. Für die Übertragung der Ergebnisse auf konkrete Anwendungen, wie etwa die Empfehlung eines Frameworks für ein Entwicklungsprojekt, müssen lediglich die Ergebnisse entsprechend den Anforderungen gewichtet werden.

Die genutzten Methoden zur Erfassung der Benutzerfreundlichkeit, wie Cognitive Walkthrough und Cognitive Dimensions, sorgen im Rahmen der beschränkten Ressourcen dieser Arbeit für eine objektive Betrachtung. Bei mehr verfügbaren Ressourcen kann man, insbesondere in diesem Bereich, auf noch objektivere Prozesse zurückgreifen, etwa klassische Usability-Tests mit mehreren Testpersonen.

Da in dieser Arbeit die umfassende Evaluierung der Frameworks im Vordergrund stand, wurde nur ein Anwendungsfall zur Leistungsmessung herangezogen. Zur Objektivierung dieses Kriterium sind gegebenenfalls noch weitere Messungen mit anders gearteten Aufgabenstellungen durchzuführen.

8.3 Ausblick

Diese Arbeit betrachtet mit dem Vergleich von Stream Processing Frameworks nur die Grundlage, die zur Verarbeitung von Datenströmen nötig ist. Sie bildet den Ausgangspunkt für eine Reihe weiterer Untersuchungen.

Bisher ist die effiziente Integration von Batch- und Stream-Verarbeitung noch nicht realisiert. Zwar können mit Spark und Flink sowohl Batch- als auch Streaming-Anwendungen entwickelt werden, sie sind aber trotzdem noch weitgehend getrennt. Daher ist es eine offene Fragestellung, welche Hilfsmittel nötig sind, um die beiden Techniken effizient miteinander zu verbinden. Durch Marz und Warren wurde die *Lambda-Architektur* [MW15] vorgestellt, die beide Domänen vereinigen soll. Bisher müssen Anwendungen auf deren Basis allerdings spezifisch dafür entwickelt werden. Ein weiterer Ansatz ist das Projekt Summingbird [Twi15], das Algorithmen sowohl in einer Batch- als auch in einer Streaming-Umgebung ausführen kann.

Abseits der Architektur sind auch die verwendeten Algorithmen Gegenstand aktueller Forschung. Herausforderungen dabei sind die geringe Latenz der Verarbeitung sicherzustellen und das hohe Volumen der ankommenden Daten zu bewältigen. Ein Ansatz dafür ist die Verwendung von approximativen Algorithmen [Lal+06], der allerdings nicht bei jedem Anwendungsfall infrage kommt. Ein Weiterer ist die Nutzung von Qualitätssicherungsmaßnahmen, um im Falle der Überlastung eine definierte Beschaffenheit der Ergebnisse zu erhalten [Cha09]. Die aufgezeigten Fragestellungen stehen stellvertretend für das Potential, dass das Thema Stream Processing für weitere Forschungsarbeiten bietet.

Abkürzungsverzeichnis

ACL Access Control List . . . 114
ANSI American National Standards Institute . . . 53
API Application Programming Interface . . . 19
CEP Complex Event Processing . . . 9
CLI Command Line Interface . . . 21
CQL Continous Query Language . . . 11
CSV Comma-Separated Values . . . 121
DAG Directed Acyclic Graph . . . 13
DBMS Database Management System . . . 10
DSMS Data Stream Management System . . . 10
EPL Esper Processing Language . . . 9
ETL Extract, Transform, Load . . . 10
GUI Graphical User Interface . . . 21
HDFS Hadoop Distributed File System . . . 22
ID Identifier . . . 111
ISO Internationale Organisation für Normung . . . 6
JAAS Java Authentication and Authorization Service . . . 115
JAR Java Archive . . . 56
JDBC Java Database Connectivity . . . 94
JMX Java Management Extensions . . . 118
JSON JavaScript Object Notation . . . 74
NYSE New York Stock Exchange . . . 42
RDD Resilient Distributed Dataset . . . 27
REST Representational State Transfer . . . 119
SASL Simple Authentication and Security Layer . . . 114
SLF4J Simple Logging Facade for Java . . . 122
SPOF Single Point of Failure . . . 107
SQL Structured Query Language . . . 8
SQuaRE Software product Quality Requirements and Evaluation . . . 6
ssh Secure Shell . . . 46
TLS Transport Layer Security . . . 116

Literaturverzeichnis

[Aab15] Lars Aaberg. *Sql2o: Easy database query library*. 2015. URL: http://www.sql2o.org/ (besucht am 15.09.2015).

[Aba+03] Daniel J. Abadi u. a. „Aurora: A new model and architecture for data stream management“. In: *The VLDB Journal The International Journal on Very Large Data Bases* 12.2 (2003), S. 120–139.

[ASK07] Aditya Agarwal, Mark Slee und Marc Kwiatkowski. *Thrift: Scalable Cross-Language Services Implementation*. 2007.

[Age15] Agendaless Consulting Inc., Hrsg. *Supervisor: A Process Control System*. 2015. URL: http://supervisord.org (besucht am 15.09.2015).

[Ale15] Alexander Alexandrov. *Add support for building Flink with Scala 2.11*. 2015. URL: https://github.com/apache/flink/pull/477 (besucht am 15.09.2015).

[Ale+14] Alexander Alexandrov u. a. „The Stratosphere platform for big data analytics“. In: *The VLDB Journal* 23.6 (2014), S. 939–964.

[Alm12] Dion Almaer. *MUPD8*. 2012. URL: https://github.com/walmartlabs/mupd8 (besucht am 15.09.2015).

[Ama15a] Amazon Web Services Inc., Hrsg. *Amazon AWS | Amazon Elastic Compute Cloud (EC2)*. 2015. URL: https://aws.amazon.com/ec2 (besucht am 15.09.2015).

[Ama15b] Amazon Web Services Inc., Hrsg. *Amazon Simple Storage Service S3*. 2015. URL: https://aws.amazon.com/s3 (besucht am 15.09.2015).

[Ama15c] Amazon Web Services Inc., Hrsg. *AWS Real Time Analytics & Echtzeitdaten | Amazon Kinesis*. 2015. URL: https://aws.amazon.com/kinesis (besucht am 15.09.2015).

[AGT14] Henrique C. M. Andrade, Buğra Gedik und Deepak S. Turaga. *Fundamentals of stream processing: Application design, systems, and analytics*. Cambridge, England: Cambridge University Press, 2014.

[Apa15a] Apache Commons, Hrsg. *Math - Commons Math: The Apache Commons Mathematics Library*. 2015. URL: https://commons.apache.org/proper/commons-math/ (besucht am 15.09.2015).

[Apa15b] Apache Flink, Hrsg. *Apache Flink 0.10-SNAPSHOT Documentation: JobManager High Availability (HA)*. 2015. URL: https://ci.apache.org/projects/flink/flink-docs-master/setup/jobmanager_high_availability.html (besucht am 15.09.2015).

[Apa15c] Apache Flink, Hrsg. *Apache Flink 0.9.0 Documentation: Cluster Execution*. 2015. URL: https://ci.apache.org/projects/flink/flink-docs-release-0.9/apis/cluster_execution.html (besucht am 15.09.2015).

[Apa15d] Apache Flink, Hrsg. *Apache Flink 0.9.0 Documentation: Cluster Setup: Flink Setup*. 2015. URL: https://ci.apache.org/projects/flink/flink-docs-release-0.9/setup/cluster_setup.html#flink-setup (besucht am 15.09.2015).

[Apa15e] Apache Flink, Hrsg. *Apache Flink 0.9.0 Documentation: Command-Line Interface*. 2015. URL: https://ci.apache.org/projects/flink/flink-docs-release-0.9/apis/cli.html (besucht am 15.09.2015).

[Apa15f] Apache Flink, Hrsg. *Apache Flink 0.9.0 Documentation: Flink Programming Guide*. 2015. URL: https://ci.apache.org/projects/flink/flink-docs-release-0.9/apis/programming_guide.html.

[Apa15g] Apache Flink, Hrsg. *Apache Flink 0.9.0 Documentation: Flink Stream Processing API*. 2015. URL: https://ci.apache.org/projects/flink/flink-docs-release-0.9/apis/streaming_guide.html (besucht am 15.09.2015).

[Apa15h] Apache Flink, Hrsg. *Apache Flink 0.9.0 Documentation: General Architecture and Process Model*. 2015. URL: https://ci.apache.org/projects/flink/flink-docs-release-0.9/internals/general_arch.html (besucht am 15.09.2015).

[Apa15i] Apache Flink, Hrsg. *Apache Flink 0.9.0 Documentation: Google Compute Engine Setup*. 2015. URL: https://ci.apache.org/projects/flink/flink-docs-release-0.9/setup/gce_setup.html (besucht am 15.09.2015).

[Apa15j] Apache Flink, Hrsg. *Apache Flink 0.9.0 Documentation: Java 8 Programming Guide*. 2015. URL: https://ci.apache.org/projects/flink/flink-docs-release-0.9/apis/java8.html (besucht am 15.09.2015).

[Apa15k] Apache Flink, Hrsg. *Apache Flink 0.9.0 Documentation: Jobs and Scheduling*. 2015. URL: https://ci.apache.org/projects/flink/flink-docs-release-0.9/internals/job_scheduling.html (besucht am 15.09.2015).

[Apa15l] Apache Flink, Hrsg. *Apache Flink 0.9.0 Documentation: Local Execution*. 2015. URL: https://ci.apache.org/projects/flink/flink-docs-release-0.9/apis/local_execution.html (besucht am 15.09.2015).

[Apa15m] Apache Flink, Hrsg. *Apache Flink 0.9.0 Documentation: Running Flink on YARN leveraging Tez*. 2015. URL: https://ci.apache.org/projects/flink/flink-docs-release-0.9/setup/flink_on_tez.html (besucht am 15.09.2015).

[Apa15n] Apache Flink, Hrsg. *Apache Flink 0.9.0 Documentation: Web Client*. 2015. URL: https://ci.apache.org/projects/flink/flink-docs-release-0.9/apis/web_client.html (besucht am 15.09.2015).

[Apa15o] Apache Flink, Hrsg. *Apache Flink 0.9.0 Documentation: YARN Setup*. 2015. URL: https://ci.apache.org/projects/flink/flink-docs-release-0.9/setup/yarn_setup.html (besucht am 15.09.2015).

[Apa15p] Apache Flink, Hrsg. *Apache Flink: Announcing Apache Flink 0.9.0*. 2015. URL: https://flink.apache.org/news/2015/06/24/announcing-apache-flink-0.9.0-release.html (besucht am 15.09.2015).

[Apa15q] Apache Flink, Hrsg. *Apache Flink: Scalable Batch and Stream Data Processing*. 2015. URL: https://flink.apache.org (besucht am 15.09.2015).

[Apa15r] Apache Flink, Hrsg. *Flink 0.9.0 Documentation - How to use logging*. 2015. URL: https://ci.apache.org/projects/flink/flink-docs-release-0.9/internals/logging.html (besucht am 15.09.2015).

[Apa12a] Apache Flume, Hrsg. *Welcome to Apache Flume*. 2012. URL: https://flume.apache.org/ (besucht am 15.09.2015).

[Apa15s] Apache Hadoop, Hrsg. *Apache Hadoop 2.7.1: HDFS Architecture*. 2015. URL: https://hadoop.apache.org/docs/r2.7.1/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html (besucht am 15.09.2015).

[Apa15t] Apache Hadoop, Hrsg. *Hadoop 2.7.1: Hadoop Cluster Setup*. 2015. URL: https://hadoop.apache.org/docs/r2.7.1/hadoop-project-dist/hadoop-common/ClusterSetup.html (besucht am 15.09.2015).

[Apa15u] Apache Hadoop, Hrsg. *PoweredBy - Hadoop Wiki*. 2015. URL: https://wiki.apache.org/hadoop/PoweredBy (besucht am 15.09.2015).

[Apa15v] Apache HBase, Hrsg. *Apache HBase*. 2015. URL: https://hbase.apache.org (besucht am 15.09.2015).

[Apa15w] Apache Hive, Hrsg. *Apache Hive*. 2015. URL: https://hive.apache.org (besucht am 15.09.2015).

[Apa15x] Apache Kafka, Hrsg. *Apache Kafka: Documentation*. 2015. URL: https://kafka.apache.org/documentation.html (besucht am 15.09.2015).

[Apa12b] Apache Logging, Hrsg. *Apache log4j 1.2*. 2012. URL: https://logging.apache.org/log4j/1.2 (besucht am 15.09.2015).

[Apa15y] Apache Maven, Hrsg. *Apache Maven Assembly Plugin - Introduction*. 2015. URL: https://maven.apache.org/components/plugins/maven-assembly-plugin (besucht am 15.09.2015).

[Apa15z] Apache Maven, Hrsg. *Maven - Welcome to Apache Maven*. 2015. URL: https://maven.apache.org/ (besucht am 15.09.2015).

[Apa15aa] Apache Mesos, Hrsg. *Apache Mesos*. 2015. URL: https://mesos.apache.org (besucht am 15.09.2015).

[Apa13a] Apache S4, Hrsg. *S4: Distributed Stream Computing Framework*. 2013. URL: https://incubator.apache.org/s4/ (besucht am 15.09.2015).

[Apa15ab] Apache Samza, Hrsg. *Apache Samza - State Management*. 2015. URL: https://samza.apache.org/learn/documentation/0.9/container/state-management.html (besucht am 15.09.2015).

[Apa15ac] Apache Samza, Hrsg. *apache/samza-hello-samza: Mirror of Apache Samza*. 2015. URL: https://github.com/apache/samza-hello-samza (besucht am 15.09.2015).

[Apa15ad] Apache Samza, Hrsg. *Documentation: Comparison Introduction*. 2015. URL: https://samza.apache.org/learn/documentation/0.9/comparisons/introduction.html (besucht am 15.09.2015).

[Apa15ae] Apache Samza, Hrsg. *Samza*. 2015. URL: https://samza.apache.org (besucht am 15.09.2015).

[Apa15af] Apache Samza, Hrsg. *Samza - Application Master*. 2015. URL: https://samza.apache.org/learn/documentation/0.9/yarn/application-master.html (besucht am 15.09.2015).

[Apa15ag] Apache Samza, Hrsg. *Samza - Background*. 2015. URL: https://samza.apache.org/learn/documentation/0.9/introduction/background.html (besucht am 15.09.2015).

[Apa15ah] Apache Samza, Hrsg. *Samza - Checkpointing*. 2015. URL: https://samza.apache.org/learn/documentation/0.9/container/checkpointing.html (besucht am 15.09.2015).

[Apa15ai] Apache Samza, Hrsg. *Samza - Coding Guide*. 2015. URL: https://samza.apache.org/contribute/coding-guide.html (besucht am 15.09.2015).

[Apa15aj] Apache Samza, Hrsg. *Samza - Concepts*. 2015. URL: https://samza.apache.org/learn/documentation/0.9/introduction/concepts.html (besucht am 15.09.2015).

[Apa15ak] Apache Samza, Hrsg. *Samza Configuration Reference*. 2015. URL: https://samza.apache.org/learn/documentation/0.9/jobs/configuration-table.html (besucht am 15.09.2015).

[Apa15al] Apache Samza, Hrsg. *Samza - Documentation*. 2015. URL: https://samza.apache.org/learn/documentation/0.9 (besucht am 15.09.2015).

[Apa15am] Apache Samza, Hrsg. *Samza - Hello Samza*. 2015. URL: https://samza.apache.org/startup/hello-samza/0.9 (besucht am 15.09.2015).

[Apa15an] Apache Samza, Hrsg. *Samza - Logging*. 2015. URL: https://samza.apache.org/learn/documentation/0.9/jobs/logging.html (besucht am 15.09.2015).

[Apa15ao] Apache Samza, Hrsg. *Samza: Metrics*. 2015. URL: https://samza.apache.org/learn/documentation/0.9/container/metrics.html (besucht am 15.09.2015).

[Apa15ap] Apache Samza, Hrsg. *Samza - Packaging*. 2015. URL: https://samza.apache.org/learn/documentation/0.9/jobs/packaging.html (besucht am 15.09.2015).

[Apa15aq] Apache Samza, Hrsg. *Samza - Remote Debugging with Samza*. 2015. URL: https://samza.apache.org/learn/tutorials/0.9/remote-debugging-samza.html (besucht am 15.09.2015).

[Apa15ar] Apache Samza, Hrsg. *Samza - SamzaContainer*. 2015. URL: https://samza.apache.org/learn/documentation/0.9/container/samza-container.html (besucht am 15.09.2015).

[Apa15as] Apache Samza, Hrsg. *Samza - Security*. 2015. URL: https://samza.apache.org/learn/documentation/0.9/operations/security.html (besucht am 15.09.2015).

[Apa15at] Apache Samza, Hrsg. *Samza - Windowing*. 2015. URL: https://samza.apache.org/learn/documentation/0.9/container/windowing.html (besucht am 15.09.2015).

[Apa15au] Apache Software Foundation, Hrsg. *Flink - ASF JIRA*. 2015. URL: https://issues.apache.org/jira/browse/FLINK (besucht am 15.09.2015).

[Apa14a] Apache Spark, Hrsg. *Spark Versioning Policy*. 2014. URL: https://cwiki.apache.org/confluence/display/SPARK/Spark+Versioning+Policy (besucht am 15.09.2015).

[Apa15av] Apache Spark, Hrsg. *Apache Spark - Lightning-Fast Cluster Computing*. 2015. URL: https://spark.apache.org/ (besucht am 15.09.2015).

[Apa15aw] Apache Spark, Hrsg. *Building Spark - Spark 1.4.0 Documentation*. 2015. URL: https://spark.apache.org/docs/1.4.0/building-spark.html#building-for-scala-211 (besucht am 15.09.2015).

[Apa15ax] Apache Spark, Hrsg. *Cluster Mode Overview - Spark 1.4.0 Documentation*. 2015. URL: https://spark.apache.org/docs/1.4.0/cluster-overview.html (besucht am 15.09.2015).

[Apa15ay] Apache Spark, Hrsg. *Configuration - Spark 1.4.0 Documentation*. 2015. URL: https://spark.apache.org/docs/1.4.0/configuration.html (besucht am 15.09.2015).

[Apa15az] Apache Spark, Hrsg. *Job Scheduling - Apache Spark 1.4.0 Documentation*. 2015. URL: https://spark.apache.org/docs/1.4.0/job-scheduling.html (besucht am 15.09.2015).

[Apa15ba] Apache Spark, Hrsg. *Monitoring and Instrumentation - Spark 1.4.0 Documentation*. 2015. URL: https://spark.apache.org/docs/1.4.0/monitoring.html (besucht am 15.09.2015).

[Apa15bb] Apache Spark, Hrsg. *Security - Spark 1.4.0 Documentation*. 2015. URL: https://spark.apache.org/docs/1.4.0/security.html (besucht am 15.09.2015).

[Apa15bc] Apache Spark, Hrsg. *Spark Standalone Mode - Spark 1.4.0 Documentation*. 2015. URL: https://spark.apache.org/docs/1.4.0/spark-standalone.html (besucht am 15.09.2015).

[Apa15bd] Apache Spark, Hrsg. *Spark Streaming Custom Receivers - Spark 1.4.0 Documentation*. 2015. URL: https://spark.apache.org/docs/1.4.0/streaming-custom-receivers.html (besucht am 15.09.2015).

[Apa15be] Apache Spark, Hrsg. *Spark Streaming - Spark 1.4.0 Documentation*. 2015. URL: https://spark.apache.org/docs/1.4.0/streaming-programming-guide.html (besucht am 15.09.2015).

[Apa15bf] Apache Spark, Hrsg. *[SPARK-5682] Add encrypted shuffle in spark*. 2015. URL: https://issues.apache.org/jira/browse/SPARK-5682 (besucht am 15.09.2015).

[Apa15bg] Apache Spark, Hrsg. *Submitting Applications - Apache Spark 1.4.0 Documentation*. 2015. URL: https://spark.apache.org/docs/1.4.0/submitting-applications.html (besucht am 15.09.2015).

[Apa13b] Apache Storm, Hrsg. *Storm 0.9.0 Released*. 2013. URL: https://storm.apache.org/2013/12/08/storm090-released.html (besucht am 15.09.2015).

[Apa14b] Apache Storm, Hrsg. *Common Topology Patterns*. 2014. URL: https://storm.apache.org/documentation/Common-patterns.html (besucht am 15.09.2015).

[Apa14c] Apache Storm, Hrsg. *Companies using Apache Storm*. 2014. URL: https://storm.apache.org/documentation/Powered-By.html (besucht am 15.09.2015).

[Apa14d] Apache Storm, Hrsg. *Concepts*. 2014. URL: https://storm.apache.org/documentation/Concepts.html (besucht am 15.09.2015).

[Apa14e] Apache Storm, Hrsg. *Creating a New Storm Project*. 2014. URL: https://storm.apache.org/documentation/Creating-a-new-Storm-project.html (besucht am 15.09.2015).

[Apa14f] Apache Storm, Hrsg. *FAQ*. 2014. URL: https://storm.apache.org/documentation/FAQ.html (besucht am 15.09.2015).

[Apa14g] Apache Storm, Hrsg. *Fault Tolerance*. 2014. URL: https://storm.apache.org/documentation/Fault-tolerance.html (besucht am 15.09.2015).

[Apa14h] Apache Storm, Hrsg. *Guaranteeing Message Processing*. 2014. URL: https://storm.apache.org/documentation/Guaranteeing-message-processing.html (besucht am 15.09.2015).

[Apa14i] Apache Storm, Hrsg. *Local Mode*. 2014. URL: https://storm.apache.org/documentation/Local-mode.html (besucht am 15.09.2015).

[Apa14j] Apache Storm, Hrsg. *Multi-Lang Protocol*. 2014. URL: https://storm.apache.org/documentation/Multilang-protocol.html (besucht am 15.09.2015).

[Apa14k] Apache Storm, Hrsg. *Running Topologies on a Production Cluster*. 2014. URL: https://storm.apache.org/documentation/Running-topologies-on-a-production-cluster.html (besucht am 15.09.2015).

[Apa14l] Apache Storm, Hrsg. *Setting up a Storm Cluster*. 2014. URL: https://storm.apache.org/documentation/Setting-up-a-Storm-cluster.html (besucht am 15.09.2015).

[Apa14m] Apache Storm, Hrsg. *Storm DSLs and Multi-Lang Adapters*. 2014. URL: https://storm.apache.org/documentation/DSLs-and-multilang-adapters.html (besucht am 15.09.2015).

[Apa14n] Apache Storm, Hrsg. *Strom Metrics*. 2014. URL: https://storm.apache.org/documentation/Metrics.html (besucht am 15.09.2015).

[Apa14o] Apache Storm, Hrsg. *Trident API Overview*. 2014. URL: https://storm.apache.org/documentation/Trident-API-Overview.html (besucht am 15.09.2015).

[Apa14p] Apache Storm, Hrsg. *Trident Spouts*. 2014. URL: https://storm.apache.org/documentation/Trident-spouts.html (besucht am 15.09.2015).

[Apa14q] Apache Storm, Hrsg. *Trident State*. 2014. URL: https://storm.apache.org/documentation/Trident-state.html (besucht am 15.09.2015).

[Apa14r] Apache Storm, Hrsg. *Understanding the Parallelism of a Storm Topology*. 2014. URL: https://storm.apache.org/documentation/Understanding-the-parallelism-of-a-Storm-topology.html (besucht am 15.09.2015).

[Apa15bh] Apache Storm, Hrsg. *Aapche Storm - Tutorial*. 2015. URL: https://storm.apache.org/documentation/Tutorial.html (besucht am 15.09.2015).

[Apa15bi] Apache Storm, Hrsg. *Apache Storm - Documentation*. 2015. URL: https://storm.apache.org/documentation/Documentation.html (besucht am 15.09.2015).

[Apa15bj] Apache Storm, Hrsg. *Storm, distributed and fault-tolerant realtime computation*. 2015. URL: https://storm.apache.org/ (besucht am 15.09.2015).

[Apa15bk] Apache Storm, Hrsg. *storm/external at v0.10.0-beta1*. 2015. URL: https://github.com/apache/storm/tree/v0.10.0-beta1/external (besucht am 15.09.2015).

[Apa15bl] Apache Storm, Hrsg. *storm/SECURITY.md at v0.10.0-beta1: Running Apache Storm Securely*. 2015. URL: https://github.com/apache/storm/blob/v0.10.0-beta1/SECURITY.md (besucht am 15.09.2015).

[Apa15bm] Apache Tez, Hrsg. *Apache Tez*. 2015. URL: https://tez.apache.org (besucht am 15.09.2015).

[Apa15bn] Apache ZooKeeper, Hrsg. *ZooKeper: Overview*. 2015. URL: https://zookeeper.apache.org/doc/r3.4.6/zookeeperOver.html (besucht am 15.09.2015).

[Ara+04] Arvind Arasu u. a. „STREAM: The Stanford Data Stream Management System“. In: *Book chapter* 2004-20 (2004).

[Bea+08] Jack K. Beaton u. a. „Usability evaluation for enterprise SOA APIs“. In: *Proceedings of the 2nd international workshop on Systems development in SOA environments*. Hrsg. von Kostas Kontogiannis. New York, NY, USA: ACM, 2008, S. 29–34.

[Bib15] Bibliographisches Institut GmbH, Hrsg. *Duden | Batch-Processing*. 2015. URL: http://www.duden.de/rechtschreibung/Batch_Processing (besucht am 15.09.2015).

[BIT12] BITKOM - Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e. V, Hrsg. *Leitfaden Big Data im Praxiseinsatz – Szenarien, Beispiele, Effekte*. Berlin, Deutschland, 2012.

[BIT14] BITKOM - Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e. V, Hrsg. *Leitfaden Big-Data-Technologien - Wissen für Entscheider*. Berlin, Deutschland, 2014.

[Boc14] Christian Bockermann. *A Survey of the Stream Processing Landscape*. Hrsg. von Technische Universität Dortmund, Lehrstuhl für künstliche Intelligenz. Dortmund, Deutschland, 2014.

[Can15a] Canonical Ltd., Hrsg. *OpenSSH Server*. 2015. URL: https://help.ubuntu.com/lts/serverguide/openssh-server.html (besucht am 15.09.2015).

[Can15b] Canonical Ltd., Hrsg. *The leading OS for PC, tablet, phone and cloud | Ubuntu*. 2015. URL: http://www.ubuntu.com/ (besucht am 15.09.2015).

[Car+15] Paris Carbone u. a. „Lightweight Asynchronous Snapshots for Distributed Dataflows“. In: *Computing Research Repository* abs/1506.08603 (2015).

[Cas15] Cask Data Inc., Hrsg. *Tigon*. 2015. URL: http://tigon.io (besucht am 15.09.2015).

[Cat11] Rick Cattell. „Scalable SQL and NoSQL data stores“. In: *ACM SIGMOD Record* 39.4 (2011), S. 12.

[Cha09] Sharma Chakravarthy. *Stream data processing: A quality of service perspective*. Bd. 36. Advances in database systems. New York, NY, USA: Springer, 2009.

[CA08] Sharma Chakravarthy und Raman Adaikkalavan. „Events and streams“. In: *Proceedings of the second international conference on Distributed event-based systems*. Hrsg. von Roberto Baldoni. New York, NY, USA: ACM, 2008, S. 1–12.

[Che+00] Jianjun Chen u. a. „NiagaraCQ: A Scalable Continuous Query System for Internet Databases“. In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. Hrsg. von Maggie Dunham. New York, NY, USA: ACM, 2000, S. 379–390.

[Cla03] Steven Clarke. *Using the cognitive dimensions - working framework*. 2003. URL: http://blogs.msdn.com/b/stevencl/archive/2003/12/03/57112.aspx (besucht am 15.09.2015).

[Cla04a] Steven Clarke. *Using the cognitive dimensions - consistency*. 2004. URL: http://blogs.msdn.com/b/stevencl/archive/2004/03/17/91626.aspx (besucht am 15.09.2015).

[Cla04b] Steven Clarke. *Using the cognitive dimensions - Penetrability*. 2004. URL: http://blogs.msdn.com/b/stevencl/archive/2004/02/03/66713.aspx (besucht am 15.09.2015).

[CB03] Steven Clarke und Curtis Becker. „Using the Cognitive Dimensions Framework to evaluate the usability of a class library“. In: *Proceedings of the 15h Workshop of the Psychology of Programming Interest Group (PPIG 2003)*. Hrsg. von Marian Petre und David Budgen. 2003.

[Clo15] Cloudera Inc., Hrsg. *CDH*. 2015. URL: http://de.cloudera.com/content/cloudera/en/products-and-services/cdh.html (besucht am 15.09.2015).

[CM12] Gianpaolo Cugola und Alessandro Margara. „Processing Flows of Information: From Data Stream to Complex Event Processing“. In: *ACM Computing Surveys* 44.3 (2012), S. 1–62.

[Dan15] Al Danial. *CLOC - Count Lines of Code*. 2015. URL: http://cloc.sourceforge.net (besucht am 15.09.2015).

[Das15] Akhil Das. *Re: spark streaming doubt*. 2015. URL: https://mail-archives.apache.org/mod_mbox/spark-user/201505.mbox/%3CCAHUQ+_YHamxdpOEHURCOZQRXhSLB+ufaYt81vGVDE03+SbziEw@mail.gmail.com%3E (besucht am 15.09.2015).

[Dat15a] Data Artisans GmbH, Hrsg. *data Artisans*. 2015. URL: http://data-artisans.com/ (besucht am 15.09.2015).

[Dat15b] Database Systems and Information Management Research Group, Fak. IV, TU Berlin, Hrsg. *Stratosphere: Next Generation Big Data Analytics Platform*. 2015. URL: http://stratosphere.eu (besucht am 15.09.2015).

[Dat15c] Databricks Inc., Hrsg. *About Us | Databricks*. 2015. URL: https://databricks.com/company/about-us (besucht am 15.09.2015).

[Dat15d] Databricks Inc., Hrsg. *Certified Spark Distribution | Databricks*. 2015. URL: https://databricks.com/spark/certification/certified-spark-distribution (besucht am 15.09.2015).

[DG04] Jeffrey Dean und Sanjay Ghemawat. „MapReduce: Simplified Data Processing on Large Clusters“. In: *Proceedings of the Symposium on Opearting Systems Design & Implementation*. Bd. 6. Berkeley, CA, USA: USENIX Association, 2004, S. 10.

[Dor15] Joachim Dorschel. *Praxishandbuch Big Data: Wirtschaft – Recht – Technik*. Weisbaden, Deutschland: Springer Gabler, 2015.

[Ecl15] Eclipse Foundation Inc., Hrsg. *Eclipse IDE for Java Developers | Packages*. 2015. URL: https://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/marsr (besucht am 15.09.2015).

[Éco15] École Polytechnique Fédérale de Lausanne, Hrsg. *The Scala Programming Language*. 2015. URL: http://www.scala-lang.org (besucht am 15.09.2015).

[Ell14] Byron Ellis. *Real-time analytics: Techniques to analyze and visualize streaming data*. Indianapolis, IN, USA: Wiley, 2014.

[End15] Markus Enderlein. *Unternehmensprofil Infomotion GmbH*. Frankfurt am Main, Deutschland, 2015.

[Eso15] Esoteric Software, Hrsg. *EsotericSoftware/kryo: Java serialization and cloning: fast, efficient, automatic*. 2015. URL: https://github.com/EsotericSoftware/kryo (besucht am 15.09.2015).

[Esp15] EsperTech Inc., Hrsg. *EsperTech - Esper*. 2015. URL: http://www.espertech.com/esper/ (besucht am 15.09.2015).

[EN11] Opher Etzion und Peter Niblett. *Event processing in action*. Greenwich, CT, USA: Manning, 2011.

[Eug+03] Patrick Th. Eugster u. a. „The many faces of publish/subscribe“. In: *ACM Computing Surveys* 35.2 (2003), S. 114–131.

[Ewe15] Stephan Ewen. *[FLINK-1675] Rework Accumulators*. Hrsg. von Apache Flink. 2015. URL: https://issues.apache.org/jira/browse/FLINK-1675 (besucht am 15.09.2015).

[Fac15] Facebook, Hrsg. *RocksDB*. 2015. URL: http://rocksdb.org (besucht am 15.09.2015).

[Fow05] Martin Fowler. *FluentInterface*. 2005. URL: http://www.martinfowler.com/bliki/FluentInterface.html.

[FC02] Xavier Franch und Juan P. Carvallo. „A quality-model-based approach for describing and evaluating software packages“. In: *Proceedings / IEEE Joint International Conference on Requirements Engineering*. Los Alamitos, CA, USA: IEEE Computer Society, 2002, S. 104–111.

[Fre14] Kirsten Freiding. *Infomotion laut PAC RADAR „Best in Class“-Anbieter für BI-Services*. 2014.

[Fue15] Miguel Fuentes Buchholtz. *storm-esper-bolt - Esper Bolt for Storm*. 2015. URL: https://github.com/miguelantonio/storm-esper-bolt (besucht am 15.09.2015).

[FYL02] Gabriel Pui Cheong Fung, Jeffrey Xu Yu und Wai Lam. „News Sensitive Stock Trend Prediction“. In: *Advances in Knowledge Discovery and Data Mining*. Hrsg. von G. Goos u. a. Bd. 2336. Lecture Notes in Computer Science. Berlin, Deutschland: Springer Berlin Heidelberg, 2002, S. 481–493.

[Gar15] Gartner Inc, Hrsg. *Gartner Survey Highlights Challenges to Hadoop Adoption*. 2015. URL: https://www.gartner.com/newsroom/id/3051717 (besucht am 15.09.2015).

[GW04] Boby George und Laurie Williams. „A structured experiment of test-driven development“. In: *Information and Software Technology* 46.5 (2004), S. 337–342.

[Gla08] Beate Glaubitz. *Standards für Evaluation*. 4. Aufl. Köln, Deutschland: Geschäftsstelle DeGEval, 2008.

[Goe15] Taylor P. Goetz. *Storm 0.10.0-beta Released*. Hrsg. von Apache Storm. 2015. URL: https://storm.apache.org/2015/06/15/storm0100-beta-released.html (besucht am 15.09.2015).

[Goo15a] Google Inc., Hrsg. *Google Compute Engine*. 2015. URL: https://cloud.google.com/compute (besucht am 15.09.2015).

[Goo15b] Google Inc., Hrsg. *google/guava: Google Core Libraries for Java 6+*. 2015. URL: https://github.com/google/guava (besucht am 15.09.2015).

[Gui+11] Dominique Guinard u. a. „From the Internet of Things to the Web of Things: Resource-oriented Architecture and Best Practices“. In: *Architecting the Internet of Things*. Hrsg. von Dieter Uckelmann, Mark Harrison und Florian Michahelles. Berlin, Deutschland: Springer Berlin Heidelberg, 2011, S. 97–129.

[Hal14a] Coda Hale. *Metrics*. Hrsg. von Yammer Inc. 2014. URL: http://metrics.dropwizard.io/3.1.0 (besucht am 15.09.2015).

[Hal14b] Coda Hale. *Metrics Core | Metrics*. Hrsg. von Yammer Inc. 2014. URL: http://metrics.dropwizard.io/3.1.0/manual/core (besucht am 15.09.2015).

[Heg03] Marcus Hegner. *Methoden zur Evaluation von Software: IZ-Arbeitsbericht Nr. 29*. Hrsg. von Informationszentrum Sozialwissenschaften der Arbeitsgemeinschaft Sozialwissenschaftlicher Institute e. V. Bonn, Deutschland, 2003.

[Hen09] Michi Henning. „API design matters“. In: *Communications of the ACM* 52.5 (2009), S. 46.

[Hic15] Rich Hickey. *Clojure - home*. 2015. URL: http://clojure.org (besucht am 15.09.2015).

[Hir+14] Martin Hirzel u. a. „A catalog of stream processing optimizations“. In: *ACM Computing Surveys* 46.4 (2014), S. 1–34.

[Hom15] Jakob Homan. *[SAMZA-646] Remove support for JDK6*. 2015. URL: https://issues.apache.org/jira/browse/SAMZA-646 (besucht am 15.09.2015).

[Hor15] Hortonworks Inc., Hrsg. *Hortonworks Data Platform*. 2015. URL: http://hortonworks.com/hdp (besucht am 15.09.2015).

[Hun+10] Patrick Hunt u. a. „ZooKeeper: Wait-free Coordination for Internet-scale Systems“. In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. Hrsg. von Paul Barham und Timothy Roscoe. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, S. 11.

[Hwa+05] Jeong-Hyon Hwang u. a. „High-Availability Algorithms for Distributed Stream Processing“. In: *Proceedings / 21st International Conference on Data Engineering, ICDE 2005*. Los Alamitos, CA, USA: IEEE Computer Society, 2005, S. 779–790.

[IBM15] IBM Deutschland GmbH, Hrsg. *IBM - InfoSphere Streams*. 2015. URL: http://www-03.ibm.com/software/products/de/infosphere-streams (besucht am 15.09.2015).

[iMa14] iMatix Corporation, Hrsg. *Code Connected - zeromq*. 2014. URL: http://zeromq.org (besucht am 15.09.2015).

[Int06] Internet Engineering Task Force. *RFC4422 - Simple Authentication and Security Layer (SASL)*. 2006.

[ISO99a] ISO. *ISO 9075-2 - Database Language SQL - Part 2: Foundation (SQL/-Foundation)*. Genf, Schweiz, 1999.

[ISO99b] ISO. *ISO 9126-1 Software Engineering - Product quality - Part 1: Quality model*. Genf, Schweiz, 1999.

[ISO01a] ISO. *ISO 9126-2: Software Engineering - Product Quality Part 2 - External Metrics*. Genf, Schweiz, 2001.

[ISO01b] ISO. *ISO 9126-3: Software Engineering - Product Quality Part 3 - Internal Metrics*. Genf, Schweiz, 2001.

[ISO01c] ISO. *ISO 9126-4: Software Engineering – Software Product Quality - Part 4: Quality In Use Metrics*. Genf, Schweiz, 2001.

[ISO08] ISO. *ISO 9075-1 - Database Language SQL - Part 1: SQL/Framework*. Genf, Schweiz, 2008.

[ISO11a] ISO. *ISO 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. Genf, Schweiz, 2011.

[ISO11b] ISO. *ISO 25040 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Evaluation process*. Genf, Schweiz, 2011.

[ID05] ISO und DIN. *ISO 25000 - Software-Engineering - Qualitätskriterien und Bewertung von Softwareprodukten (SQuaRE) - Leitfaden für SQuaRE*. Berlin, Deutschland, 2005.

[Jai+08] Namit Jain u. a. „Towards a streaming SQL standard: An empirical analysis on a German sample of non-family and family businesses“. In: *Proceedings of the VLDB Endowment*. Hrsg. von H. V. Jagadish. Bd. 1. 2008, S. 1379–1390.

[JUn15] JUnit, Hrsg. *JUnit - About*. 2015. URL: http://junit.org/ (besucht am 15.09.2015).

[Kre14] Jay Kreps. *[KAFKA-1682] Security for Kafka*. 2014. URL: https://issues.apache.org/jira/browse/KAFKA-1682 (besucht am 15.09.2015).

[KNR11] Jay Kreps, Neha Narkhede und Jun Rao. „Kafka: A distributed messaging system for log processing“. In: *Proceedings of the NetDB*. Hrsg. von Srikanth Kandula und Christopher Olston. 2011, S. 1–7.

[Kro01] Helmut Kromrey. „Evaluation - ein vielschichtiges Konzept“. In: *Sozialwissenschaften und Berufspraxis* 24.2 (2001), S. 105–132.

[Lal+06] Ashwin Lall u. a. „Data streaming algorithms for estimating entropy of network traffic“. In: *Proceedings of the joint international conference on Measurement and modeling of computer systems*. Hrsg. von Raymond Marie. New York, NY, USA: ACM, 2006, S. 145.

[Läm08] Ralf Lämmel. „Google's MapReduce programming model — Revisited“. In: *Science of Computer Programming* 70.1 (2008), S. 1–30.

[Lan01] Doug Laney. „3D data management: Controlling data volume, velocity and variety“. In: *META Group Research Note* 6 (2001), S. 70.

[Lei15] Jonathan Leibiusky. *xetorthio/jedis: Jedis*. 2015. URL: https://github.com/xetorthio/jedis (besucht am 15.09.2015).

[Lin15] LinkedIn Corporation, Hrsg. *Apache Kafka | LinkedIn Data Team*. 2015. URL: http://data.linkedin.com/opensource/kafka (besucht am 15.09.2015).

[LIX14] Xiufeng Liu, Nadeem Iftikhar und Xike Xie. „Survey of real-time processing systems for big data“. In: *Proceedings of the 18th International Database Engineering & Applications Symposium*. Hrsg. von Bipin C. Desai u. a. ICPS. New York, NY, USA: Association for Computing Machinery, 2014, S. 356–361.

[LM07] Dirk Louis und Peter Müller. *Das Java 6 Codebook*. Premium Codebook. München, Deutschland: Addison-Wesley, 2007.

[MBW10] Peter Mandl, Andreas Bakomenko und Johann Weiss. *Grundkurs Datenkommunikation: TCP/IP-basierte Kommunikation: Grundlagen, Konzepte und Standards*. 2., überarbeitete und aktualisierte Aufl. Studium. Wiesbaden, Deutschland: Vieweg+Teubner Verlag / GWV Fachverlage, Wiesbaden, 2010.

[Map15] MapR Technologies Inc., Hrsg. *MapR Distribution*. 2015. URL: https://www.mapr.com/products/mapr-distribution-including-apache-hadoop (besucht am 15.09.2015).

[Mar13] Nathan Marz. *nathanmarz/storm-deploy: One click deploy for Storm clusters on AWS*. 2013. URL: https://github.com/nathanmarz/storm-deploy (besucht am 15.09.2015).

[MW15] Nathan Marz und James Warren. *Big data: Principles and best practices of scalable real-time data systems*. Safari Tech Books Online. Shelter Island, NY, USA: Manning, 2015.

[Mas15] Massachusetts Institute of Technology, Hrsg. *Kerberos: The Network Authentication Protocol*. 2015. URL: http://web.mit.edu/kerberos (besucht am 15.09.2015).

[MD89] Dennis McCarthy und Umeshwar Dayal. „The architecture of an active database management system“. In: *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*. Hrsg. von James Clifford. Bd. 18.2. SIGMOD record. New York, NY, USA: ACM, 1989, S. 215–224.

[MBM13] Marcelo R. N. Mendes, Pedro Bizarro und Paulo Marques. „FINCoS: Benchmark Tools for Event Processing Systems“. In: *Proceedings of the 4th ACMSPEC International Conference on Performance Engineering*. Hrsg. von Petr Tůma. New York, NY, USA: ACM, 2013, S. 431.

[Met15] Robert Metzger. *[FLINK-2386] Implement Kafka connector using the new Kafka Consumer API*. Hrsg. von Apache Flink. 2015. URL: https://issues.apache.org/jira/browse/FLINK-2386 (besucht am 15.09.2015).

[Mic15] Microsoft Corporation, Hrsg. *Event Hubs | Microsoft Azure*. 2015. URL: https://azure.microsoft.com/de-de/services/event-hubs (besucht am 15.09.2015).

[Mil06] David L. Mills. *Computer network time synchronization: The network time protocol*. Boca Raton, FL, USA: CRC/Taylor & Francis, 2006.

[Mil13] Adrian Milne. *Complex Event Processing Made Easy (using Esper) - DZone Integration*. 2013. URL: https://dzone.com/articles/complex-event-processing-made.

[Nab+14] Zubair Nabi u. a. *Of Streams and Storms: A Direct Comparison of IBM InfoSphere Streams and Apache Storm in a Real World Use Case - Email Processing*. 2014. URL: https://developer.ibm.com/streamsdev/2014/04/22/streams-apache-storm/ (besucht am 15.09.2015).

[NTP15] NTP Project, Hrsg. *ntp.org: Home of the Network Time Protocol*. 2015. URL: http://www.ntp.org (besucht am 15.09.2015).

[NYS14] NYSE Technologies, Hrsg. *DAILY TAQ CLIENT SPECIFICATION*. 2014.

[Ole15] Damir Olejar. *Learning Apache Spark: How to mix Java and the Scala code (use one with another) in Eclipse*. 2015. URL: http://learningapachespark.blogspot.de/2015/04/14-how-to-mix-java-and-spark-code-use.html (besucht am 15.09.2015).

[OI13] Open Group und IEEE, Hrsg. *wc*. 2013. URL: http://pubs.opengroup.org/onlinepubs/9699919799/utilities/wc.html (besucht am 15.09.2015).

[Ora15a] Oracle Corporation, Hrsg. *JAAS Reference Guide*. 2015. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/JAASRefGuide.html (besucht am 15.09.2015).

[Ora15b] Oracle Corporation, Hrsg. *Java SE | Oracle Technology Network | Oracle*. 2015. URL: http://www.oracle.com/technetwork/java/javase/overview/index.html (besucht am 15.09.2015).

[Ora15c] Oracle Corporation, Hrsg. *Java SE Technologies - Database*. 2015. URL: http://www.oracle.com/technetwork/java/javase/jdbc/index.html (besucht am 15.09.2015).

[Ora15d] Oracle Corporation, Hrsg. *Object (Java Platform SE 8)*. 2015. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode-- (besucht am 15.09.2015).

[Ora15e] Oracle Corporation, Hrsg. *The Essentials of Filters*. 2015. URL: http://www.oracle.com/technetwork/java/filters-137243.html (besucht am 15.09.2015).

[Piv15] Pivotal Software Inc., Hrsg. *RabbitMQ - Messaging that just works*. 2015. URL: https://www.rabbitmq.com (besucht am 15.09.2015).

[Pol+92] Peter G. Polson u. a. „Cognitive walkthroughs: A method for theory-based evaluation of user interfaces“. In: *International Journal of Man-Machine Studies* 36.5 (1992), S. 741–773.

[Pos15] PostgreSQL Global Development Group, Hrsg. *PostgreSQL*. 2015. URL: http://www.postgresql.org/ (besucht am 15.09.2015).

[Pre14] Tom Preston-Werner. *Semantic Versioning 2.0.0*. 2014. URL: http://semver.org (besucht am 15.09.2015).

[Pyt15] Python Software Foundation, Hrsg. *Welcome to Python.org*. 2015. URL: https://www.python.org (besucht am 15.09.2015).

[QOS15a] QOS.ch Sàrl, Hrsg. *Logback Home*. 2015. URL: http://logback.qos.ch (besucht am 15.09.2015).

[QOS15b] QOS.ch Sàrl, Hrsg. *SLF4J*. 2015. URL: http://www.slf4j.org (besucht am 15.09.2015).

[R F15] R Foundation, Hrsg. *R: The R Project for Statistical Computing*. 2015. URL: https://www.r-project.org (besucht am 15.09.2015).

[Ras15] Imran Rashid. *Spark Accumulators, What Are They Good For?* 2015. URL: http://imranrashid.com/posts/Spark-Accumulators (besucht am 15.09.2015).

[Ric13] Chris Riccomini. *Aapche Samza: LinkedIn's Real-time Processing Framework*. Hrsg. von LinkedIn Corporation. 2013. URL: https://engineering.linkedin.com/data-streams/apache-samza-linkedins-real-time-stream-processing-framework (besucht am 15.09.2015).

[RW97] David S. Rosenblum und Alexander L. Wolf. „A design framework for Internet-scale event observation and notification“. In: *ACM SIGSOFT Software Engineering Notes* 22.6 (1997), S. 344–360.

[Sam15] Samba Project, Hrsg. *Samba - opening windows to a wider world*. 2015. URL: https://www.samba.org (besucht am 15.09.2015).

[San15a] Salvatore Sanfilippo. *Introduction to Redis - Redis*. 2015. URL: http://redis.io/topics/introduction (besucht am 15.09.2015).

[San15b] Salvatore Sanfilippo. *Redis: Clients - Java*. 2015. URL: http://redis.io/clients/#java (besucht am 15.09.2015).

[Set13] Siddharth Seth. *[YARN-666] Support rolling upgrades in YARN*. Hrsg. von Apache Hadoop. 2013. URL: https://issues.apache.org/jira/browse/YARN-666 (besucht am 15.09.2015).

[SH06] Tony Shan und Winnie Hua. „Taxonomy of Java Web Application Frameworks“. In: *IEEE International Conference on e-Business Engineering, 2006*. Hrsg. von Wei-Tek Tsai. Los Alamitos, CA, USA: IEEE Computer Society, 2006, S. 378–385.

[Sho04] Jim Shore. „Fail Fast“. In: *IEEE Software* 21.05 (2004), S. 21–25.

[Sic15] Tony Siciliani. *Streaming Big Data: Storm, Spark and Samza*. 2015. URL: https://dzone.com/articles/streaming-big-data-storm-spark (besucht am 15.09.2015).

[Son11] Sonatype Inc, Hrsg. *The Central Repository Search Engine*. 2011. URL: https://search.maven.org (besucht am 15.09.2015).

[SN15] Andy Stanford-Clark und Arlen Nipper. *MQTT*. 2015. URL: http://mqtt.org (besucht am 15.09.2015).

[SÇZ05] Michael Stonebraker, Uğur Çetintemel und Stan Zdonik. „The 8 requirements of real-time stream processing“. In: *ACM SIGMOD Record* 34.4 (2005), S. 42–47.

[Stö15] Uta Störl. *Fachbereich Informatik: Big Data Cluster*. 2015. URL: https://www.fbi.h-da.de/organisation/personen/stoerl-uta/big-data-cluster.html (besucht am 15.09.2015).

[Ter+92] Douglas Terry u. a. „Continuous queries over append-only databases“. In: *ACM SIGMOD Record* 21.2 (1992), S. 321–330.

[The15] The Netty Project, Hrsg. *Netty: Home*. 2015. URL: http://netty.io/index.html (besucht am 15.09.2015).

[Tos+14] Ankit Toshniwal u. a. „Storm @twitter“. In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. Hrsg. von Richard Hull, Curtis Dyreson und Pablo Barcelo. New York, NY, USA: ACM, 2014, S. 147–156.

[Tro14] Jason Trost. *Strom Metrics How-To*. 2014. URL: https://www.endgame.com/blog/storm-metrics-how (besucht am 15.09.2015).

[Twi15] Twitter Inc., Hrsg. *twitter/summingbird*. 2015. URL: https://github.com/twitter/summingbird (besucht am 15.09.2015).

[Typ15] Typesafe Inc., Hrsg. *Akka*. 2015. URL: http://akka.io/ (besucht am 15.09.2015).

[TS15] Kostas Tzoumas und Henry Saputra. *Flink Roadmap - Apache Flink*. 2015. URL: https://cwiki.apache.org/confluence/display/FLINK/Flink+Roadmap (besucht am 15.09.2015).

[Vav+13] Vinod Kumar Vavilapalli u. a. „Apache Hadoop YARN“. In: *Proceedings of the 4th annual Symposium on Cloud Computing*. Hrsg. von Guy Lohman. ACM Digital Library. New York, NY, USA: ACM, 2013, S. 1–16.

[VRR10] Krešimir Vidačković, Thomas Renner und Sascha Rex. *Marktübersicht Real-Time Monitoring Software: Event Processing Tools im Überblick ; [Forschungsprojekt] iC RFID - intelligent catering*. Stuttgart, Deutschland: Fraunhofer-Verlag, 2010.

[VN10] Kashi Venkatesh Vishwanath und Nachiappan Nagappan. „Characterizing cloud computing hardware reliability“. In: *Proceedings of the 1st ACM symposium on Cloud computing*. Hrsg. von Joseph M. Hellerstein, Surajit Chaudhuri und Mendel Rosenblum. New York, NY, USA: ACM, 2010, S. 193.

[WC10] Dean Wampler und Tony Clark. „Guest Editors' Introduction: Multiparadigm Programming“. In: *IEEE Software* 27.5 (2010), S. 20–24.

[WM12] Michael E. Whitman und Herbert J. Mattord. *Principles of information security*. 4. ed., International ed. Stamford, CT, USA: Course Technology Cengage Learning, 2012.

[Woo15] Brett Wooldridge. *brettwooldridge/HikariCP: HikariCP*. 2015. URL: https://github.com/brettwooldridge/HikariCP (besucht am 15.09.2015).

[WT03] Heinrich Wottawa und Heike Thierau. *Lehrbuch Evaluation*. 3., korrigierte Aufl. Psychologie Lehrbuch. Bern, Schweiz: Huber, 2003.

[Xu13a] James Xu. *[STORM-151] Support multilang in Trident*. 2013. URL: https://issues.apache.org/jira/browse/STORM-151 (besucht am 15.09.2015).

[Xu13b] James Xu. *[STORM-167] proposal for storm topology online update*. 2013. URL: https://issues.apache.org/jira/browse/STORM-167 (besucht am 15.09.2015).

[Yah14] Yahoo Inc., Hrsg. *yahoo/storm-yarn: Storm for Yarn*. 2014. URL: https://github.com/yahoo/storm-yarn (besucht am 15.09.2015).

[Zah+10] Matei Zaharia u. a. „Spark: Cluster Computing with Working Sets“. In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. Hrsg. von Erich Nahum und Dongyan Xu. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, S. 10.

[Zah+12] Matei Zaharia u. a. „Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters“. In: *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*. Hrsg. von Rodrigo Fonseca und Dave Maltz. HotCloud'12. Berkeley, CA, USA: USENIX Association, 2012, S. 10.

[Zah+13] Matei Zaharia u. a. „Discretized Streams: Fault-Tolerant Streaming Computation at Scale“. In: *SOSP '13*. Hrsg. von Michael Kaminsky und Mike Dahlin. New York, NY, USA: ACM, 2013, S. 423–438.

[Zel06] Andreas Zeller. *Why programs fail: A guide to systematic debugging*. San Francisco, CA, USA und Heidelberg, Deutschland: Morgan Kaufmann Publishing und dpunkt.verlag, 2006.

[Zha+09] Xiaolan J. Zhang u. a. „Implementing a high-volume, low-latency market data processing system on commodity hardware using IBM middleware“. In: *Proceedings of the 2nd Workshop on High Performance Computational Finance*. Hrsg. von David Daly. New York, NY, USA: ACM, 2009, S. 1–8.