Alles Sensor

Datenverarbeitung mit Sensoren

Stefan Siprell

Die immer besser werdende Auswertung von immer größeren Datenmengen wird der Wettbewerbsfaktor der nächsten Jahre sein. Dazu ist es erforderlich, aus langen Reihen von beobachteten Ereignissen mit geringer Informationsdichte anwendbare Erkenntnisse mit hoher Informationsdichte zu generieren und entsprechend zu reagieren. Neue Tools und Architekturen sind dafür notwendig.

Deutschland bricht in die digitale Welt auf

Nach jedem Besuch beispielsweiser einer amerikanischen Messe fragt man sich als Entwickler, ob wir uns in Europa im digitalen Mittelalter befinden. Regularien, Datenschutzbestimmungen oder die gewonnene Exportweltmeisterschaft im Maschinenbau sind sicherlich nachvollziehbare Gründe hierfür. Aber auch die deutsche Tugend, einen perfekten Plan zu erstellen, bevor die erste Schraube angefasst wird, hat einen Einfluss auf die konservative deutsche IT. Während also die amerikanischen Kollegen mit den neusten Technologien gute und schlechte Erfahrungen sammeln konnten, arbeitet der gemeine Deutsche weiterhin mit RDBMS-Systemen und JEE-Servern.

Seit letztem Jahr ist allerdings gefühlt sehr viel Bewegung in der deutschen IT-Szene. DAX-Konzerne gehen in die Public Cloud, Automobilhersteller erkennen den Wert des Siliziums im Blech und die Versicherungsbranche nutzt Big Data. Die heimische Industrie hat offensichtlich die Warnsignale vernommen und reagiert [Dör16].

Die ersten IT-Systeme haben komplizierte Vorgänge in Silos automatisiert. Die Enterprise-Welt von heute befasst sich mit komplexen Vorgängen, die mehrere Silos orchestrieren beziehungsweise choreografieren. EAI, ESB, BPM, SOAP und Co. waren die Schlagworte auf Konferenzen, in der Literatur und in Marketing Claims der Toolhersteller. Manche Trends sind in neueren aufgegangen, andere sind wieder untergegangen – aber die Art der Projekte und der Tools war doch recht statisch.

Erfolg durch IT – nicht trotz IT

Allerdings wird die Digitalisierung alles für uns ändern. Arbeitsprozesse, Kunden-Unternehmen-Beziehungen, persönliche Gesundheit, Freundeskreis und Soziales, Fertigung usw. werden digitalisiert. Die Digitalisierung erzeugt Unmengen an Daten, welche wiederum analysiert werden müssen, um die jeweils bestmögliche Entscheidung automatisiert zu treffen. Big Data ist nicht mehr ein Nischenthema für amerikanische Startups – es ist ein neues und bleibendes Thema.

Die immer besser werdende Auswertung von immer größeren Datenmengen wird der Wettbewerbsfaktor der nächsten Jahre sein. Vor diesem Hintergrund ist dieser Artikel entstanden – ein Versuch, das Handwerkzeug für aktuelle IT-Probleme zu beschreiben. Es werden einige Tools und Handgriffe vorgestellt – aufgrund der Breite des Themas und Kürze des Artikels muss in Folgendem auf Codebeispiele verzichtet werden.

Sensordaten und Fast Data

Sensordaten werden die Standarddatenquelle der Zukunft sein. Bisher hat die IT stets Transaktionen und Stammdaten abgebildet:

- Die Kundin hat ein Produkt.
- Der Zählerstand ist erfasst worden.
- Das Geld wurde auf das Konto überwiesen.

Die Auswirkungen der Transaktion haben die Architekten dann in den IT-Systemen abgebildet – häufig in einer normalisierten Form [4NF] mit einer relationalen Datenbank. Die Anzahl der Geschäftsobjekte war stets überschaubar – selten über den zweistelligen Millionenbereich hinaus – und die Informationsdichte war sehr hoch.

Transaktionen und eben auch Stammdaten sind weiterhin sehr wichtig, allerdings werden in der Digitalisierung Beobachtungen genutzt, um eben mehr oder bessere Transaktionen abzuschließen:

- Welche Interaktionen hatten wir mit dem Kunden, der gerade im Support anruft? Möchte er womöglich kündigen?
- Wie war der Fahrstil des Versicherungskunden, der gerade einen Schaden gemeldet hat?
- Welche Produkte haben ähnliche Kunden in der Vergangenheit gekauft?

Beobachtungen unterscheiden sich in einigen Punkten von Transaktionen:

- Sie passieren viel häufiger – ein Kunde kann sich Hunderte von Produkten anschauen, bevor er sich entscheidet, doch nichts zu kaufen.
- Sie sind ungenau und haben einzeln eine sehr geringe Informationsdichte – ein Kunde kann verschiedene Browser oder mehrere Kunden können einen Browser nutzen.
- Sie können unvollständig sein – speziell im IoT-Umfeld können Sensoren ausfallen oder sich temporär nicht mit dem Internet verbinden.
- Sie passieren kontinuierlich – auch nach der Transaktion wird der Strom an Beobachtungen nicht abbrechen.
- Sie sind flach – einzelne Ereignisse reihen sich wie Perlen an einer Kette und haben zunächst nicht die tiefe Vernetzung, wie sie relationale Datenbanken besitzen.

Visualisierung

Zeppelin [Zeppelin] ist ein tolles Konzept: Man kann kleine Spark-Jobs, Cassandra-Abfragen oder Ähnliches direkt im Browser schreiben und gegen die jeweiligen Backends abfeuern. Die Ergebnisse lassen sich als einfache Diagramme oder auch Tabellen darstellen. Sobald man aber die dahinter liegenden Backends (zum Beispiel Spark) für eine neue Funktion aktualisiert, fängt der beschriebene Check-out/Build/Deploy-Zyklus von vorne an und man hofft auf möglichst wenig breaking changes.

Daher verwenden wir immer wieder Elasticsearch und Kibana. Es ist sehr stabil und kann ebenfalls direkt aus Spark genutzt werden. Mit Kibana kann man intuitiv eigene Aggregationen und Analysen definieren und auch mit Heatmaps visualisieren – eine Funktion, die noch in Zeppelin fehlt. Im Unterschied zu Zeppelin, welches Spark-Jobs startet und das Ergebnis visualisiert, werden hier die Ergebnisse vom Spark-Job in Elasticsearch gespeichert und anschließend mit Kibana visualisiert.

- Sie wollen interpretiert werden – aufgrund der genannten Punkte obliegt es dem Entwickler zu entscheiden, ob und wie die Beobachtung relevant ist.

SMACK-Stack

Um aus den langen Reihen von beobachteten Ereignissen mit geringer Informationsdichte anwendbare Erkenntnisse mit hoher Informationsdichte zu generieren und entsprechend zu reagieren, sind neue Tools und Architekturen notwendig.

Ein Beispiel für solch eine Architektur ist der SMACK-Stack. Der Name entstand in Anlehnung an den LAMP-Stack und beschreibt eine Architektur bestehend aus den folgenden Tools:

- Spark: Datenverarbeitungstool,
- Mesos: Verteilter Scheduler für große Cluster,
- Akka: Actor-Framework und Runtime,
- Cassandra: Datenbank für große Mengen an strukturierten Daten,
- Kafka: Hochperformantes Messaging-System.

In [Wamp15] ist eine komplette Beschreibung der Architektur zu finden. In der Kurzfassung liefert Kafka eine verlässliche Möglichkeit, eine hohe Anzahl an Nachrichten zu verarbeiten. Dies beinhaltet auch die Möglichkeit, Milliarden von Nachrichten mehrere Tage nicht zu verarbeiten oder die Verarbeitung zu wiederholen.

Die Nachrichten kann man in der Regel mit Spark verarbeiten, um eine hohe Anzahl von Nachrichten in hochverdichtete Aussagen zu komprimieren. Zeitreihen, berechnete Aussagen und Co. können nicht nur flüchtig in Spark gehalten werden, sondern können in großer Anzahl über lange Dauer in Cassandra gespeichert werden.

Eintreffende Nachrichten können in Akka mit dem Kontext aus Spark und Cassandra verglichen werden, um direkt mit entsprechenden Anweisungen zu reagieren. Um den hohen Fluktuationen der Last gerecht zu werden, kann Mesos genutzt werden, um automatisiert und lastgerecht die Maschinen im Rechenzentrum einzusetzen. Zudem können sowohl Spark als auch Akka sinnvoll mit [BackPressure] umgehen.

Allerdings werden im Folgenden nur Cassandra und Spark untersucht, um die Zusammenarbeit der Module innerhalb der Architektur zu demonstrieren.

Cassandra

Cassandra wurde bereits in mehreren JavaSPEKTRUM-Artikeln vorgestellt [Wies10,GhJa15]. Diese NoSQL-Datenbank kann sehr schnell Daten speichern und laden, unterstützt allerdings nur flache Strukturen – Joins, Aggregationen und ähnliches sind, wenn überhaupt, nur sehr beschränkt möglich.

Um verschiedene Abfragen mit entsprechenden Zielen, zum Beispiel aggregiert oder nach unterschiedlichen Schlüsseln, zu ermöglichen, ist man gezwungen, die Daten denormalisiert abzulegen. Beim Speichern der Daten muss bereits bekannt sein, wie sie später abgefragt werden. Für jeden Abfragetyp wird eine eigene Tabelle mit redundanten Daten gepflegt.

Mit Spark können Daten transformiert, aggregiert und referenziert werden. Spark ist von Haus aus verteilt und kann Daten in Partitionen auf verschiedenen Servern verarbeiten. Spark unterstützt Datenlokalität für Cassandra, sodass jede Spark-Partition nur lokale Daten aus Cassandra auf dem Knoten lädt und verarbeitet. Da die Operationen zu den Daten wandern und nicht umgekehrt, können solche Operationen unheimlich gut skalieren – die Anzahl der Daten pro Maschine ist der limitierende Faktor und nicht mehr die Gesamtmenge der Daten.

Um das Ganze zu demonstrieren, kann man sich die „Microsoft Taxi Traversal"-Daten [Zheng11] anschauen. Innerhalb von sechs Tagen wurden 9 Millionen gefahrene Kilometer von über 10 000 Taxis aufgezeichnet. Regelmäßig wurde pro Taxi (bis zu 60-mal pro Stunde) die aktuelle GPS-Koordinate aufgezeichnet. Als Beispiel ein Ausschnitt aus einer der CSV-Dateien:

```
taxi id, date time, longitude, latitude
1,2008-02-02 15:36:08,116.51172,39.92123
1,2008-02-02 15:46:08,116.51135,39.93883
1,2008-02-02 15:46:08,116.51135,39.93883
1,2008-02-02 15:56:08,116.51627,39.91034
1,2008-02-02 16:06:08,116.47186,39.91248
1,2008-02-02 16:16:08,116.47217,39.92498
1,2008-02-02 16:26:08,116.47179,39.90718
1,2008-02-02 16:36:08,116.45617,39.90531
```

Möchte man nun pro Straße die durchschnittliche Geschwindigkeit zu verschiedenen Uhrzeiten berechnen, sind folgende Schritte notwendig.

- Speichern der Daten: Die Daten werden nahezu identisch zum ursprünglichen Format gespeichert. Die Taxi-ID wird als Primärschlüssel für die Zeile genommen und die Spaltengruppe DateTime, Longitude, Latitude wird pro Sensormeldung wiederholt.
- Herausfiltern redundanter Sensormeldungen: Mit Spark wird jede Zeile komplett eingelesen und alle Spaltengruppen werden analysiert. Diese Vorgehensweise wird sich wiederholen und ist genau eine der Stärken von Spark und Cassandra. Spark lädt jede Zeile aus der jeweils lokalen Cassandra-Instanz und iteriert dann sehr schnell durch die Spalten. Jede Spaltengruppe, welche die gleiche Koordinate des Vorgängers hat, wird herausgefiltert. Alle erhaltenen Spaltengruppen werden in eine neue Tabelle gespeichert. Die Dauer

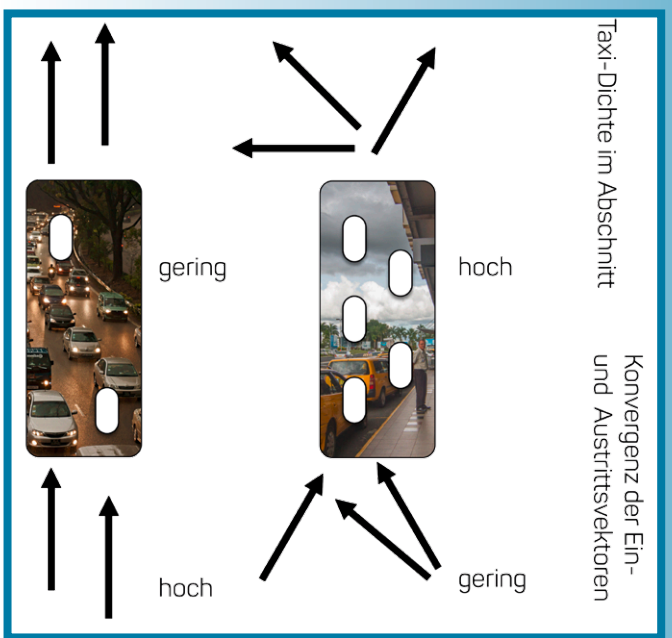

Abb. 1: Aus den Sensordaten ist der Unterschied nicht unmittelbar zu erkennen: Stehen die Autos im Stau oder warten sie auf den nächsten Fahrgast? Höherwertige Auswertungen geben neue Indizien. Kommen die Taxis aus unterschiedlichen Richtungen, warten dicht gepackt aneinander und verteilen sich danach wieder, liegt die Vermutung nahe, dass die Taxis hier Gäste abgeben und aufnehmen (l. Foto: epSos.de, r. Foto: Maksym Kozlenko – Wikimedia Commons)

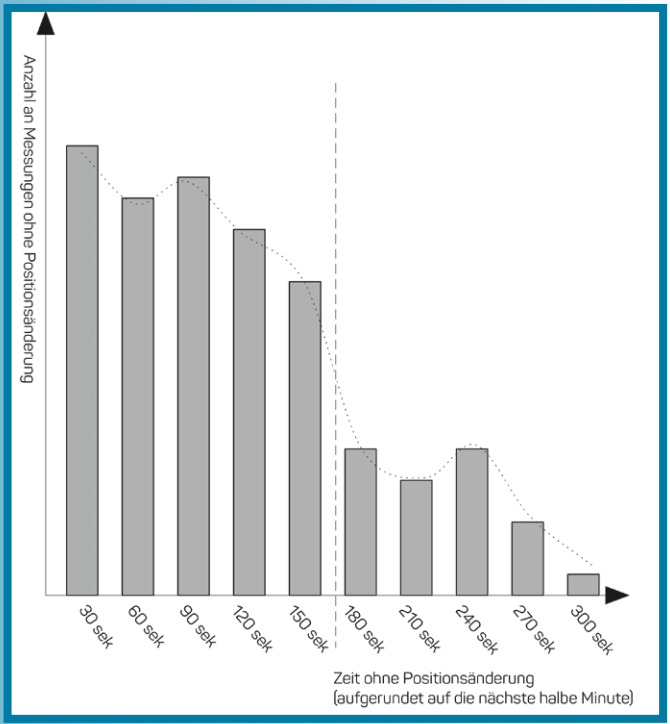

Abb. 2: Fiktives Histogramm: Nach 150 Sekunden nimmt die Anzahl der Messungen rapide ab, daher nehmen wir 150 Sekunden als Schwellwert an. Wir nehmen an, dass Taxis, welche weniger als 150 Sekunden stehen, im Stop&Go, vor einer Ampel oder einer Kreuzung stehen

zum vorherigen Eintrag pro Fahrzeug wird als zusätzliche Spalte in die Spaltengruppe gespeichert.

Der Vorgang ist sehr effizient, da für die Operation pro Zeile nur drei floats im Speicher gehalten werden müssen und immer nur zwei Zeilen verglichen werden. Durch die Reduktion der Daten werden auch spätere Operationen schneller.

- Touren berechnen: Da parkende Taxis in der Regel nicht direkt auf der Straße stehen, ist es sinnvoll, nur fahrende Autos für die Verkehrsdichte zu analysieren. Da die Daten keine Auskunft darüber geben, ob ein Auto im Stau oder vor einer Ampel steht oder gerade parkt, müssen diese interpretiert werden.

In dem Beispiel haben wir einen Schwellwert bestimmt und nur Einträge, welche diesen Wert überschreiten, leiten eine neue Tour ein. Alternativ könnte auch die Durchschnittsgeschwindigkeit aller Fahrzeuge in der Nähe zur gleichen Zeit betrachtet werden. Sinkt dies global gegen 0, kann also von einem Stau ausgegangen werden – alternativ auch von einem Flughafen-Terminal, bei dem tatsächlich viele Touren anfangen und aufhören.

Genau diese Art von Problem taucht im Umfeld von IoT und Big Data immer wieder auf. Wie kann man aus einer Reihe von Beobachtungen die korrekte Erkenntnis ableiten?

In der Regel kann aus weiteren Analysen ein Entscheidungsbaum [GraphHopper] stetig erweitert werden, um bessere Ergebnisse zu erzielen. Man kann zum Beispiel die Vektoren von Start- und Endpunkt der Touren vergleichen oder die Dichte der Fahrzeuge zur gegeben Zeit messen. Fahren die Fahrer nach dem Halt in eine andere Richtung oder sammeln sich besonders viele Taxis an einer Stelle, kann es ein Flughafenterminal, Bahnhof, eine Messe oder ähnliches sein.

Die Auswahl solcher Algorithmen ist sehr kreativ und explorativ und profitiert ungemein von einem guten Tooling. Man möchte die Thesen schnell definieren und verifizieren. So kann man manuell prüfen, ob der Terminal korrekt gefunden wurde. Leider ist das eines der aktuellen Schwächen des Stacks – mehr dazu später.

Weiter mit unserem Beispiel. Hier möchten wir ein Histogramm generieren und die Dauer zwischen den Meldungen in 30 Sekunden-Buckets ablegen und auswerten. Wir haben „willkürlich" einen Wert genommen, bei dem das Histogramm stark abgeflacht ist. Auch das kann knapp als Map/Reduce-Job geschrieben werden und wird lokal ausgeführt.

Mit einem neuen Spark-Job legen wir wieder die Zeilen aus der zuletzt angelegten Tabelle an. Sobald der festgelegte Wert überschritten ist, legen wir in einer neuen Tabelle eine neue Zeile an, sonst kopieren wir die Spaltengruppe in die angelegte Zeile. So haben wir pro Zeile eine Tour mit allen Messpunkten.

Berechnen der Straßenabschnitte

Selbstverständlich müssen die Daten auf Straßenabschnitte abgebildet werden. Pro Abschnitt (zwei aufeinanderfolgende Messpunkte in einer Tour) wird ein Routing ausgeführt.

Mit Tools wie Open Graph Hopper [GraphHopper] kann man sich eine Route zwischen zwei Koordinaten berechnen. Die Route beinhaltet Straßennamen, Straßenabschnitts-IDs, die Länge des zu befahrenden Abschnitts usw. Man kann Open Graph Hopper direkt in dem jeweiligen Spark-Prozess nutzen und die Touren-Tabelle analysieren.

In der Regel liegen zwischen 0 und 12 Minuten zwischen den Sensormeldungen oder im Schnitt etwa 600 Meter. Somit werden die reinen Sensordaten nicht genügend Datenpunkte liefern, um den Streckenverlauf und alle befahrenen Abschnitte zu erfassen. Die Route von Open Graph Hopper ist lediglich ein Vorschlag, das Taxi kann natürlich eine andere Route gewählt haben. Aber für diese Untersuchung sind die Annahmen akzeptierbar.

Für den Gesamtstreckenabschnitt sind die Dauer (zwischen Sensormeldungen) und Länge (berechnet durch Open Graph Hopper) bekannt, sodass die Zeitpunkte zwischen den Übergängen der Straßenabschnitte interpoliert werden können.

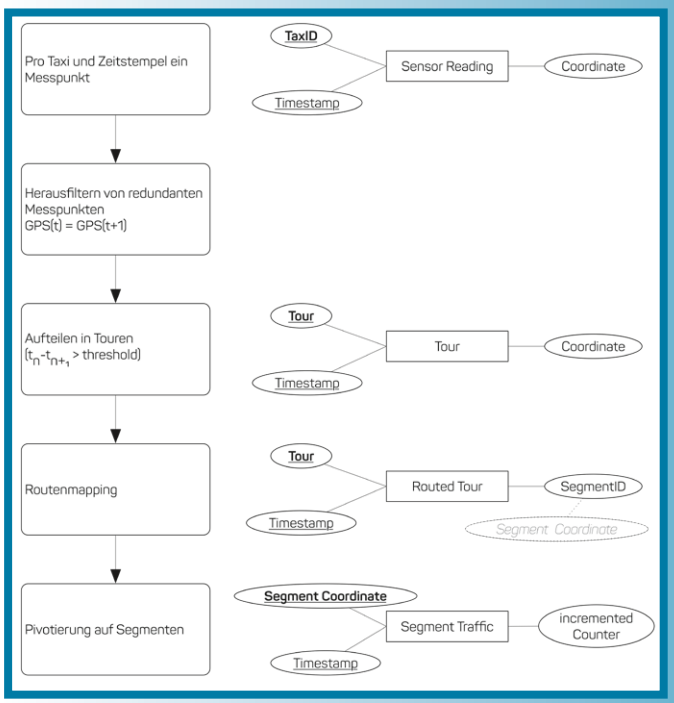

Abb. 3: Ablauf der Transformationen der Daten inklusive jeweiligem ERD-Diagramm (Partition Key, Clustering Key und Attributen)

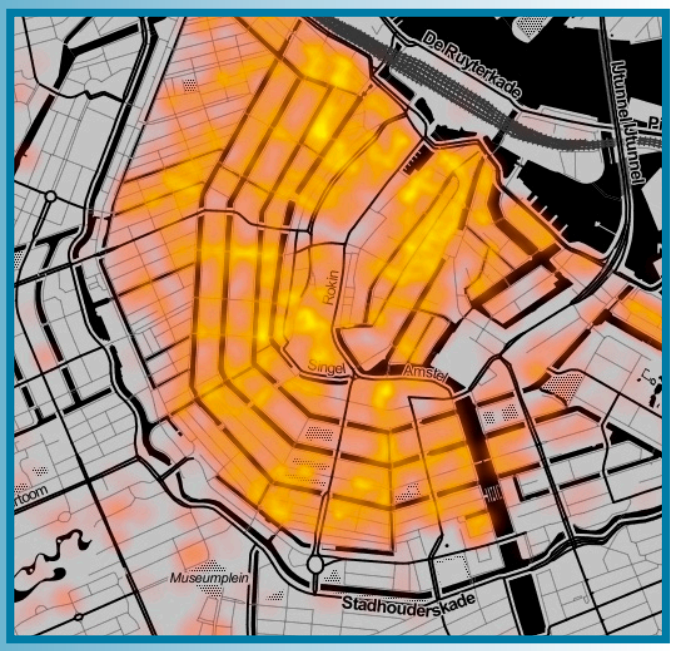

Abb.4: Beispiel einer Heatmap: Der Farbwert berechnet sich jeweils aus einer Aggregation wie Verkehrsdichte, Durchschnittsgeschwindigkeit usw. Foto: Multichill – Wikimedia Commons

Der Spark-Job wird also pro Tourenabschnitt eine Liste von Straßenabschnitten und deren Befahrungszeitpunkten berechnen. Von dem Straßenabschnitt wird die Geo-Koordinate berechnet, encodiert und als Primärschlüssel für eine Zeile verwendet. Nun wird lediglich der berechnete Zeitpunkt als Spaltenwert eingetragen.

Aus einer Tabelle mit einer Zeitserie von Koordinaten wird eine Tabelle erstellt, die pro Straßenabschnitt die Zeitpunkte der vorbeifahrenden Taxis auflistet.

Pivotierung der Daten

Hieraus eine Heatmap zu berechnen, ist nun einfach möglich. Man kann aus Performanzgründen noch die Messpunkte über Lokalität (Zusammenfassen nahe gelegener Straßenabschnitte) oder über Zeit (Bucketing auf Minutenebene oder ähnliches) aggregieren [McFa15].

Ausblick

Die Handgriffe sind stets dieselben. Man verarbeitet einzelne Nachrichten zu einer Kontext-Schicht, analysiert diese Schicht und baut die nächste Schicht darüber. In diesem Beispiel haben wir die Schichten explizit abgespeichert, diese explorativ analysiert und den nächsten Spark-Job für die folgende Schicht geschrieben.

Alternativ wäre es auch möglich, diese Schichten in weniger Schritten in Spark zu berechnen (das Refactoring ist durch das RDD-Konzept ein sehr einfaches Method-Chaining [RDD]). Das wäre für produktive Anwendungen deutlich schneller in der Berechnung, da man zum Teil auf die Zwischenergebnisse verzichtet. Insbesondere mit der Fähigkeit von Spark in Mikro-Batches direkt auf dem Datenstrom zu arbeiten, können verhältnismäßig gute Latenzen berechnet werden.

So spannend diese Technologien sind, muss man sich darauf einstellen, viel Zeit für das Einrichten der Tools zu benötigen. Selbst die Mikroversionen bringen zum Teil signifikante neue Funktionen mit oder beheben kritische Bugs. Da die Tools hoch integrativ genutzt werden, sind selten die neusten Versionen als Binary fertig integriert.

Der Spark-Cassandra-Connector kann nur bestimmte Versionen von Spark mit Cassandra verbinden, Spark ist wiederum von der darunter liegenden Scala- und Hadoop-Implementierung abhängig, Zeppelin ist von allem abhängig, usw. Man wird daher immer wieder Github-Projekte auschecken, Branches wechseln und eigene Releases bauen.

Auch wenn das Tooling noch große Sprünge macht und die Integration schwierig ist: Rom wurde nicht an einem Tag gebaut. Die Anwendungen laufen weltweit in großen Clustern und ohne diese wären Dienste wie LinkedIn, Netflix oder Apple nicht möglich.

Da der „Business Case“ der Digitalisierung und die abgeleitete Notwendigkeit von Big Data nicht von der Hand zu weisen sind, sollten sich Entwickler und Architekten dringend in die Themen einarbeiten. Zudem macht es einfach großen Spaß in großen Dimensionen zu denken und zu programmieren.

Links

[4NF] Vierte Normalform (4NF), `https://de.wikipedia.org/wiki/Normalisierung_(Datenbank)#Vierte_Normalform_.284NF.29`

[BackPressure] Back pressure in IT, `https://en.wikipedia.org/wiki/Back_pressure`

[CC] Wikimedia Commons, `http://creativecommons.org/licenses/by/2.0`, `http://creativecommons.org/licenses/by-sa/3.0`

[Dör16] St. Dörner, So chancenlos ist Deutschland in der digitalen Welt, WeltN24, 17.2.2016, `http://www.welt.de/wirtschaft/webwelt/article152341020/So-chancenlos-ist-Deutschland-in-der-digitalen-Welt.html`

[GhJa15] Ph. Ghadir, M. Jansing, Apache Cassandra, in: JavaSPEKTRUM, 02/2015, `http://www.sigs-datacom.de/uploads/tx_mwjournals/pdf/ghadir_jansing_JS_02_15_tJPH.pdf`

[GraphHopper] GraphHopper, `https://github.com/graphhopper/`

[McFa15] P. McFadin, 1.7.2015, `http://de.slideshare.net/patrickmcfadin/analyzing-time-series-data-with-apache-spark-and-cassandra`

[RDD] A Resilient Distributed Dataset (RDD), `https://spark.apache.org/docs/0.8.1/api/core/org/apache/spark/rdd/RDD.html`

[Wamp15] D. Wampler, Fast Data: Big Data Evolved, White Paper, Lightbend, 2015, `https://info.typesafe.com/rs/558-NCX-702/images/COLL-white-paper-fast-data-big-data-evolved.pdf?mkt_tok=3RkMMJWWfF9wsRokvq3KZKXonjHpfsX97e8tWrHr08Yy0EZ5VunJEUWy2oAHRdQ%2FcOedCQkZHblFnVkNQq27X7gNraUP`

[Wies10] L. Wieske, Cassandra – die NoSQL-Datenbank für große Mengen und schnelle Zugriffe, in: JavaSPEKTRUM, 06/2010, `http://www.sigs-datacom.de/uploads/tx_mwjournals/pdf/wieske_JS_06_10.pdf`

[Zeppelin] Apache Zeppelin, `https://zeppelin.incubator.apache.org`

[Zheng11] Yu Zheng, T-Drive trajectory data sample, 12.8.2011, `http://research.microsoft.com/apps/pubs/?id=152883`

Stefan Siprell ist bei der codecentric als Architekt tätig. Seine Schwerpunkte liegen in der Konzeption datengetriebener Anwendungen.
E-Mail: stefan.siprell@codecentric.de