PD Dr. Michael Mock
(michael.mock@iais.fraunhofer.de)
ist Senior Data Scientist am Fraunhofer IAIS und Privatdozent an der Universität Magdeburg. Im Zentrum seiner Forschungsarbeiten stehen die Entwicklung von verteilten Systemen und Algorithmen zur Verarbeitung großer Datenmengen insbesondere in massiven Datenströmen. Zurzeit leitet und koordiniert er das von der EU geförderte Forschungsprojekt FERARI, in dem eine hochskalierbare verteilte Architektur zur zeitnahen Verarbeitung massiver Datenströme entwickelt wird.

Karl-Heinz Sylla
(karl-heinz.sylla@iais.fraunhofer.de)
ist seit den 80er Jahren in der Szene des objektorientierten Software-Engineering aktiv. In zahllosen Seminaren und Schulungen hat er Softwareentwickler und -entscheider ausgebildet. Er arbeitet als Senior-Wissenschaftler am Fraunhofer IAIS und hat langjährige Erfahrung in der Leitung von Wirtschaftsprojekten. Zurzeit ist er zuständig für die Entwicklung von Big-Data-Architekturen.

Dr. Dirk Hecker
(dirk.hecker@iais.fraunhofer.de)
leitet die Abteilung Knowledge Discovery am Fraunhofer-Institut für Intelligente Analyse- und Informationssysteme IAIS. Er studierte Geo-Informatik an den Universitäten Köln und Bonn und promovierte an der Universität zu Köln. Seit 2014 ist Dr. Hecker Geschäftsführer der neu gegründeten »Fraunhofer-Allianz Big Data«, einem Verbund von 25 Fraunhofer-Instituten zur branchenübergreifenden Forschung und Technologieentwicklung im Bereich Big Data.

Skalierbarkeit und Architektur von Big-Data-Anwendungen

Big-Data-Komponenten, mit denen die Internetriesen wie Google, Facebook oder Amazon ihre Big-Data-Anwendungen bauen, werden von Open-Source-Communities angeboten und vielfach ergänzt. Die technologische Basis, Chancen von Big Data zu nutzen, ist sowohl frei verfügbar, als auch im Portfolio großer Systemanbieter umfangreich enthalten. Trotzdem stehen vor der Entwicklung einer Big-Data-Anwendung vergleichsweise hohe Hürden: Big-Data-Komponenten haben gemeinhin einen geringeren Funktionsumfang, als man es bisher zum Beispiel von einem Betriebssystem, einer klassischen relationalen Datenbank oder einem BI-System gewohnt ist. Dieser Beitrag gibt einen Einblick über die wesentliche technische Leistung und den damit verbundenen Nutzen von Big-Data-Komponenten. Anschließend wird an einem konkreten Beispiel gezeigt, wie im Konzept der Lambda-Architektur Komponenten aufeinander abgestimmt eingesetzt werden.

Skalierbarkeit von Big-Data-Anwendungen

Big-Data-Anwendungen werden oft gekennzeichnet durch die Eigenschaften „*Volume*, *Velocity*, *Variety*". Diese Eigenschaften beschreiben ihre Fähigkeit, große Datenmengen zu verarbeiten (*Volume*), auch bei hohen Datenraten zeitnah Antworten zu liefern (*Velocity*), und verschiedene, auch unstrukturierte Datenquellen (*Variety*) zusammenführen zu können. Je nach spezifischer Anwendung bekommen die Eigenschaften verschiedene Gewichte und werden durch verschiedene Big-Data-Komponenten unterstützt.

Der Schlüssel zur Erreichung dieser Eigenschaften liegt in der sogenannten „horizontalen Skalierbarkeit" des zugrundeliegenden Big-Data-Systems. Diese bezeich-net die im System umgesetzte Möglichkeit, durch Hinzunahme weiterer Rechnerknoten die Anwendung flexibel an die

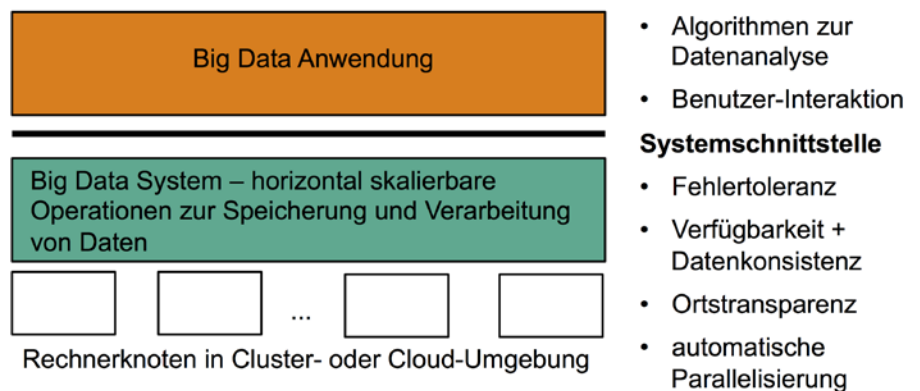

Abb. 1: Die Systemschnittstelle von Big Data-Systemen macht die horizontale Skalierung transparent für die Big-Data-Anwendung.

erforderlichen Datenvolumina und Verarbeitungskapazitäten anzupassen.

Erfordert eine Big-Data-Anwendung ein wachsendes Datenvolumen oder werden Verarbeitungs- und Antwortzeiten zu lang, so kann ein Engpass durch horizontale Skalierung behoben werden. Big-Data-Systeme, die eine solche horizontale Skalierung sogar im laufenden Betrieb vornehmen können, werden auch als „elastisch" bezeichnet.

Durch eine horizontale Skalierung ändern sich für eine Anwendung zunächst eigentlich relevante Aspekte, wie zum Beispiel der Speicherort von Daten, da die Daten auf neue, weitere Rechner verteilt werden. Dies ist scheinbar ein Nachteil gegenüber dem Verfahren der „vertikalen Skalierung", bei dem eine Anwendung nicht verteilt, sondern auf einem Rechnerknoten läuft und die Kapazität durch Austausch gegen eine größere Einheit erhöht wird.

Vorteil der horizontalen Skalierung dagegen ist es, dass die natürlichen Kapazitätsgrenzen eines Rechners beliebig überschritten werden können. Zudem sind mehrere Rechner mit Standard-Hardware oft günstiger als immer leistungsstärkere Superrechner mit Spezial-Hardware. Internetgrößen wie Google, Facebook, Twitter, Microsoft oder Amazon betreiben Rechnerfarmen mit mehreren Tausend Rechnern, auf denen Anwendungen horizontal skaliert werden können.

Wie wird nun eine horizontale Skalierung ermöglicht, ohne dass die Big-Data-Anwendung angepasst oder gar neu programmiert werden muss? Die Antwort liegt in den Big-Data-Systemen, die die komplizierten Aspekte der horizontalen Skalierung vollständig vor der Anwendung verbergen.

Dazu stellen sie eine Systemschnittstelle zur Entwicklung der Anwendung bereit und führen unterhalb der Systemschnittstelle, also verborgen für die Anwendung, Protokolle aus, welche globale Systemeigenschaften zusichern und somit quasi „für die Anwendung erledigen" (siehe Abbildung 1).

Eine Vielzahl von Komponenten für solche Big-Data-Systeme werden als Open Source entwickelt und bereitgestellt. Das bekannteste Beispiel ist Hadoop, das die von Google entwickelte MapReduce-Methode zur Programmierung von Rechner-Clustern umsetzt.

Ein anderes Beispiel sind sogenannte NoSql-Datenbaken, die große Datenmengen horizontal skalierbar verwalten können. Wir gehen später noch auf sinnvolle Kombinationen und den Ansatz der Lambda-Architektur zur Organisation der Komponenten ein.

Abbildung 1 zeigt die grundsätzliche Trennung zwischen Big-Data-Anwendung und Big-Data-System. Die Big-Data-Anwendung setzt Algorithmen zur Datenanalyse und zumeist eine Schnittstelle zum Benutzer um. Sie basiert auf einer Systemschnittstelle, die vom Big-Data-System bereitgestellt wird.

Das Big-Data-System realisiert die Systemschnittstelle auf einer Menge von Rechnern in einem oder mehreren Clustern (oder in einer Rechner-Cloud). Diese Menge kann wachsen oder auch schrumpfen, um sich den Kapazitätsanforderungen der Anwendung anzupassen. Dafür führt das Big-Data-System eine Vielzahl von verteilten Protokollen aus, die ihren Ursprung zumeist in der Forschung zu verteilten Systemen haben und nun über die Big-Data-Systeme als Open Source weltweit in Big-Data-Anwendungen zum Einsatz kommen.

Die wichtigsten Eigenschaften, die durch die Protokolle für die Anwendung zugesichert werden, sind:

- **Fehlertoleranz**: Die Anwendung muss sich nicht um Probleme kümmern, die dadurch hervorgerufen werden, dass Rechner ausfallen oder Netzwerkprobleme auftreten. Da die Wahrscheinlichkeit für das Auftreten solcher Probleme stark mit der steigenden Anzahl von genutzten Rechnern wächst, ist diese Eigenschaft unabdingbar für die horizontale Skalierbarkeit.
  Realisiert wird sie über ein Zusammenspiel verschiedener Protokolle und Verfahren: Fehlererkennungsprotokolle, die konsistent erkennen, ob ein Rechner ausgefallen ist oder nicht; Mitgliedschaftsprotokolle, die konsistent festlegen, auf welche Rechner die Anwendung verteilt wird; Konsensusprotokolle, die einheitliche Konfigu-rationsdaten über alle beteiligten Rechner verteilen sowie Verfahren, die den Fortschritt von Berechnungen kontrollieren, Sicherungspunkte anlegen und abgebrochene Berechnungen wieder aufsetzen. Als konkretes Beispiel sei hier das Open Source System „Zookeeper" genannt, das eine Variante des Paxos-Konsensus-Protokolls realisiert.
- **Verfügbarkeit und Datenkonsistenz:** Die Daten bleiben für die Anwendung zugreifbar, auch wenn einzelne Rechner ausfallen, auf denen Anwendungsdaten gespeichert sind. Es werden Verfahren zur Datenreplikation eingesetzt, welche die Daten automatisch auf mehrere Rechner verteilen und die Zugriffe auf die einzelnen Speicherorte der Daten durch ein geeignetes Zugriffsprotokoll synchronisieren. Im Prinzip bleibt die Replikation vor der Anwendung verborgen. Auch bei einer parallelen Anwendung erscheint es so, als ob alle Daten nur in einer einzelnen Kopie vorhanden sind und alle Zugriffe in einer sequenziellen Reihenfolge ausgeführt würden.
  Diese Eigenschaft ist auch als Kriterium der Linearisierbarkeit oder „Ein-Kopie-Serialisierbarkeit" bekannt. Jedoch führt ein solches strenges Konsistenzkriterium oft zu komplexen Protokollen, die verlangsamte Ausführungszeiten implizieren oder sogar die Anwendung blockieren, selbst wenn noch einige Kopien vorhanden sind. Deswegen sind schwächere Konsistenzkriterien wie „letztendliche Konsistenz" (eventual consistency) eingeführt worden, die eine temporäre Verletzung der Konsistenz zulassen [Vog09].
  Trotzdem wird dabei zugesichert, dass die Kopien über die Zeit hinweg zu einem identischen Zustand konvergierten, wobei der Zeitbegriff oft durch logische Uhren oder sogenannte „Vektor-Clocks" im verteilten System realisiert ist. Insgesamt besteht ein „Trade-off" zwischen Konsistenz, Verfügbarkeit und Fehlertoleranz, der im sogenannten CAP-Theorm von Eric Brewer formuliert worden ist [Bre00] – weitere Erläuterungen dazu finden sich zum Beispiel im Bitkom-Leitfaden zur Big-Data-Architektur [Bit14].
  Ein Beispiel für Replikation zur Erhöhung der Verfügbarkeit und Zusicherung von Konsistenz findet sich in der horizontal skalierbaren NoSQL-Datenbank Cassandra [Cas], die das „Quorum Consensus-Protokoll" zur Zugriffskontrolle auf Kopien realisiert, es dabei aber dem Benutzer erlaubt, das für den Zugriff benötigte Quorum zu senken und dann im Hintergrund letztendliche Konsistenz zusichert. Cassandra ist ursprünglich von Facebook entwickelt worden und

vereinigt Konzepte, die auf das Dynamo System von Amazon und die NoSQL-Datenbank BigTable von Google zurückgehen.

- **Ortstransparenz:** Die Anwendung kann auf Daten zugreifen, ohne wissen oder spezifizieren zu müssen, auf welchem Rechner die Daten gespeichert sind. Diese Eigenschaft ist insbesondere deswegen wichtig, weil sich der Speicherort von Daten im Big-Data-System aufgrund von Ausfällen von Rechnern oder dem Hinzufügen weiterer Rechner ändern kann.
  Das grundsätzliche Problem der Ortstransparenz ist seit langem in verschiedenen Systemen gelöst, die einen einheitlichen Namensraum realisieren. Einheitliche Namensräume gibt es bei verteilten Dateisystemen, im Internet und auch bei Computing-Grid-Systemen, die unternehmensübergreifende Clusterbildung ermöglichen.
  Im Kontext von Big Data ergibt sich jedoch eine weitere Einschränkung: Die Ortstransparenz soll derart realisiert werden, dass Änderungen in der Konfiguration (also neue Rechner hinzufügen oder Rechner rausnehmen) zu möglichst wenig Bewegung von Daten führen. Ein Big-Data-System, bei dem bei jedem Rechnerausfall alle Daten neu auf die verbliebenen Rechner verteilt werden müssten, wäre sicherlich mehr mit der internen Verteilung von Daten als mit der eigentlichen Ausführung der Anwendung beschäftigt und dadurch wieder nicht horizontal skalierbar.
  In den meisten Big-Data-Systemen wird das Verfahren des „Consistent Hashing“ angewendet, um den Rechner zu finden, auf dem die gesuchten Daten liegen. Es hat seinen Ursprung im Bereich der Peer-to-Peer-Netzwerke, die gedacht sind, Daten verteilt und repliziert über alle im Internet beteiligten Rechner einer Nutzergruppe zu verteilen. Beim Consistent Hashing wird der Speicherort eines Datums über eine spezielle hash-Funktion gefunden. „Konsistente“ Hash-Funktionen haben die Eigenschaft, bei Änderungen des zugelassenen Wertebereichs nur möglichst wenig Eingabewerte neu zuzuordnen.
- **Automatische Parallelisierung:** Die Anwendung wird parallelisiert über mehrere Rechner hinweg ausgeführt, ohne dass in der Anwendung parallele Instanzen oder Kommunikationsvorgänge zwischen Instanzen explizit programmiert werden müssen. Insbesondere sollte der Grad der Parallelisierung durch die Datenmenge und die Anzahl der verfügbaren Rechner bestimmt und geändert werden können, ohne dass die Anwendung explizit angepasst werden muss.
  Verfahren zur automatischen Parallelisierung sind schon lange im Bereich der Programmiersprachen/Rechnerarchitekturen eingeführt und basieren stets darauf, die Anwendung nur mit solchen Operationen zu konstruieren, die von vornherein parallelisiert sind. Beispiele hierfür sind Vektoroperationen, Programme in funktionalen Sprachen – wie zum Beispiel Lambda-Ausdrücke, der Namensgeber für die unten beschriebene Architektur – oder Programme, deren Ausführung nur durch den Datenfluss getrieben wird.
  Neu im Bereich der Big-Data-Systeme ist nun, dass derartige Programme auf Systeme umgesetzt werden, die zudem die oben beschriebenen Eigenschaften erfüllen. So ist zum Beispiel das MapReduce-Konzept als Konstrukt der funktionellen Programmierung schon lange wohl bekannt. Dessen Umsetzung von Google (und Hadoop) auf ein horizontal skalierbares Cluster machte daraus ein Big-Data-System [DaG04]. Datenfluss-getriebene Ansätze werden in Big-Data-Systemen typischerweise mit Systemen zur Programmierung von Workflows wie Cascading, Crunch oder Storm umgesetzt. Funktionale Programmierung findet sich heute prominent in der Programmiersprache Scala wieder, die zunehmend in Big-Data-Systemen wie Spark oder Scalding direkt unterstützt wird.

Insgesamt bietet das Big-Data-System eine Fülle von Leistungen, die die horizontale Skalierbarkeit überhaupt erst ermöglichen. Somit wird der Anwendungsentwickler davon befreit, sich mit den Problemfeldern des unterliegenden verteilten Rechnerverbundes beschäftigen zu müssen.

Diese enorme Erleichterung für die Entwicklung von Big-Data-Anwendungen hat aber auch ihren Preis: die Anwendung muss sich an die von der Systemschnittstelle des Big-Data-Systems angebotenen und unterstützen Operationen zur Datenverarbeitung und -speicherung halten.

Diese Operationen sind aber von einem wesentlich geringeren Funktionsumfang, als es der Anwendungsentwickler bisher zum Beispiel von einem Betriebssystem oder einer klassischen relationalen Datenbank her gewohnt ist. Schlimmer noch: die bislang als „Big-Data-System“ bezeichnete Schicht ist, zumindest was den Open-Source-Bereich angeht, keineswegs ein einheitliches System. Stattdessen umfasst sie eine Vielzahl von unterschiedlichsten Big-Data-Komponenten, die alle verschiedene Schwerpunkte und Anwendungsgebiete und natürlich auch verschiedene Schnittstellen haben.

Man beachte, dass es neben dem Open-Source-Bereich auch etliche kommerzielle Big-Data-Systeme gibt, die sich in der Regel als „All-in-One“-Lösung präsentieren, um dem Entwickler wieder eine einheitliche Schnittstelle zur Anwendungsentwicklung zu bieten. Mit dieser einheitlichen Lösung bindet sich der Anwender an den einen kommerziellen Anbieter, zumeist sogar sowohl die Software als auch die Hardware betreffend. Eine weitere Aufzählung oder gar Bewertung solcher Lösungen nehmen wir explizit nicht vor – dies würde den Rahmen und den Charakter des Beitrags deutlich sprengen.

Auch im Open-Source-Bereich gibt es eine Reihe kommerzieller Anbieter, die bereits zueinander passende Komponenten als „Distribution“ vertreiben. Auch dies werden wir hier nicht weiter besprechen, zumal für das Design einer Anwendung unabhängig von der Wahl der „Distribution“, immer noch vorab zu klären bleibt, auf welchen Big-Data-Komponenten diese aufsetzten soll und wie die einzelnen Komponenten und zu welchem Zweck kombiniert werden sollen.

Als Denkmuster, um diese Fragen zu beantworten und auch als Vorlage zur tatsächlichen Systemkonstruktion hat sich in der Praxis die von Nathan Marz vorgeschlagene „Lambda-Architektur“ bewährt [MaW13], die wir im Folgenden weiter vorstellen und an einem von uns im Fraunhofer IAIS durchgeführten Anwendungsbeispiel erläutern werden.

Architektur von Big-Data-Anwendungen: Die Lambda-Architektur

Eine konstruktiv nutzbare Vorlage für Konzeption und Entwurf einer Big-Data-Anwendung ist der von Nathan Marz und James Warren publizierte Ansatz der „Lambda-Architektur“ [MaW13]. Die in

der Architektur vorgesehene Modularisierung spiegelt typische Anforderungen an Big-Data-Anwendungen wider und systematisiert sie. Der Architektur-Ansatz hilft, technische und nicht-funktionale Anforderungen an neue Anwendungen aufzudecken und zu beurteilen – unabhängig davon, in welcher Form und in welchem Umfang die Module als technische Komponenten der Anwendung realisiert werden. Notwendige Leistungen können identifiziert und eine geeignete Auswahl von Komponenten begründet werden.

In eine Anwendung gehen Datenströme ein, z. B. Daten aus der Nutzung und Bedienung einer Anwendung oder Sensordaten technischer Systeme. Die Daten werden protokolliert und führen zu einem wachsenden Datenvolumen.

In der Lambda-Architektur werden original eingestellte Daten mit einem Zeitstempel versehen und ohne Informationsverlust aufgezeichnet. Im „Data Storage" steht so der gesamte und wachsende Originaldatenbestand zur Verfügung. Alle aufgezeichneten Informationen können in neuen, korrigierten oder erweiterten Funktionen berücksichtigt werden (siehe Abbildung 2).

In der Architektur werden zwei Ebenen unterschieden:

- Die **Batch-Ebene** prozessiert alle gesammelten Originaldaten in der Batch-Funktion und bereitet sie zur Präsentation der Berechnungsergebnisse im Batch View auf. Batch-Prozesse werden zyklisch oder bei Bedarf wiederholt. Dies ist einem ETL-Prozess vergleichbar, der ein Data Warehouse mit Daten füllt, die für Onlineanalysen (OLAP) vorbereitet und optimiert sind.
- In der **Speed-Ebene** werden in einkommende Daten unmittelbar und so schnell als möglich prozessiert und für die Präsentation in der Anwendung aufbereitet. Die Speed-Ebene überbrückt die Batch-Laufzeiten und die damit verbundenen Verzögerungen zwischen dem Eintreffen von Daten und deren Berücksichtigung im Batch View.

Eine Übersicht über die Komponenten und ihre Leistungen sowie über die Anforderungen und Aspekte der Skalierung enthält die Tabelle (vgl. auch [Bit14] und Tabelle).

Anmerkungen

Eine Anwendung muss nicht unter allen Aspekten skalierbar sein. Beispielsweise erfordert ein technischer Leitstand zwar ein umfangreiches, wachsendes Datenvolumen und hohe Verarbeitungsleistung zur Analyse und Aufbereitung gespeicherter und ankommender technischer Datenströme; die Anzahl unabhängiger Arbeitsplätze bleibt aber gering.

Grundsätzlich werden in jedem Ablauf des Batch-Prozesses alle Daten verarbeitet. Bei Änderung der Algorithmen ist das notwendig der Fall. Anwendungsspezifisch können neue Daten auch inkrementell verarbeitet werden. Die Gewichtung von Batch- und Speed Layer und die jeweiligen Funktionen können anwendungsspezifisch variieren.

Konstruktion der Batch- und der Speed-Funktion

Wir zeigen beispielhaft die Konstruktion der Batch- und Speed-Funktion an einer Anwendung, die wir mit unseren Kollegen im Fraunhofer IAIS zu Demonstrationszwecken und als durchgängiges Beispiel für das Schulungsprogramm zum Data Scientist [IAI] entwickelt haben. Ziel der gesamten Anwendung ist es, Beiträge zu einem Web-Forum auf technische Terme und Emotionen wie Freude, Sorge oder Ärger zu untersuchen. Die Analyseergebnisse werden dem Analysten aggregiert und aufbereitet zur interaktiven Exploration in Form eines Web-Dashboard zur Verfügung gestellt. Abbildung 3 zeigt die über einen Web-Service realisierte grafische Schnittstelle.

In unserem Beispiel haben wir prototypisch Beiträge aus dem Motortalk-Forum [mot], dem größten Europäischen Web-Forum zum Thema Automobil, anonymisiert auf Emotionen und technische Terme hin analysiert [emo]. Jeder analysierte Beitrag bezieht sich auf einen bestimmten Autotyp, der wiederum einer Automarke zugeordnet ist. Die Schnittstelle ermöglicht Abfragen nach Zeitraum und Automarke und zeigt die gefundenen Ergebnisse aggregiert nach Autotyp und getrennt nach Emotion an (vgl. Abbildung 3, Kuchendiagramme in der Mitte).

Des Weiteren werden der zeitliche Ver-

Abb. 2: Architektur-Komponenten für Big Data

Komponenten	Leistungen	Anforderungen und Aspekte der Skalierung
Neue Daten	■ Eintreffende Daten werden an die Batch- und die Speed-Ebene gesandt, ■ versehen mit einem Zeitstempel und mit einem global eindeutigen Identifikationsmerkmal	■ Redundanz gegen Datenverlust durch Rechnerausfall, ■ Pufferung von hohem Datenaufkommen im Vergleich zur durchschnittlichen Datenrate, ■ Skalierbarkeit bezüglich der Puffergrößen und Zeiträumen der Aufbewahrung im Message-System.
Data Storage	■ Die Originaldaten sind unveränderlich gespeichert. ■ Neue Daten werden der Originaldatenmenge angefügt.	■ Partitionierung und Verteilung der Daten zwecks Verarbeitung im MapReduce-Prozess des Batch Layer, ■ Skalierbarkeit bezüglich der Datenmenge und der redundanten, ausfallsicheren Speicherung.
Batch Function	■ Die Batch-Funktion berechnet die Informationen für das Batch-View anhand des Originaldatenbestandes nach dem MapReduce-Verfahren für große Datenbestände.	■ Der Algorithmus muss auf Basis der verteilten Datenspeicherung parallelisierbar sein. ■ Skalierbarkeit bezüglich der Verarbeitungsleistung.
Batch View	■ Die Ergebnisse der Batch-Funktion werden in Read-Only-Datenbanken gespeichert und für schnellen Zugriff indiziert. ■ Die Daten sind auch bei verteilter, replizierter Speicherung konsistent. ■ Inkonsistenz oder Nicht-Verfügbarkeit ist auf die Dauer des Umschaltens zwischen den Datenversionen beschränkt.	■ Schneller, möglichst über alle Knoten atomarer Übergang zu einer neuen Version aller Daten im Batch View, ■ schnelle Reaktionszeiten für anwendungsspezifische Queries, ■ Skalierbarkeit bezüglich der Datenmenge und des Anfrageaufkommens.
Speed Function	■ Die einkommenden Daten werden so schnell verarbeitet, dass der zeitliche Verzug zwischen Verfügbarkeit der Daten und ihrer Berücksichtigung in der Anwendung vertretbar ist..	■ Skalierbarkeit bezüglich der ankommenden Datenrate und der Verarbeitungsgeschwindigkeit
Realtime View	■ Die von der Speed-Funktion berechneten Daten werden in einer Lese-Schreib-Datenbank gespeichert. ■ Der notwendige Umfang ist bestimmt durch die Menge anfallender Daten im Datenstrom für die Zeitdauer von zwei Verarbeitungsläufen im Batch-Layer. Ankommende Daten sind spätestens nach dieser Zeit im Batch View berücksichtigt und davon abgeleitete Daten im Realtime View können entfernt werden.	■ Zeiträume der Inkonsistenz bei Aktualisierung der verteilten Daten (eventual consistency) sind zu minimieren. ■ Schnelle Reaktionszeiten für anwendungsspezifische Queries, ■ Skalierbarkeit bezüglich der Datenmenge, der Lese-Rate und bezüglich der Schreib-/Update-Rate der von der Speed Function produzierten Daten.
Anwendung	■ Die Anwendung wickelt Anfragen ab und kombiniert dabei die Daten des Batch View und des Realtime View..	■ Zusammenführung und Abgleich der Informationen vom Batch und vom Realtime View, ■ Skalierbarkeit bezüglich der Nutzung durch gleichzeitige, unabhängige Anfragen.

Tab.: Lambda-Architektur – Komponenten, Leistungen, Anforderungen

lauf und die räumliche Verteilung der Beiträge dargestellt. Die Auswahl kann unter verschiedenen Aspekten gruppiert und weiter eingeschränkt werden, um dann ggf. bis zu den einzelnen Texten der Forumsbeiträge vorzustoßen (sogenannter „Drill-Down“). In den Texten sind die jeweiligen mittels Text-Mining erkannten emotionalen Passagen und technischen Terme markiert.

Die gezeigte Web-Schnittstelle entspricht im Rahmen der oben besprochenen Lambda-Architektur der „Anwendung“ ganz rechts, die Informationen aus Batch- und Realtime-View zusammenführt. Sie wird realisiert über einen Web-Service, der auf die mit den gefundenen Emotionen annotierten Forumsbeiträge in den NoSQL-Datenbanken zugreifen kann. Als konkrete Komponente für den Batch View haben wir im Beispiel „Voldemort“ und für den Speed View „Cassandra“ ausgewählt. Beides sind horizontal skalierbare Key-Value-Stores (eine spezielle Variante von NoSQL-Datenbanken), wobei Voldemort lese- und Cassandra schreiboptimiert ist.

Eine weitere Anforderung an die Anwendung ist es, dass zum einen große Mengen von historischen Daten analysiert

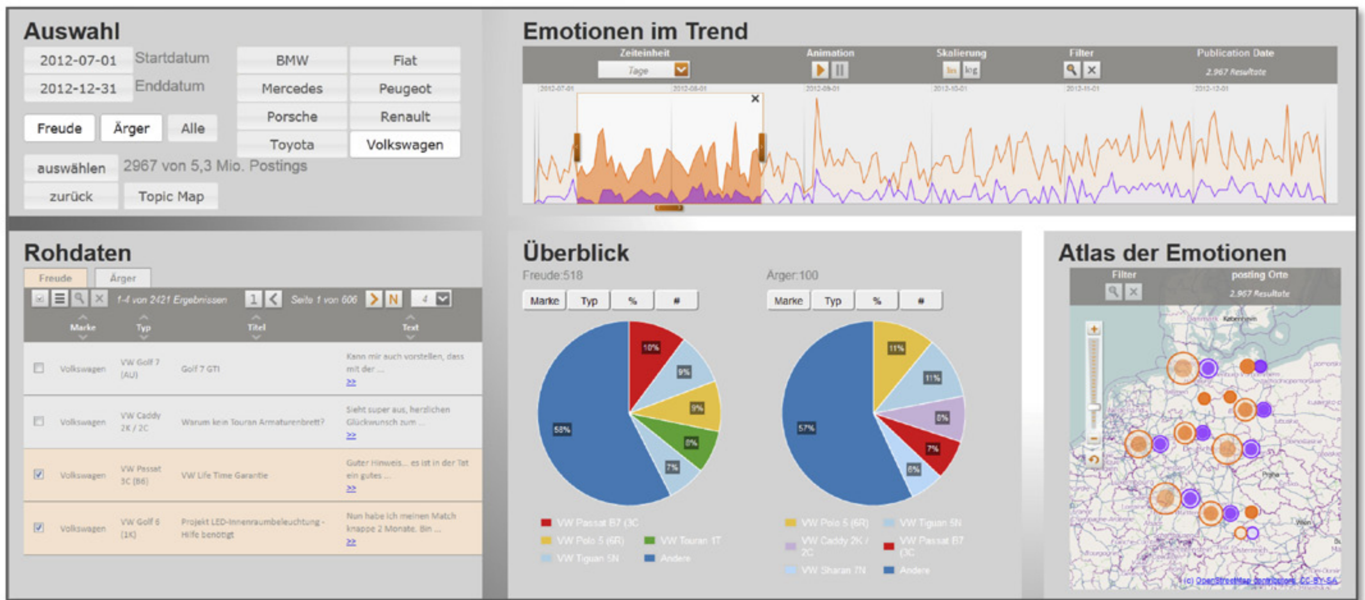

Abb. 3: Anwendungs-Dashboard zur Darstellung und Analyse von Emotionen in Forumsbeiträgen

werden müssen – die Forumsbeiträge der letzten Jahre – und zum anderen zeitnah auch Emotionen aus ganz aktuellen Beiträgen angezeigt werden sollen. Dies entspricht auf ganz natürliche Weise der Aufteilung in Batch- und Speed Layer der Lambda-Architektur.

Im Batch Layer werden die großen Datenmengen der historischen Daten analysiert, während im Speed Layer neu eintreffende Beiträge zeitnah analysiert werden können. Als Komponenten für die Ausführung der Batch- und der Speed-Funktion haben wir „Hadoop“ mit „cascading“ beziehungsweise „Storm“ gewählt. Die Verteilung der eintreffenden Daten in das Hadoop-File-System (HDFS) und in die Verarbeitung durch „Storm“ erfolgt über das Messaging-System „Kafka“.

Workflows

In dem von uns realisierten Beispiel wird im Batch- und im Speed Layer exakt die gleiche Analyse ausgeführt: In den einzelnen Forumsbeiträgen werden durch ein Textmining-Verfahren Emotionen und technische Terme gesucht und annotiert. Die Verarbeitung erfolgt in mehreren Schritten: Eine Webseite des Forums wird zunächst in verschiedene Forumsbeiträge zerlegt. Jeder Beitrag wird dann einzeln analysiert und annotiert.

Des Weiteren werden Metadaten wie Datum oder Automarke und -typ extrahiert, die zur Ablage der Beiträge in den NoSQL-Datenbanken des Batch- und der Realtime View verwendet werden. Die Umsetzung derartiger Schritte erfolgt am einfachsten in einem Workflow-Konzept, in dem einzelne Arbeitsschritte über den Datenfluss getrieben ausgeführt werden.

Im Speed Layer bietet Storm als Systemschnittstelle das Konzept der „Topology“ an, das im Grunde einen durch den Datenfluss getriebenen Workflow darstellt. Hadoop hingegen unterstützt unmittelbar nur das MapReduce-Konzept. Dieses organisiert die Datenflüsse zwischen Map (Abbildung) und Reduce (Reduktion) -Funktionen. Allerdings ist es üblich, die Batch-Funktion in einer höheren Abstraktion zu definieren und dann in eine MapReduce-Konfiguration zu übersetzen. Das von uns beispielhaft gewählte System „Cascading“ [Cas] leistet genau dies und ermöglicht die Konstruktion der Anwendung als Workflow, der im Effekt dem Workflow einer Storm Topology komplett oder in Teilen strukturgleich ist. Abbildung 4 zeigt die gleichartige Konstruktion des Batch- und des Speed Layers in unserem Beispiel.

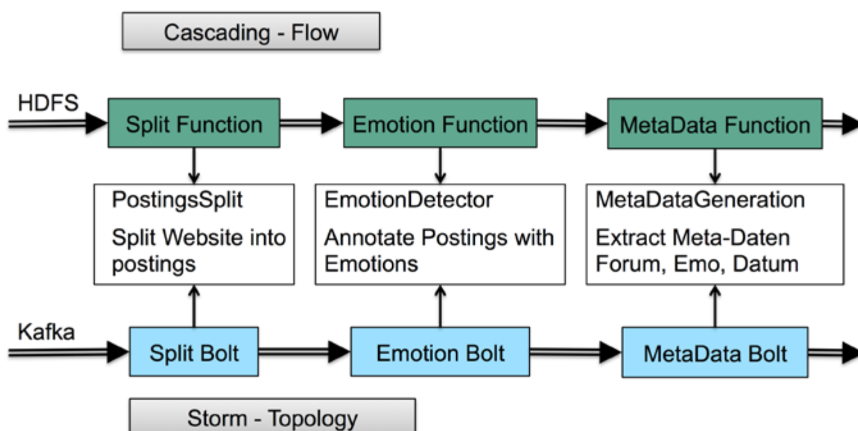

Abb. 4: Datenfluss zur Emotionsanalyse als Workflow und als Topologie

Sowohl die Storm Topology als auch der Cascading Workflow rufen Java-Methoden auf, in denen die eigentliche Anwendungsfunktionalität gekapselt ist. Storm und Cascading/Hadoop dienen somit nur als treibende Kraft, die die parallele Ausführung der Analysefunktion im Cluster, gesteuert durch den Datenfluss, organisiert und überwacht. Die grün und blau hervorgehobenen Elemente sind sehr leichtgewichtige Adapter zur Integration der Funktionen in das jeweilige Ablaufsystem.

Jeder einzelne Funktionsschritt kann in den Grenzen der gegebenen Rechner-Hardware im Cluster beliebig oft parallel ausgeführt werden. Die gesamte Rechenkapazität des Clusters wird automatisch genutzt, ohne dass die Anwendung explizit parallelisiert werden muss. Die Ergebnisse der Analyse werden dann in den

ebenfalls verteilten Big-Data-Datenbanken gespeichert, sodass sie vom Webservice des Anwendungs-„Frontend“ effizient abrufbar sind.

Nach Abbildung 4 wird die Implementierung der einzelnen Funktionen von den beiden Workflows gemeinsam genutzt. Aber die Workflows selbst müssen für Cascading und Storm, obwohl strukturgleich, separat und quasi in Kopien definiert werden. Die Situation ist typisch: Batch- und Speed-Funktion arbeiten komplett oder in Teilen nach dem gleichen Workflow. Mehrfache Implementierung des gleichen Konzepts ist ein softwaretechnischer Makel, lästig und änderungsunfreundlich. Das „Summingbird“-System [Sum] geht hier z. B. einen Schritt weiter: Workflows werden in einer kompakten domänenspezifischen Sprache (DSL) einmal definiert und für die jeweiligen Plattformen geeignet übersetzt.

Zusammenfassung und Ausblick

Big-Data-Systeme sind horizontal skalierbar und ermöglichen somit die Verarbeitung von großen Datenmengen in kurzer Zeit. Big-Data-Anwendungen profitieren von einer Vielzahl von Automatismen, die in den Big-Data-Systemen realisiert werden. Dafür müssen sie sich jedoch auf die Schnittstellen beschränken, die von den Big-Data-Systemen angeboten werden. Da es im Open-Source-Bereich eine große Reihe von Big-Data-Komponenten gibt, die alle verschiedene Schwerpunkte und Schnittstellen haben, ist eine auf die Anforderungen der Anwendung abgestimmte Auswahl und Komposition der Systemkomponenten unbedingt notwendig.

Eine konstruktive Vorlage zum Treffen einer solchen Auswahl ist die Lambda-Architektur. Diese unterscheidet zwischen einem Batch Layer für die Verarbeitung von großen Datenmengen und einem Speed Layer für die zeitnahe Verarbeitung von Datenströmen. Beide Layer erzeugen Analyseergebnisse, die in skalierbaren Big-Data-Datenbanken abgelegt werden. Ein Anwendungsservice führt die Ergebnisse aus Batch- und Speed Layer zusammen und stellt sie dem Anwender zur Verfügung.

Ausblick: Die dargestellten Big-Data-Ansätze gehen immer noch davon aus, dass alle Daten, sei es im Batch- oder im Datenstrom, in irgendeiner Form in einem oder mehreren Clustern zentralisiert werden. Denkt man an zukünftige Anwendungen, in denen Daten kontinuierlich von großen Mengen von räumlich verteilten Sensoren erzeugt werden, wird dies nicht mehr möglich sein (z. B. in der Produktion oder in großen Netzwerken).

Aktuelle Forschungsprojekte, wie die EU-Projekte FERARI [fer] und INSIGHT [ins] beschäftigen sich damit, logische Teile der Gesamtanwendung bereits in der Sensorik abhandeln zu können, sodass eine vollständige Zentralisierung aller Daten nicht mehr benötigt wird. Damit könnten dann zum Beispiel zeitnah Teilprobleme in großen Telekommunikationsnetzwerken oder frühzeitig Warnhinweise in Katastrophenszenarien erkannt werden. ■

Referenzen

[Vog09] Werner Vogels, „Eventually Consistent“, Communications of the ACM 2009,lvoL. 52, no. 1, Doi:10.1145/1435417.1435432

[Bre00] E. Brewer, „Towards Robust Distributed Systems,“ Proc. 19th Ann. ACM Symp.Principles of Distributed Computing (PODC 00), ACM, 2000, pp. 7-10; http://www.cs.berkeley.edu/~brewer/PODC2000.pdf

[Bit14] http://www.bitkom.org/files/documents/BITKOM_Leitfaden_Big-Data-Technologien-Wissen_fuer_Entscheider_Febr_2014.pdf

[DaG04] Jeffrey Dean und Sanjay Ghemawat (Google Inc.): MapReduce: Simplified Data Processing on Large Clusters, OSDI 2004

[MaW13] Nathan Marz und James Warren, “Big Data – Principles and best practices of scalable real-time data systems”, Manning Publications, 2013

[IAI] http://www.iais.fraunhofer.de/data-scientist.html

[emo] http://www.emotionsradar.com

[fer] http://www.ferari-project.eu, FP7, Grant No. 619461

[ins] http://www.insight-ict.eu, FP7, Grant. No. 318225

[mot] http://www.motortalk.de

[Sum] https://twitter.com/summingbird

[Ccd] http://www.cascading.org/

[Kaf] http://kafka.apache.org/ cascading

[Sto] https://storm.apache.org/

[Vol] http://www.project-voldemort.com/voldemort/

[Cas] http://cassandra.apache.org/