OPITZ CONSULTING

Überraschend mehr Möglichkeiten

Lambda-Architektur in der Praxis

Das wichtigste Architektur-Konzept für Big Data am konkreten Beispiel

Whitepaper

Lambda-Architektur in der Praxis

Das wichtigste Architektur-Konzept für Big Data am konkreten Beispiel

Der Autor

Lukas Berle – Als Big Data Software Engineering Lead bei OPITZ CONSULTING beschäftigt sich Lukas Berle intensiv mit dem Thema Big Data. Dabei liegt sein Fokus insbesondere auf den Themen skalierbarer Anwendungen, Datenspeicherung, In-Memory Stream Processing Engines und deren Integration im Hadoop Ökosystem.

Kontakt

Lukas Berle
OPITZ CONSULTING Deutschland GmbH
Standort Nürnberg
Zeltnerstraße 3, 90443 Nürnberg
lukas.berle@opitz-consulting.com
+49 89 680098-0

Impressum

OPITZ CONSULTING Deutschland GmbH
Kirchstr. 6
51647 Gummersbach
+49 (0)2261 6001-0
info@opitz-consulting.com

Disclaimer

Inhalt

1 Einführung

Bei Diskussionen zum Thema Big-Data-Architekturen, dauert es meist nicht lange, bis der Begriff Lambda-Architektur in den Raum geworfen wird. Und tatsächlich ist die Lambda-Architektur eines der zentralsten Architekturkonzepte im Big-Data-Umfeld.

Die Lambda-Architektur beschreibt aus konzeptioneller Sicht den Aufbau einer Big-Data-Architektur, bestehend aus

- einem zentralen Eintrittspunkt in das Big-Data-System,
- einem Speed- und einem Batch-Verarbeitungslayer,
- sowie der Datenspeicherung.

In der Welt von Big Data existieren unterschiedliche Tools, mit denen sich eine solche Architektur basierend auf Open-Source-Technologien entwickeln lässt. Welches Tool sich am besten eignet, hängt dabei immer vom Einzelfall ab.

In diesem Whitepaper beschreibe ich den Einsatz einer Lambda-Architektur am Beispiel eines Automatenherstellers.

Viel Spaß beim Lesen!

Lukas Berle

2 Lambda-Architektur – Was ist das?

Bei der Verarbeitung von Datenströmen stellen sich häufig mehrere Herausforderungen: Zum einen müssen Daten in Echtzeit verarbeitet werden, um schnellstmöglich Reaktionen durch Technik oder Management in die Wege leiten zu können. Auf der anderen Seite erschwert die hohe Rechenkomplexität in Kombination mit großen Datenmengen die Verarbeitung in Echtzeit.

Als Lösung für dieses Problem stellte Nathan Marz vor einigen Jahren die Lambda-Architektur vor. [1][2]

Die Lambda-Architektur besteht aus zwei Kernkomponenten: Speed Layer und Batch Layer. Vorgeschaltet ist beiden Layern in der Regel ein Data Ingestion Layer, der für die Pufferung und die kurzfristige Wiederherstellbarkeit der Daten zuständig ist. Damit die Daten aus den Verarbeitungslayern auch angezeigt werden können, besitzen diese einen Serving Layer, der die Daten z. B. für ein Real-Time- oder Management Dashboard aufbereitet und bereithält.

Sobald Daten, die zum Beispiel von Maschinensensoren erzeugt werden, den Ingestion Layer erreichen, werden diese sowohl an den Speed Layer als auch an den Batch Layer weitergegeben.

Der Speed Layer verarbeitet die Daten direkt und optimaler Weise komplett „In-Memory“. Der Batch-Layer arbeitet die Daten dagegen zeitversetzt in bestimmten Intervallen ab.

2.1 Speed Layer

Ziel des Speed Layers ist es, nachgelagerten Systemen oder Endanwendern über einen Serving Layer möglichst aktuelle, aber nicht unbedingt exakte Zahlen zu liefern. Da die Datenmengen in einer solch hohen Geschwindigkeit möglicherweise nicht komplett verarbeitet werden können – und auch nicht müssen, ist es möglich, hier nur eine Teilmenge, z. B. jeden hundertsten Sensorwert, zu verarbeiten. Des Weiteren können Aggregationsfunktionen über definierte Zeitfenster (z. B. 15 Minuten) auf den Datenstrom angewendet werden.

2.2 Batch Layer

Der Batch Layer arbeitet in der Regel auf der gesamten Datenmenge. Seine Aufgabe besteht darin, exakte Ergebnisse zu liefern. Während der Speed Layer in der Regel alle Daten im Speicher halten muss, ist der Batch Layer dieser Restriktion nicht unterworfen und kann daher auch Analysen über lange Zeitfenster durchführen. Eine weitere Aufgabe des Batch Layers kann auch das Trainieren von statistischen Modellen aus dem Umfeld des Machine Learnings sein.

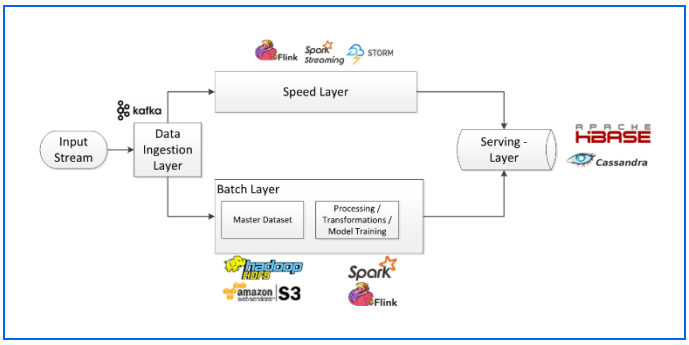

Abb. 1. Das Lambda-Architektur-Konzept

3 … und wie sieht das in der Praxis aus?

Ausgehend von dieser abstrakten Definition möchte ich an einem konkreten Anwendungsfall darstellen, wie eine Lambda-Architektur in der Praxis aussehen kann und welche Tools aus dem Hadoop-Ökosystem in einem solchen Fall verwendet werden können.

Unser Anwendungsfall ist ein Automatenhersteller, der seine Geräte (z. B. Snack-Automaten an Bahnhöfen) mit Hilfe einer IoT-Cloud verwaltet.

Jede Verkaufsaktion an den Automaten, aber auch jede Störung und sämtliche Sensorwerte (zum Beispiel die Temperatursteuerung) werden in unserem Use Case an ein Backend kommuniziert. Der Automatenhersteller kann den Bedarf seiner Produkte in bestimmten Regionen auf einem Live-Dashboard verfolgen und entsprechend unvermittelt handeln.

Darüber hinaus kann das Backend selbstständig Einsatzplanungen für Servicekräfte erstellen, um die Automaten zu befüllen. So werden die Servicekräfte bei kurzfristig ansteigenden Umsätzen an einem Automaten (z. B. bei Großevents wie Fußballspielen) automatisiert und direkt über den hohen Bedarf informiert. Sie können die entsprechenden Produkte daraufhin rechtzeitig auffüllen.

Technisch sieht das Ganze so aus: Das Backend wird durch einen Hadoop-Cluster realisiert. In diesem Cluster kommen verschiedene Tools des Hadoop Ökosystems zum Einsatz. Sie bilden gemeinsam eine Lambda-Architektur. (Abb. 2)

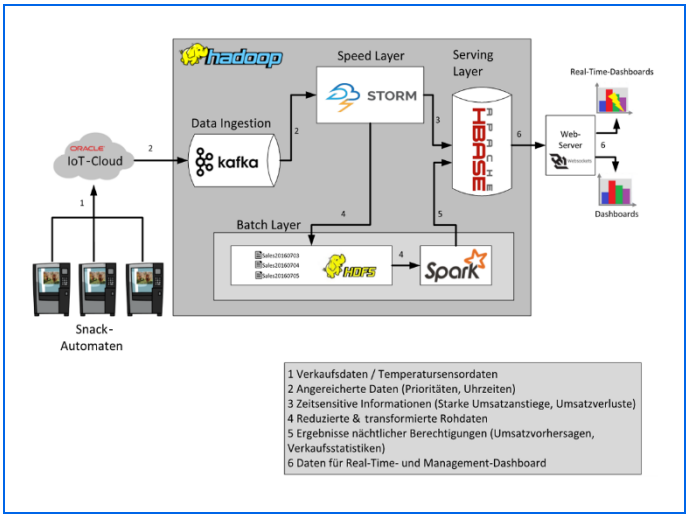

Abb. 2. Lambda-Architektur im Anwendungsfall eines Automatenaufstellers

3.1 Apache Kafka

Der Eintrittspunkt in das Hadoop-Cluster wird durch Apache Kafka realisiert. Kafka ist aufgrund seiner hohen Nachrichten-Durchsatzrate und der Möglichkeit, bereits abgerufene Datentupel eine bestimmte Zeit lang vorzuhalten, hervorragend geeignet.

Apache Kafka ist eine Grundvoraussetzung für die sichere Nutzung des Speed Layers. Denn durch die Persistenz der Daten in Kafka kann im Falle von Lastspitzen ein Datenverlust praktisch ausgeschlossen werden und Lastspitzen werden durch die Pufferung der Daten ausgeglichen. Im Gegensatz zu klassischen JMS-Lösungen ist Kafka von Hause aus hoch skalierbar. Des Weiteren sind die Daten im Kafka Topic (aus JMS-Sicht ist dieser eher eine Kombination aus Queue und Topic) redundant gespeichert.

Für den Speed Layer selbst bieten sich eine Reihe von Technologien an. Beispiele wären Storm, Spark Streaming, Flink und Kafka Streams.

3.2 Apache Storm

Aufgrund der spezifischen Anforderungen haben wir Apache Storm verwendet.

Storm bietet im Gegensatz zu vergleichbaren Streaming-Technologien einen mittelmäßigen Durchsatz, dafür aber eine, für den Anwendungsfall passende, sehr hohe Performance und relative kurze Einarbeitungszeiten für die Entwickler. Dazu erfüllt das Tool im Systemaufbau zwei Aufgaben:

- Die erste Aufgabe umfasst das Befüllen des Data Lakes, also des Archivs aller verfügbaren Daten. Der Data Lake wird hier durch das Hadoop File System (HDFS) realisiert. Das HDFS eignet sich hervorragend zur redundanten Speicherung extrem großer Datenmengen. Als Kehrseite dazu muss das HDFS aber auch als das gesehen werden, was es ist: ein Datenarchiv, zunächst ohne große Indizierungs-möglichkeiten. Daher führen Zugriffe auf Einzeldaten im HDFS zu enormen Ladezeiten (analog zu „Full Table Scans“ aus der Datenbankwelt). In vielen Fällen eignen sich zum Befüllen des Data Lakes auch Technologien wie Apache Flume, die darauf spezialisiert sind, große Datenmengen in ein Hadoop Cluster zu persistieren. In diesem besonderen Anwendungsfall mussten die Rohdaten aber zunächst transformiert werden. Deshalb haben wir hier Apache Storm verwendet.
- Die zweite Aufgabe ist die In-Memory-Durchführung von Analysen des Datenstroms in gewissen (meist eher kurzen) Zeitfenstern. Aus dieser Analyse gewonnene Erkenntnisse, wie eine überdurchschnittliche Anzahl an Verkäufen eines bestimmten Produkts, werden im Serving Layer gespeichert.

Diese Daten werden über Real-Time Dashboards zur Verfügung gestellt, damit der Systemanwender jederzeit den Status aller Automaten einsehen kann. Der Serving Layer selbst ist verantwortlich für die Speicherung der in Speed Layer und Batch Layer berechneten Daten.

3.3 NoSQL-Datenbanken

Typischerweise gibt es beim Zugriff auf den Serving Layer hohe Anforderungen an die Latenzzeiten, sodass sich hier in vielen Fällen NoSQL-Datenbanken anbieten. Bei NoSQL-Datenbanken unterscheidet man grundsätzlich vier Typen:

- Graph-Datenbanken, wie Neo4J, die sich hervorragend zum Speichern von Beziehungsinformationen zwischen Entitäten eignen,
- Dokument-Datenbanken wie MongoDB, die darauf ausgelegt sind, ganze Dokumente in JSON Form zu speichern,
- Key Value Stores, die nur Daten mit sehr geringer Komplexität speichern, diese aber dafür oft In-Memory,

- sowie Wide- bzw. Column Family Stores wie Apache Cassandra oder Apache HBase, die im Gegensatz zu vielen relationaler Datenbanken bei richtiger Anwendung, extrem geringe Zugriffszeiten und horizontale Skalierbarkeit bieten.

Im Gegenzug bringen viele NoSQL-Datenbanken allerdings Einbußen hinsichtlich Konsistenz und Transaktionsverhalten mit sich, und auch Joins von Daten sind häufig sehr ineffizient.

Im beschriebenen Anwendungsfall wurde Apache HBase verwendet. HBase ist prädestiniert für sehr große Datenmengen, viele Abfragen durch viele Clients, häufige Schemaänderungen und wahlfreien Zugriff. HBase selbst ist in der Regel bereits Bestandteil jeder Hadoop Installation.

3.4 Apache Spark

Aber zurück zu unserem Beispiel: Den Batch Layer haben wir hier mit Apache Spark realisiert, dem „State of the Art"-Framework der Batch-Verarbeitung.

Spark hat sich in den letzten Jahren stark verbreitet und ist heute wahrscheinlich die am meiste verbreitete Technologie zur Verarbeitung großer Datenmengen und dazu mit Hadoop integrierbar.

Spark erfordert allerdings etwas Einarbeitungszeit, da für die skalierbare Verwendung funktionaler Programmiercode geschrieben werden muss, und das dürfte für viele Softwareentwickler zunächst gewöhnungsbedürftig sein.

Für strukturierte Daten dagegen hat Spark mittlerweile Abhilfe geschaffen: Mithilfe des Moduls SparkSQL ist es möglich, SQL-Abfragen auf Daten im Hadoop Cluster auszuführen. Der Batch Layer bzw. der Spark Job läuft einmal täglich und führt Berechnungen in Form von Aggregationen und Analysen auf dem gesamten Datenbestand aus. Dazu sagt er z. B. Kundenkonsumverhalten vorher oder errechnet Bestandsprognosen anhand externer Einflussfaktoren wie dem Wetter.

4 Fazit

Big-Data-Architekturen sind komplex.

In unserem Anwendungsbeispiel wurde schnell klar, dass es nicht „das eine" Big Data Tool gibt, das sämtliche Anwendungsfälle abdeckt, auch wenn verschiedene Technologien schon an den Randbereich ihres Kerneinsatzzwecks gehen. So beherrscht Apache Spark seit einiger Zeit mit Spark Streaming auch Stream Processing in Form von Micro Batching. Und Apache Kafka drängt mit Kafka Streams in das Stream-Processing-Umfeld.

Dennoch bleibt der Aufbau einer Big-Data-Architektur stets ein Zusammenspiel verschiedener Frameworks und Tools, die in der Regel für sehr spezielle Einsatzzwecke entwickelt wurden.

Um die vollen Möglichkeiten moderner Big-Data-Technologien so effizient wie möglich auszunutzen, braucht es daher die Kombination einer zahlenmäßig so gering wie möglich zu haltenden Anzahl an Frameworks.

5 Anmerkungen

[1] Der Name der Lambda-Architektur kommt vermutlich aufgrund der Ähnlichkeit zu einem nach links gedrehten „Lambda". Andere vermuten die Herkunft des Namens im Lambda-Kalkül, der Grundlage funktionaler Programmierung.

[2] Marz, Nathan; Warren, James. Big Data: Principles and best practices of scalable realtime data systems. Manning Publications, 2013

