HAW HAMBURG

Bachelorarbeit

Mike Wüstenberg

Konzept einer Lambda-Architektur für verteilte mobile Sensorknoten

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Mike Wüstenberg

Konzept einer Lambda-Architektur für verteilte mobile Sensorknoten

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr. Thomas Lehmann

Eingereicht am: 23. Mai 2022

Mike Wüstenberg

Thema der Arbeit

Konzept einer Lambda-Architektur
für verteilte mobile Sensorknoten

Stichworte

Lambda-Architektur, MapReduce, Apache Spark, Apache Hadoop

Kurzzusammenfassung

Um die immer steigende Anzahl an Daten zu bewältigen, entwickelte Nathan Marz die Lambda-Architektur. Diese berechnet Daten auf zwei unterschiedlichen Pfaden. Dadurch kann die Lambda-Architektur *Consistency* und *Availability* vereinen und bestmögliche Ergebnisse liefern. Um die Machbarkeit einer Lambda-Architektur zu zeigen, wird hier ein Prototyp entwickelt. Für die Umsetzung des Prototyps werden eine Vielzahl an Technologien wie Apache Hadoop, Apache Spark und Apache Druid benutzt. Mit dem erfolgreich umgesetzten Prototyp wird gezeigt, dass eine Lambda-Architektur gute Ergebnisse liefert, aber auch in ihrer Umsetzung komplex ist.

Mike Wüstenberg

Title of Thesis

Concept of a Lambda-Architecture
for distributed mobile sensor nodes

Keywords

lambda-architectur, MapReduce, Apache Spark, Apache Hadoop

Abstract

To handle the ever-increasing amount of data, Nathan Marz developed the lambda-architecture. The architecture calculates data on two different paths. This allows the combination of *consistency* and *availability* to deliver the best possible results. A prototype is developed to show the feasibility of the lambda-architecture. Therefor a variety of technologies, such as Apache Hadoop, Apache Spark and Apache Druid are used to implement the prototype. The successfully implemented prototype shows that a lambda-architecture delivers good results, but that an implementation is complex.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

CRDT conflict-free replicated data types.

DAG Directed Acyclic Graph.

dBm decibel-milliwatts.

HDFS Hadoop Distributed File System.

mDAU monetarisierbaren, täglich aktiven Nutzer.

OLAP online analytical processing.

1 Einleitung

Das Datenaufkommen, welches durch Benutzer und Maschinen erzeugt wird, steigt mit jedem Jahr. So hatte Twitter im Jahr 2020 einen Anstieg der monetarisierbaren, täglich aktiven Nutzer (mDAU) von 27 % [19] im Vergleich zum Vorjahr. Um aus den so entstehenden Daten benutzbare Informationen zu erstellen, wird die Last auf Datenbanken immer höher.

Damit unter diesen Bedingungen weiterhin schnelle und genaue Informationen geliefert werden können, müssen Datenbanken immer stärker skaliert werden. Dies kann bei relationalen Datenbanken schnell zu sehr großem Wartungsaufwand führen. Bei NoSQL Datenbanken wiederum, welche für diese Datenmengen gut geeignet sind, kann das eingeschränkte Datenmodell zu Problemen führen. Diese Probleme zeigen sich auch in der Implementierung von Applikationen. Der Aufbau von *queues, shards, replicas* und anderen Konzepten, welche die Datenbank zum Skalieren benutzt, müssen beachtet werden. Das erhöht den Wartungsaufwand und die Komplexität der Applikation. Mit der steigenden Komplexität des Systems steigt auch die Fehleranfälligkeit. Die Lambda-Architektur will mit ihrem Design diese Probleme umgehen und die Handhabung von großen Datenmengen vereinfachen. Dabei ist sie jedoch nicht frei von Kritik. Besonders ihr komplexer Aufbau wird kritisiert, da hier zwei unterschiedliche Programme die gleichen Ausgangsdaten liefern müssen [12, 8].

Mithilfe eines Prototyps wird in dieser Arbeit die Umsetzung einer Lambda-Architektur gezeigt. Diese Umsetzung wird dazu verwendet, den Streaming-Layer der Architektur auf seine Leistungsfähigkeit zu untersuchen. Damit dieses Teilsystem funktionsfähig ist, müssen gewisse Erwartungswerte erfüllt werden. In den nachfolgenden Kapiteln wird dazu zunächst die Lambda-Architektur beschrieben. Dabei werden die unterschiedlichen Layer erklärt und wie das Problem des *CAP-Theorems* umgangen wird, indem man es auf zwei Wege aufteilt. Danach wird auf Technologien wie *Apache Spark* eingegangen und wie mit ihnen die Lambda-Architektur umgesetzt werden kann.

Bei der Umsetzung werden Sensordaten, die zuvor mithilfe eines *Loomos* gesammelt wurden, in der Lambda-Architektur verarbeitet und die so gewonnenen Informationen auf einem Dashboard visualisiert. Die Berechnungen auf den Daten bestehen aus unterschiedlichen Aggregationen durch *Apache Spark*. Diese können einen starken Anstieg bei der Berechnungszeit verursachen und werden deswegen genauer untersucht. Es soll beobachtet werden, wie sich die Anzahl an Datenbankeinträgen auf die Berechnungszeit auswirkt. Daraus wird abgeleitet, ob die Lambda-Architektur auch bei größeren Datenmengen perfomant ist.

1.1 Aufbau der Arbeit

Diese Arbeit ist in sechs Kapitel unterteilt. Während sich die erste Hälfte mit dem Aufbau der Lambda-Architektur beschäftigt, wird im zweiten Teil auf die Umsetzung eingegangen.

Das erste Kapitel ist eine Einleitung in das Thema und beschreibt die Ausgangssituation sowie Hintergründe zur Lambda-Architektur.

Im zweiten Kapitel werden die Ziele der Lambda-Architektur beschrieben und welche Probleme so umgangen werden. Nachdem die Ziele dargelegt wurden, wird der Aufbau der Lambda-Architektur in den drei Unterabschnitten Batch-, Streaming- und Serving-Layer erklärt. Das dritte Kapitel beschäftigt sich mit der Umsetzung. Hierbei liegt der Fokus auf den unterschiedlichen Technologien wie Apache Kafka, Apache Hadoop und Apache Spark, mit denen die Lambda-Architektur realisiert wird.

Kapitel vier beschreibt den Aufbau der Umsetzung des Prototyps sowie die durchgeführten Versuche und welche Ergebnisse erwartet werden. Die Beschreibung der Versuchsergebnisse und deren Auswertung findet im fünften Kapitel statt. Im sechsten und letzten Kapitel wird ein Fazit aus den Forschungsergebnissen gezogen.

1.2 Verwandte Arbeiten

Das Prinzip der Lambda-Architektur wurde erstmals von Nathan Marz auf seinem *Blog* in dem Artikel *How to beat the CAP theorem* [14] im Oktober 2011 vorgestellt. Der Name Lambda-Architektur ist in diesem Artikel zwar noch nicht zu finden, aber alle

Kerneigenschaften sind bereits vorhanden. Im April 2015 veröffentlichte Nathan Marz sein Buch *Big Data*, welches die Architektur im Detail erklärt und hier auch den Namen Lambda-Architektur verwendet.

Diese Arbeit wird sich dicht an die von Nathan Marz beschriebene Architektur halten. Wie Jimmy Lin in dem Artikel *The Lambda and the Kappa* [12, S. 61] jedoch erwähnt, ist das Prinzip der Lambda-Architektur kein neuartiger Gedanke. Als Beispiel nennt er den Artikel von Butler W. Lampson [11] vom Oktober 1983, dessen Ausführungen der Beschreibung der Lambda-Architektur sehr ähnlich sind. Ein neueres Beispiel ist die *Suro-Architektur* [6] von Netflix, welche die Daten ähnlich der Lambda-Architektur in Teilprobleme aufteilt, aber kein einheitliches Ergebnis liefert.

1.3 CAP-Theorem: Diskrepanz zwischen *Consistency*, *Availability* und *Partition Tolerance*

Das Ziel der Lambda-Architektur ist es einen hohen Grad an *Consistency* und *Availability* für Datenzugriffe zu gewährleisten. Dies klingt zunächst nach einer einfachen Anforderung, ist jedoch eine große Herausforderung. Aber warum ist es eine so große Herausforderung? Den Grundgedanken zur Beantwortung dieser Frage legte Eric A. Brewer im Juli 2000 mit der Vorstellung des CAP-Theorems. Das CAP-Theorem definiert die drei Anforderungen *Consistency*, *Availability* sowie *Partition Tolerance* und dass nur zwei von ihnen zur selben Zeit erfüllt werden können [2].

- **Consistency:** Nach einem erfolgreichen Schreibvorgang müssen alle zukünftigen Lesezugriffe diesen beachten.
- **Availability:** Es ist immer möglich Lese- oder Schreibvorgänge durchzuführen.
- **Partition Tolerance:** Das System funktioniert auch bei zufällig fehlenden Nachrichten und Netzwerkunterbrechungen.

Im Juni 2002 wurde das CAP-Theorem von Seth Gilbert und Nancy Lynch formal formuliert und bestätigt [16].

Zwar sind alle Anforderungen wichtig, aber es können nicht alle drei gleichzeitig erfüllt sein. Der Grund hierfür ist, dass in jedem Netzwerk Partitionen auftreten können. Diese werden durch Timeouts oder Hardwareausfall ausgelöst. Eine solche Netzwerk-Partition

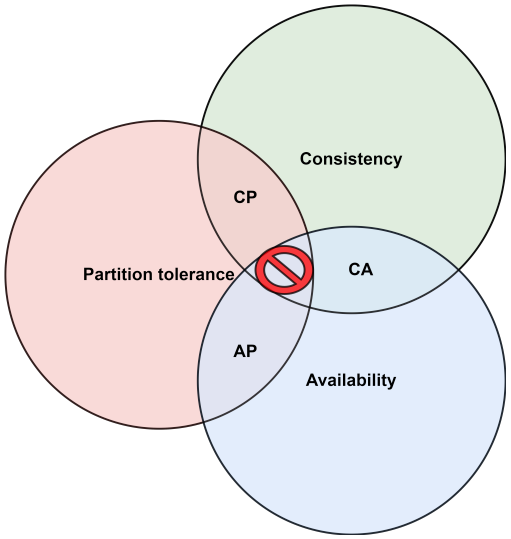

Abbildung 1.1: CAP-Theorem

führt dazu, dass zwei oder mehr Geräte nicht mehr miteinander kommunizieren können. Wird in diesem Zustand auf das System zugegriffen, muss entschieden werden, wie damit umgegangen werden soll. Entweder wird die Nachricht verworfen, da nicht alle Geräte von dem neuen Zustand erfahren haben (*Consistency*), oder sie wird auf die Gefahr hin übernommen, sodass veraltete Lesezugriffe zu Schreibkonflikten führen können (*Availability*).

Dies führt dazu, dass es drei mögliche Kombinationen von Anforderungen gibt, welche in Abbildung 1.1 veranschaulicht sind.

- **Consistency-Availability(CA):** Ist nur auf einem einzelnen Knoten möglich, auf dem es keine Netzwerk-Partitionen gibt. Damit ist es aber auch kein verteiltes System.
- **Consistency-Partition Tolerance(CP):** *Consistency* ist wichtiger als *Availability* und das System tut alles, um Schreib- und Lesezugriffe atomar zu halten.
- **Availability-Partition Tolerance(AP):** Alle Anfragen werden beantwortet, auch wenn dies zu Konflikten führen kann.

Sowohl die CP- als auch die AP-Variante haben Vor- und Nachteile. Systeme, die auf *Consistency* als oberste Priorität setzten, verlieren Anfragen, falls nicht alle Systemteile durch Störungen wie Netzwerkpartitionen erreichbar sind. Anwendungen, welche auf so

ein System zugreifen, funktionieren dann nicht mehr. Allerdings ist gewährleistet, dass immer korrekte Daten geliefert werden. Wird *Availability* bevorzugt, wie es bei vielen modernen Real-Time Anwendungen der Fall ist, werden alle Anfragen beantwortet. Anwendungen, die davon abhängig sind, müssen damit umgehen können, dass die Daten nur *eventual consistency* sind. Dies führt zu einer erhöhten Komplexität in den Anwendungen. Auch die Datenbank wird komplexer, da sie die *replica* reparieren können muss, beispielsweise mithilfe von *conflict-free replicated data types (CRDT)*.

Beide Varianten haben ihre Anwendungsfälle und sind mögliche Optionen. Auch ist es in vielen Fällen keine binäre Entscheidung, sondern eine fließende. So kann man zum Beispiel eine Datenbank mit einem Buffer ausrüsten. Dies schwächt zwar die *Consistency*, weil Schreibvorgänge bereits als erfolgreich gewertet werden, obwohl sie noch nicht in die Datenbank geschrieben wurden, aber es erhöht die *Availability*. Gemeinsam mit den anderen Anforderungen führt dies zu der hohen Komplexität und den vielen zu beachtenden Implementierungsdetails in verteilten Systemen.

Das Ziel der Lambda-Architekturist es sowohl *Consistency*, als auch *Availability* bereitzustellen. Da dies innerhalb einer Umsetzung aber nicht möglich ist, enthält die Architektur zwei Implementierungen. Eine konzentriert sich vollständig auf *Consistency*, während die andere *Availability* gewährleistet. Dies erzeugt zwei unabhängig voneinander verlaufende Pfade im System, welche am Ende aber ein (gemeinsames) Ergebnis liefern.

2 Einführung in die Lambda-Architektur

In den nachfolgenden Kapiteln wird die Theorie der Lambda-Architektur beschrieben. Der Großteil der Informationen stammt hierfür aus dem Buch *Big Data Principles and Best Practices of Scalable Realtime Data Systems* [15], welches von dem Entwickler der Lambda-Architektur Nathan Marz geschrieben wurde.

2.1 Problemstellung

Die Menge an Daten, die täglich generiert wird, überschreitet schon längst das, was traditionelle, relationale Datenbanken bewältigen können. Horizontale Skalierung wird zu einer Voraussetzung, um die vielen Schreib- und Lesezugriffe abzuarbeiten [15, S. 3-5]. Für so eine Skalierung wird häufig *sharding* eingesetzt. Dabei wird ein Teil der Daten über mehrere Datenbankserver verteilt. Alte Daten müssen hierfür gleichmäßig auf alle *shards* verteilt werden.

Während die Daten auf die unterschiedlichen Server verteilt werden, müssen diese offline genommen werden, da sonst neue Daten verloren gehen. Dieser Vorgang muss wiederholt werden, wenn neue *shards* hinzugefügt werden. All das sorgt für einen komplizierten Aufbau der Datenbank und wird dadurch verursacht, dass die Datenbank sich ihrer verteilten Natur nicht bewusst ist. Die Datenbankverwaltung ist aber nicht das einzige, was komplizierter wird. Programme, die auf die Datenbank zugreifen, müssen wissen wie, sie auf die unterschiedlichen *shards* zugreifen und wo sie sich befinden. Das erhöht die Programmkomplexität.

Mit zunehmender Größe werden Ausfälle immer häufiger und Backups für Server und Daten kommen zu dem ohnehin schon komplizierten Aufbau hinzu. Diese Komplexität führt dazu, dass immer mehr beachtet werden muss, wenn etwas verändert oder hinzugefügt wird. Das erhöht auch das Risiko von menschengemachten Fehlern. Durch solche Fehler werden korrupte Daten in die Datenbank geschrieben und ohne zu wissen,

welche Daten korrupt sind, gibt es keine Möglichkeit diese Fehler zu beheben. Mit der Lambda-Architektur sollen diese Probleme vermieden und die Skalierbarkeit sowie die Fehlertoleranz verbessert werden.

2.2 Anforderungen an die Lambda-Architektur

Die Lambda-Architektur ist eine Datenverarbeitungs-Architektur. Neben dem Verarbeiten von Daten, besteht ihre Aufgabe auch darin Daten zu speichern. Diese können im verarbeiteten oder unverarbeiteten Zustand vorliegen. Dabei setzt sie hohe Priorität auf eine möglichst geringe Implementierungskomplexität und hohe Fehlertoleranz. Die folgenden acht Anforderungen sollen hierbei erfüllt werden [15, S. 7-9].

Verallgemeinerung Es muss möglich sein jede Art von Programm zu unterstützen. Die Funktionalität der Lambda-Architektur basiert darauf, dass man eine Gleichung (*query = function(all data)*) [15, S. 13] über alle Daten ausführen kann.

Ad hoc Queries Um die bestmöglichen Informationen aus Daten herauszuholen, muss es möglich sein willkürliche *queries* zu erstellen.

Lese- und Aktualisierungsvorgänge mit geringer Latenz Lesevorgänge müssen mit geringer Latenz durchgeführt werden. Schreibzugriffe variieren oft in ihrer Geschwindigkeit, dürfen aber trotz allem die Robustheit nicht gefährden.

Robustheit und Fehlertoleranz In Systemen mit vielen Komponenten ist es nicht selten, dass Fehler auftreten. Das System muss sich sowohl vor zufällig abstürzenden Komponenten, als auch korrupten Daten schützen und wiederherstellen können.

Skalierbarkeit Die Lambda-Architektur ist durch das Hinzufügen neuer Server horizontal skalierbar. So kommt sie auch mit einer steigenden Anzahl an Daten zurecht.

Erweiterbarkeit Erweiterbarkeit ist die Möglichkeit bestehende Programme zu erweitern, ohne dabei alles neu erfinden zu müssen. Es ist einfach neue Funktionen hinzuzufügen oder eine große Menge an Daten zu migrieren.

Minimale Wartung Einfachere Algorithmen reduzieren die nötigen Wartungen, um das System am Laufen zu halten. Komplexität, die nicht vermieden werden kann, befindet sich nur in nicht kritischen Teilen des Systems.

Debuggability Es müssen Informationen vorhanden sein, um auftretende Fehler zu finden und zu korrigieren.

2.3 Übersicht der Lambda-Architektur

Die Lambda-Architektur ist in drei Schichten aufgeteilt. Dies erlaubt es jeder Schicht sich auf ihren Anwendungsbereich zu konzentrieren. Die Schichten bestehen aus Batch-Layer, Serving-Layer und Streaming-Layer.

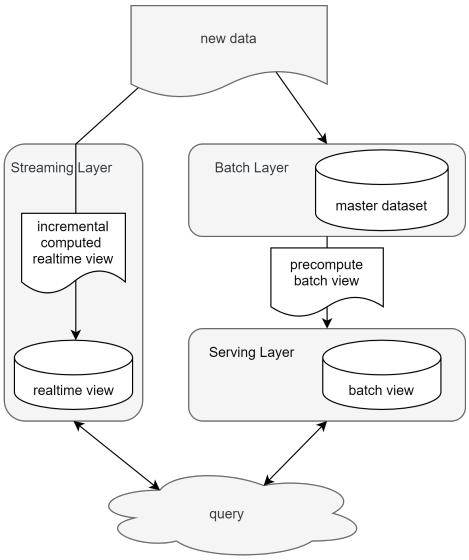

Abbildung 2.1: Lambda-Architektur Übersicht

Der Batch-Layer hat zwei Aufgaben. Die erste dieser Aufgaben ist die Aufnahme neuer Daten und das Speichern dieser in einem unveränderten Zustand. Die zweite Aufgabe ist es aus diesen gespeicherten Daten in Abständen Batch-Views zu berechnen. Diese Batch-Views werden in jedem Durchlauf komplett neu berechnet und anschließend an den Serving-Layer gereicht.

Der Serving-Layer nimmt die vorberechneten Batch-Views und speichert diese. Alte Batch-Views werden gelöscht und von den neuen ersetzt. Auf die gespeicherten Batch-Views kann von außerhalb der Lambda-Architektur mit Leseoperationen zugegriffen werden. Die Lesezugriffe können mit sehr geringer Latenz beantwortet werden, da der Großteil an Berechnungen bereits im Batch-Layer durchgeführt wurde.

Parallel zu Batch- und Serving-Layer (sieh Abbildung 2.1) verläuft der Streaming-Layer. Der Streaming-Layer, häufig auch Speed-Layer genannt, enthält Daten, die im Batch-Layer noch nicht berechnet worden sind und liefert aktuelle Daten. Hierbei liegt der Fokus auf Geschwindigkeit, auch auf Kosten der Datengenauigkeit. Wird ein neuer Batch-View berechnet, dann werden alle bis dahin im Streaming-Layer berechneten Daten verworfen, da diese in dem neuen Batch-View mit inbegriffen sind. Das Neuberechnen des Batch-Layers ist aufgrund der Datenmenge sehr rechenintensiv und wird nur alle paar Tage oder sogar Wochen durchgeführt. Entscheidend für den Abstand der Neuberechnungen ist die Berechnungsdauer und wie wichtig aktuelle Werte sind. Das Starten der Neuberechnung sowie welche Daten im Streaming-Layer verworfen werden, müssen wird von außerhalb der Lambda-Architektur gesteuert. Einzeln bietet weder der Serving-Layer noch der Streaming-Layer eine komplette Sicht auf alle Daten. Für eine vollständige Sicht auf alle Daten müssen diese aus beiden Layern kombiniert werden. Dies geschieht entweder in der Anwendung, welche diese Daten abfragt, oder in einem extra Query-Layer, welcher beide Daten enthält.

2.4 Batch-Layer

Der Fokus des Batch-Layers liegt auf der *Consistency* der Daten und den Berechnungen, welche auf diesen durchgeführt werden. Er soll erreichen, dass Daten und Berechnungen vollständig, korrekt und gesichert sind. *Availability* spielt keine wichtige Rolle im Batch-Layer und wird häufig zurückgestellt, um eine bessere *Consistency* zu erreichen.

Wie in der Übersicht der Lambda-Architektur bereits beschrieben, hat der Batch-Layer die Aufgabe Daten in ihrem unveränderten Zustand zu speichern. Dieser *Master-Dataset* steht im Kern der Lambda-Architektur. Solange dieser intakt ist, können alle anderen Layer wiederhergestellt werden. Das bedeutet, dass dieser besonders vor Hardwareausfällen, Datenkorruption und menschlichen Fehlern geschützt werden muss. Aber auch, dass der Serving- und Streaming-Layer keine besonderen Sicherheitsvorkehrungen benötigen. Der Serving-Layer kann mit dem *Master-Dataset* Daten neu berechnen und wiederherstellen. Der Streaming-Layer enthält nur Daten, die vom Batch-Layer ersetzt werden und baut sich innerhalb weniger Stunden wieder auf.

2.4.1 Eigenschaften des Datenmodells

Die Sicherheit der Daten ist besonders im Batch-Layer wichtig. Damit diese gewährleistet werden kann, ist nicht nur die physikalische Datenspeicherung wichtig, sondern auch welche Form das Datenmodell hat. Es gibt drei Eigenschaften, die den Nutzen und die Sicherheit der Daten erhöhen [15, S. 28-36]:

raw Um möglichst viel aus den gesammelten Daten zu gewinnen, werden diese im unveränderten (raw) Zustand gespeichert. Das bedeutet, dass keine der gesammelten Daten zusammengefasst oder gelöscht werden. Dadurch erhöht sich zwar die verwaltete Datenmenge, aber darauf sind die eingesetzten Technologien ausgelegt. Außerdem erlaubt dies auf unerwartete Anfragen zu antworten. In System wo die Daten überschreiben werden verliert man diese antworten, was den Informationsgehalt der Daten veringert.

So erfasste Daten werden in einer unstrukturierten Form gespeichert und keiner semantischen Normalisierung unterzogen. Dadurch können später verbesserte Normalisierungsalgorithmen auf die alten Daten angewandt werden, um bessere Ergebnisse zu erzielen.

immutable Daten, die im System gespeichert werden, sind unveränderlich und werden nur in Ausnahmefällen gelöscht. Neu hinzugefügte Daten werden an bereits bestehende Datensätze angehängt und überschreiben keine existierenden Daten. Um Dateneinträge voneinander unterscheiden zu können, werden diese mit einem Timestamp versehen (siehe Tabelle: 2.1). Der aktuelle Wert wird durch das Auslesen des neuesten Timestamps abgefragt.

Driving distances		
ID	Distance	Timestamp
Loomo-01	10	2021-09-23T21:00:00Z
Loomo-01	15	2021-09-24T21:00:00Z
Loomo-01	20	2021-09-25T21:00:00Z

Tabelle 2.1: Loomo Fahrdistanz unveränderlich gemacht mithilfe eines Timestamps

Die Unveränderlichkeit der Daten hat zwei Vorteile. Zum einen wird kein Index für das Updaten der Daten benötigt, da diese nur angehängt werden. Das erleichtert wie die Daten gespeichert werden. Zum anderen dient die Unveränderlichkeit der Daten der Absicherung gegenüber menschlichen Fehlern. Bei einem System

mit mutable Daten gehen bei einem Fehler die *guten* Daten verloren und werden überschrieben. Sind die Daten jedoch immutable, werden die schlechten Daten nur hinten angehängt, während die *guten* Daten unverändert bleiben. Gelangen korrupte Daten in die Datenbank, können weiterführende Algorithmen diese herausfiltern oder in Ausnahmefällen sogar komplett aus der Datenbank entfernen. Auch werden keine komplexen Algorithmen vor dem Speichern der Daten ausgeführt, da diese zunächst in einem unveränderten (raw) Zustand gespeichert werden. Dadurch wird Datenkorruption durch fehlerhafte Algorithmen vermieden und an eine Stelle verschoben, wo diese hinterher leichter korrigiert werden kann.

Eternally true Aus der Unveränderlichkeit der Daten folgt, dass diese für immer wahr sind. Eine Aussage zu einem bestimmten Timestamp ist für immer wahr, auch wenn neue Einträge hinzukommen.

Hinzu kommt, dass der Datensatz mit der Zeit immer größer wird. Nur in Ausnahmefällen, beispielsweise aus Performancegründen, werden Daten entfernt.

Diese drei Eigenschaften sind von großem Nutzen und helfen dabei die Anforderungen der Lambda-Architektur (siehe Kapitel 2.2) zu erfüllen. Dass die Daten immutable sind, sorgt für mehr **Robustheit und Fehlertoleranz** des Systems. Die **Erweiterbarkeit** wird dadurch unterstützt, dass die Daten in einem unveränderten Zustand gespeichert sind. Außerdem werden neue Daten nur angehängt und beeinflussen deshalb keine alten Daten.

2.4.2 Datenspeicher

Durch den Aufbau der Lambda-Architektur gibt es nur wenige Anforderungen, die der Datenspeicher erfüllen muss. Aber es wird mit einer sehr großen (Terabyte, Petabyte) Menge an Daten gerechnet. Das bedeutet, dass es unrealistisch ist die Daten auf nur einer physikalischen Maschine zu speichern. Es wird also zur Voraussetzung, dass die Daten verteilt gespeichert werden. [15, S. 54-64].

In Tabelle 2.2 werden die Anforderungen an den Datenspeicher beschrieben. Die Abwesenheit von zufälligen Lese- und Schreibzugriffen ist besonders auffällig und erlaubt fast jede mögliche Art Daten zu speichern. So könnten beispielsweise die Daten in einer Key-Value-Datenbank gespeichert werden. Dies führt aber zu zusätzlichem Aufwand, da Schlüssel für die Daten generiert werden müssen. Neben dem zusätzlichen Aufwand liefert

Anforderungen an den Datenspeicher		
Operation	Anforderungen	Beschreibung
Schreiben	Effizientes Anhängen von Daten	Der einzige Schreibzugriff, der unterstützt werden muss, da Daten nur angehängt werden.
	Skalierbar	Muss mit den immer weiter wachsenden Datenmengen skalieren.
Lesen	Parallele Datenverarbeitung	Der Batch-View wird aus dem gesamten Datenset erstellt und muss parallel Daten verarbeiten können, um mit den stetig mehr werdenden Daten zu skalieren.
Beides	Wahl der Datenkompression	Speicherplatz ist teuer und eine Datenkompression kann die Kosten senken. Sie kann aber auch die Datenverarbeitung verlangsamen.
	Immutable Daten erzwingen	Es muss sichergestellt werden, dass die Daten nicht verändert werden können und Sicherheitschecks dafür eingerichtet werden.

Tabelle 2.2: Übersicht der Anforderungen an den Datenspeicher [15, S. 56 Tabelle 4.1]

so eine Datenbank auch viele Funktionen, die nicht benötigt werden und zu unnötiger Komplexität führen.

Ein geeigneteres Speicherverfahren wäre die Speicherung der Daten auf einem einfachen Dateisystem. Ein herkömmliches Dateisystem ist aber nicht verteilt aufgebaut und damit nicht für das Datenaufkommen geeignet. Jedoch gibt es Dateisysteme wie beispielsweise das *Hadoop Distributed File System (HDFS)*, welche genau für diesen Anwendungsfall gedacht sind. Es bietet die Freiheit eines Dateisystems und der Kompression und Formatierung der Daten sind keine Beschränkungen auferlegt. Das HDFS ermöglicht es die Daten verteilt auf mehrere Maschinen zu speichern und erlaubt so eine horizontale Skalierung des Datenspeichers und erhöht die Ausfallsicherheit.

2.4.3 Berechnung der Batch-Views mithilfe von MapReduce

Der Batch-Layer hat neben dem Speichern der Daten auch die Aufgabe auf diesen Berechnungen durchzuführen. Die Ergebnisse aus diesen Berechnungen werden *Batch-Views* genannt und werden aus dem kompletten Datensatz erstellt. Dies führt zu einer hohen Berechnungszeit, welche es unmöglich macht diese nach Bedarf zu berechnen. Aus diesem Grund berechnet der Batch-Layer diese *Batch-Views* einmal vorab und speichert sie im *Serving-Layer* für zukünftige Zugriffe. [15, S. 83-97]

Der Batch-Layer hat keine zeitkritischen Anforderungen und erlaubt eine tiefe und genaue Analyse der Daten. Hierbei ist jedoch zu beachten, dass das Ergebnis erst später gespeichert wird. Aufgrund des hohen Datenvolumens ist es nicht möglich alle erdenklichen Varianten der Daten zu speichern. Eine bessere Vorgehensweise ist es für sich allein stehende Gruppen zu berechnen, welche dann von der Anwendung beliebig kombiniert werden können. So ist es beispielsweise nicht möglich das Ergebnis einer Berechnung für alle möglichen Stundenbereiche eines Jahres zu speichern, da es zu viele Einträge wären. Stattdessen ist es besser das Ergebnis für jede Stunde einzeln zu berechnen, was zu lediglich 8760 (365 Tage * 24 Stunden) Ergebnissen führt. Um das Ergebnis für einen bestimmten Stundenbereich zu erhalten, müssen nur die Ergebnisse im gewünschten Bereich summiert werden. Dies kann in sehr kurzer Zeit berechnet werden und wird vom Serving-Layer unterstützt.

Damit der Batch-Layer diese Berechnungen auf lange Sicht bewältigen kann, muss die Art, wie die Berechnungen durchgeführt werden, skalierbar sein. Die Skalierung wird durch das Hinzufügen von neuen Ressourcen, in Form von mehr Rechenleistung durch neue, physikalische Maschinen, umgesetzt. Damit dies für die Terra- und Petabyte an Daten, die verarbeitet werden sollen funktioniert, muss die Skalierung linear sein. Ist dies nicht der Fall, müssen für die gleiche Menge an neuen Daten mehr Ressourcen hinzugefügt werden, als für die vorangegangenen. Nach kürzester Zeit ist dies jedoch nicht mehr realistisch.

Ein dafür ausgelegtes Programmiermodell ist *MapReduce*. Das *MapReduce* Modell wurde von Google [7] mit dem Ziel entwickelt, die vielen unterschiedlichen und verteilten Anwendungen mit einem allgemeinen Modell zu beschreiben. Es entstand ein Framework, welches die parallele Ausführung erleichtert und eine Automatisierung erlaubt. Dabei ist das Modell allgemein genug, um in fast allen Anwendungsbereichen eingesetzt zu werden.

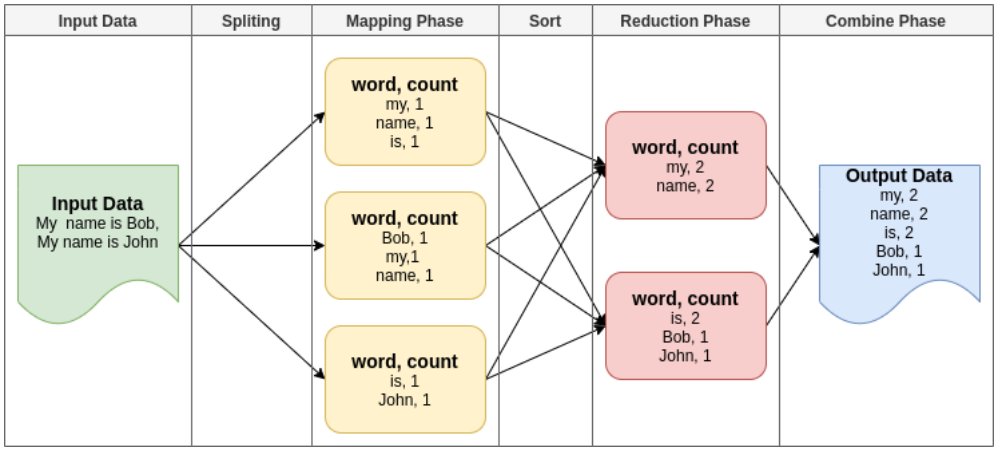

Abbildung 2.2: Berechnung der Wortanzahl mit MapReduce

Beim *MapReduce* Verfahren werden in einem ersten Schritt die Daten auf unterschiedliche *Worker* aufgeteilt. Die darauf folgende *Mapping Phase* wandelt die Daten in *Key/Value* Paare um. In Abbildung 2.2 besteht ein Paar zum Beispiel aus *Name* und *Wortanzahl*. Diese Paare werden nach ihren *Keys* sortiert und in der *Reduction Phase* deren *Values* kombiniert. In diesem Fall wird eine einfache Summe berechnet. Durch die Aufteilung der Daten wird Skalierung möglich gemacht. Bei größeren Datenmengen wird die Anzahl der *Worker* erhöht und die Last verteilt. Alle Details der Parallelisierung werden dabei vom Framework übernommen.

2.4.4 Zusammenfassung

Der Batch-Layer erlaubt es komplexe Analysen auf bestehenden Daten durchzuführen. Die Modelle, die dabei verwendet werden, erlauben die Erweiterung und Skalierung des Systems, sodass auch große Datenmengen kein Problem darstellen. Um vor Fehlern zu schützen, werden Daten im *Master-Dataset* gespeichert, welcher eine Neuberechnung der Batch-Views erlaubt. Somit erfüllt der Batch-Layer fast alle von der Lambda-Architektur gestellten Anforderungen. Einzig die geringe Latenz bei Lese- und Schreibzugriffen erfüllt der Batch-Layer nicht.

2.5 Serving-Layer

Der Serving-Layer liegt in der Lambda-Architektur direkt hinter dem Batch-Layer [15, S. 177-204]. Er hat die Aufgabe Lesezugriffe mit geringer Latenz und hohem Durchsatz auf die *Batch-Views* zu ermöglichen. Dafür werden die vorberechneten *Batch-Views* in einer Datenbank gespeichert. Wichtig bei dieser Datenbank ist, wie bei allen anderen Komponenten in der Lambda-Architektur, dass eine Skalierung über mehrere Server möglich ist.

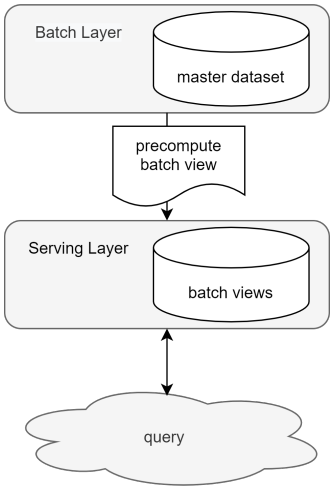

Abbildung 2.3: Übersicht Serving-Layer

Die gespeicherten *Batch-Views* werden bei jeder Neuberechnung vom Batch-Layer vollständig ersetzt. Dafür müssen große Datenmengen in die Datenbank geschrieben und Batch-Schreibzugriffe unterstützt werden. Jedoch bedarf es keiner zufälligen Schreibzugriffe, da der Serving-Layer nur in großen Batch-Operationen aktualisiert wird. Dies bringt viele Vorteile und reduziert die Komplexität des Layers. So muss zum Beispiel keine Verdichtung der Daten durchgeführt werden, um ungenutzten Speicher wiederzugewinnen. Auch müssen Lese- und Schreibzugriffe nicht synchronisiert werden, um das Lesen halb geschriebener Werte zu verhindern.

Durch die Abwesenheit von zufälligen Schreibzugriffen kann der Fokus auf die Optimierung der Lesezugriffe gelegt werden. Für gewöhnlich ist es wichtig eine Normalisierung der Daten durchzuführen, um deren Redundanz zu reduzieren und sie leichter konsistent halten zu können. Dadurch entsteht aber eine Vielzahl von unterschiedlichen Tabellen, bei deren Zugriff die Latenz erhöht wird. Die Konsistenz der Datenhaltung ist jedoch

kein Problem für den Serving-Layer, da dieser keine zufälligen Schreibzugriffe hat und mögliche Konsistenzprobleme vom Batch-Layer überschrieben werden. Dies erlaubt es Daten in einem denormalisierten Zustand zu speichern.

2.6 Streaming-Layer

Mit dem Batch- und Serving-Layer sind fast alle Anforderungen an die Lambda-Architektur erfüllt. Eine Ausnahme bilden Updates mit geringer Latenz [15, S. 209-219]. Hierfür ist der Streaming- oder Speed-Layer verantwortlich und bewältigt dabei zwei Aufgaben. Zum einen das Verarbeiten der Daten und zum anderen deren Speicherung.

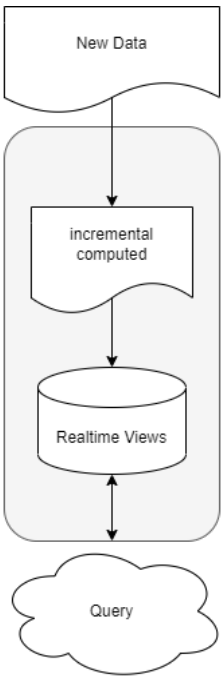

Abbildung 2.4: Übersicht Streaming-Layer

Der Streaming-Layer läuft parallel zum Batch-Layer und bekommt die gleichen Daten. Aber im Vergleich zu diesem werden Daten nicht neu berechnet, sondern inkrementell aktualisiert. Dies ist notwendig, um eine geringe Latenz bei sehr hohen Datenmengen zu erreichen. Inkrementelle Algorithmen sind jedoch komplexer und weniger fehlertolerant, gegenüber korrupten Daten, da alte Werte denn aktuellen beeinflussen aber nicht zu rückverfolgbar sind. Außerdem hat hier das CAP-Theorem einen besonders hohen Einfluss. Da eine geringe Latenz gewollt ist, muss zwischen Erreichbarkeit und Konsistenz

entschieden werden. Die Implementierung von Lösungsstrategien, beispielsweise conflict-free replicated data types, erhöht zusätzlich die Komplexität. Jedoch beschränkt sich dies auf einen sehr kleinen Teil der verwendeten Daten. Sollten dennoch Fehler auftreten, sind diese nur von kurzer Dauer, da alle Daten regelmäßig vom Batch-Layer neu berechnet und ersetzt werden.

Des Weiteren müssen die so berechneten Daten gespeichert werden. Die Datenbank dafür muss fehlertolerant und skalierbar sein sowie zufällige Lese- und Schreibzugriffe unterstützen. Ein typischer Datenbanktyp, der dies bei geringer Latenz unterstützt, sind NoSQL Datenbanken. Darüber hinaus stellt die Art der Daten bei der Lambda-Architektur eine Besonderheit dar. Dadurch, dass die Daten überschrieben werden, ist es nicht unbedingt notwendig, dass die Daten genau sind. So kann der Wert zuerst eine Schätzung sein, um damit eine bessere Latenz zu erreichen und später im Batch-Layer durch *eventual accuracy* präzisiert werden. Zuletzt muss die Datenbank auch das Löschen von Daten unterstützen. Daten, die vom Batch-Layer berechnet wurden und aus dem Serving-Layer abfragbar sind, müssen aus dem Streaming-Layer entfernt werden. Dabei ist es wichtig die restlichen, noch nicht im Serving-Layer vorhandenen Daten, zu behalten.

2.7 Kritik an der Lambda-Architektur

Die Lambda-Architektur hat viel Aufmerksamkeit auf sich gezogen, jedoch nicht nur gute, sondern auch viel Kritik. Dazu gehören zum einen die vielen unterschiedlichen Technologien, Modelle und Frameworks, die notwendig sind, um eine vollständige Lambda-Architektur abzubilden [12]. Das erhöht die Komplexität, welche die Lambda-Architektur gerade versucht zu reduzieren. Hinzu kommt, dass es notwendig ist zwei unterschiedliche Implementierungen für die Datenverarbeitung zu entwickeln. Eine für den Batch- und eine für den Streaming-Layer, welche beide gewartet und synchron gehalten werden müssen. Verarbeitet einer der Layer die Daten nicht korrekt ist das komplette Endergebnis falsch. Zum anderen wird häufig die Menge an benötigten Ressourcen kritisiert [1]. Der Batch- und Streaming-Layer werden schnell zu zwei unterschiedlichen Projekten, welche beide ein eigenes Team und Wissen benötigen. Das führt zu höheren Kosten oder einer längeren Entwicklungszeit. Hinzu kommen die Kosten für Hardware und Strom, die besonders bei großen Projekten zum Tragen kommen, da sowohl der Batch-, als auch der Streaming-Layer eigene Hardware benötigen.

3 Umsetzung der Lambda-Architektur mit Visualisierung

Um die Lambda-Architektur in einem funktionsfähigen Prototyp umzusetzen, sind eine Vielzahl an Technologien notwendig. In diesem Kapitel werden diese beschrieben sowie einige Alternativen aufgezeigt. Durch die Entwicklung eines Prototyps wird die Machbarkeit der Lambda-Architektur gezeigt. Die Aufgabe des Prototyps wird es sein, Daten von einem Loomo zu empfangen, diese aufzuarbeiten und auf einem Dashboard auszugeben. Die benötigten Technologien für die Lambda-Architektur können dabei grob auf drei Einsatzgebiete aufgeteilt werden. Das erste davon ist die Übertragung der Daten zwischen den Komponenten. Dies geschieht sowohl von externen Geräten wie dem Loomo, als auch intern zwischen den Komponenten. Das zweite Einsatzgebiet ist das Speichern der Daten, welches an unterschiedlichen Stellen nötig ist. Die Berechnung von Daten ist das letzte Einsatzgebiet und wird für die Umsetzung des Batch- und Streaming-Layer benötigt.

Ein typisches Szenario würde damit beginnen, dass der Loomo seine Sensordaten, bestehend aus Odometrie und Wi-Fi Signalstärke, an die Lambda-Architektur überträgt. Der Streaming-Layer nimmt diese an und speichert die Daten zunächst im *Master-Dataset*. Von Dort aus können sie später vom Batch-Layer verwendet werden, um den Serving-Layer zu aktualisieren. Danach berechnet der Streaming-Layer die zurückgelegte Distanz des Loomo sowie weitere Werte und speichert diese ab. Im Unterschied zur normalen Lambda-Architektur passiert dies nicht in einer eigenen Datenbank, sondern die Daten werden mit im Serving-Layer gespeichert. Dadurch kann das Dashboard zentral auf alle Daten zugreifen und benötigt keine separaten Zugriffe auf den Serving- und Streaming-Layer. Für die Umsetzung eines solchen Szenarios müssen Datenübertragung, Datenspeicherung und Datenverarbeitung in der Lambda-Architektur möglich sein.

3.1 Datenübertragung

Die Lambda-Architektur ist für sehr große Datenmengen ausgelegt, daher wird mit einer großen Anzahl an zu übertragenden Nachrichten gerechnet. Diese Nachrichten kommen dabei von vielen unterschiedlichen externen Quellen. Um diese zu übertragen, wird ein Messaging-System benötigt, welches diese Menge an Daten bewältigen kann. Hierfür steht eine Vielzahl unterschiedlicher Frameworks zur Verfügung. Unter anderem der Messaging-Broker RabbitMQ [17]. Dieser hebt sich durch eine besonders einfache Bedienung hervor. Allerdings besitzt er keinen permanenten Speicher. Eine weitere, neuartige Alternative stellt *Pulsar* dar, welches *Geo-Replication* und *Tiered Storage* unterstützt. Der inoffizielle Standard der Lambda-Architektur ist jedoch weiterhin *Apache Kafka* [4]. Die Gründe hierfür sind die gute Skalierbarkeit [9] und das große Ökosystem. Die Kafka API unterstützt eine Vielzahl von Programmiersprachen, was es erleichtert unterschiedliche Geräte mit der Lambda-Architektur zu verbinden. Zusätzlich unterstützen viele Frameworks zur Verarbeitung von Streams die Kafka API, wodurch das Versenden von Daten vereinfacht wird.

Apache Kafka wird zum Verbinden der Komponenten eingesetzt (siehe Abbildung 3.1). Dabei handelt es sich zum einen um Datenquellen außerhalb der Lambda-Architektur, die zu verarbeitende Daten generieren. Zum anderen wird Apache Kafka vom Streaming-Layer für das Einlesen von unbearbeiteten Daten und das Übertragen von verarbeiteten Daten an die Datenbank verwendet.

Damit die Übertragung der Daten erfolgreich abgeschlossen werden kann und es im weiteren Verlauf zu keinen Fehlern kommt, müssen die Daten in einem einheitlichen Format gesendet werden. Bei kleinen Projekten ist ein Schema im *JSON-Format* meist ausreichend und kann leicht gepflegt werden. Es hat jedoch den Nachteil, dass es nicht typsicher ist. So kann es schnell passieren, dass unbeabsichtigt ein String-Wert irrtümlich als Integer übertragen wird. Um dies zu vermeiden, empfiehlt es sich ein Serialisierung-Framework wie *Apache Avro* zu benutzen. Diese sind typsicher und lassen eine frühzeitige Erkennung von fehlerhaften Daten zu, damit diese nicht in die Lambda-Architektur gelangen [15](S. 48). Außerdem helfen sie dabei ein einheitliches Schema zu erzwingen und bieten die Option es im späteren Verlauf auch weiterzuentwickeln und zu verändern.

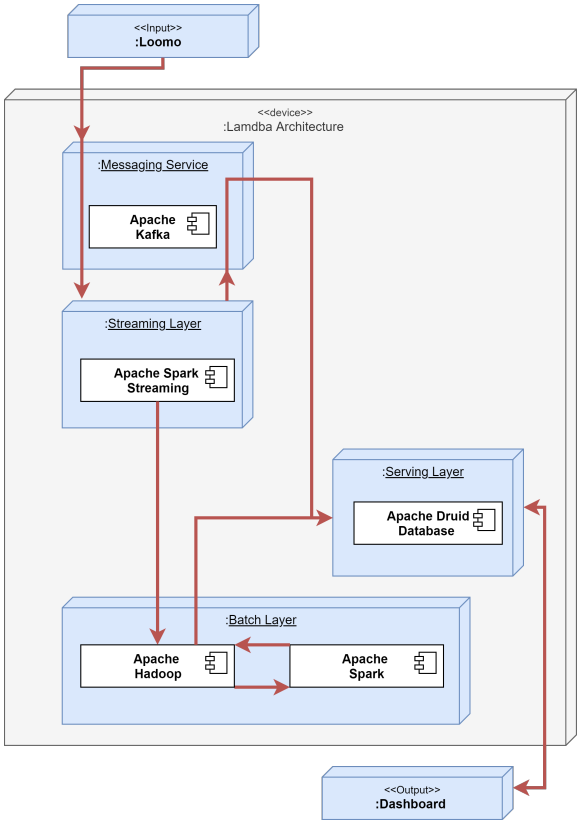

Abbildung 3.1: Technischer Aufbau der Lambda-Architektur

3.2 Datenspeicher

Die Lambda-Architektur hat das Speichern von Daten an mehreren Positionen vorgesehen. Diese sind im Batch-Layer für die Langzeitspeicherung, im Serving-Layer um Anfragen zeitnah zu beantworten und im Streaming-Layer, welcher den Serving-Layer ergänzt und häufig neue Daten erhält. Der Serving-Layer könnte mit einer einfachen Datenbank wie *ElephantDB* [13] umgesetzt werden. Da sich die Anforderungen jedoch mit dem Streaming-Layer überschneiden und die Anzahl der eingesetzten Datenbanken gering gehalten werden soll, wird sowohl der Serving-, als auch der Streaming-Layer mit *Apache Druid* [5] umgesetzt.

Apache Druid ist eine analytische Echtzeit-Datenbank mit Unterstützung für *online analytical processing (OLAP)*. Sie unterstützt sowohl *Echtzeit-Ingestion* von Apache Kafka, als auch *Batch-Ingestion* von HDFS. Das Datenschema ist flach und Event-basiert, wobei jeder Eintrag einen Timestamp besitzt (siehe Tabelle 3.1). Der flache Aufbau der

Einträge vermeidet *joins*, welche die Performance deutlich senken. Einträge werden in Segmenten gespeichert, die alle Einträge aus einem bestimmten Zeitraum enthalten. Das erlaubt es alte Segmente des Streaming-Layers zu löschen und diese durch die Ergebnisse des Batch-Layers zu ersetzen. Dadurch können sowohl die Ergebnisse des Serving-, als auch des Streaming-Layers in einer *Druid Datasource* gespeichert werden.

Odometry Distances		
time	distance	senderName
2021-07-07T09:55:23.285Z	2080.26	Sender01
2021-08-25T12:42:12.102Z	2131.60	Sender01

Tabelle 3.1: Beispiel einer Druid Datasource

Daten aus dem Streaming-Layer werden über Apache Kafka zu Apache Druid übertragen. Daten für den Serving-Layer werden aus dem HDFS direkt geladen. Das HDFS [3] ist der Hauptspeicher des Batch-Layers, eignet sich für große Datenmengen und ist sehr fehlertolerant aufgebaut. Um das teure Bewegen von Daten zu verringern, sollte das HDFS möglichst dicht an der Hardware sitzen, welche die Berechnungen durchführt. Da das HDFS von sich aus keine Möglichkeit besitzt Daten aus Apache Kafka einzulesen, benötigt es ein Verbindungsstück. Dieses Verbindungsstück wird mithilfe des Streaming-Layers umgesetzt. Hierbei werden Daten, die zuvor geladen wurden, zuerst in das Dateisystem geschrieben und erst dann die Berechnungen gestartet (siehe Abbildung 3.1).

3.3 Umsetzung des Batch- und Streaming-Layers

Für die Umsetzung des Batch- und Streaming-Layers hat Nathan Marz Hadoop MapReduce [3] und *Apache Storm* [20] eingesetzt. Hadoop MapReduce ist, wie auch HDFS, ein Teil des Apache Hadoop Projektes und eine Implementierung von MapReduce. Da Hadoop direkt auf den Festplatten arbeitet und dabei kaum Arbeitsspeicher benötigt, ist es kosteneffizient, weil teurer Arbeitsspeicher eingespart werden kann. Außerdem ist es dadurch in der Lage auch Batch-Berechnungen durchzuführen, die ansonsten nicht in den Arbeitsspeicher passen würden. Apache Storm wird von Nathan Marz für die Umsetzung des Streaming-Layers eingesetzt. Es ermöglicht Echtzeitberechnungen und folgt dem *one-at-a-time* Modell. Hierbei werden die Daten einzeln verarbeitet, sobald sie verfügbar sind. Das sorgt für eine geringe Latenz [15](S. 229), da Daten sofort verarbeitet werden. Allerdings schadet es dem Durchsatz, weil dabei alle Daten einzeln geladen und

verarbeitet werden. Außerdem hat man mit Hadoop MapReduce und Apache Storm zwei unterschiedliche Systeme, die verschiedenen Projekten angehören und folglich auch unterschiedlichen Programmcode aufweisen. Um diese Diskrepanz zu vermeiden, wird bei der Umsetzung dieser Arbeit *Apache Spark* [18] eingesetzt.

Der große Vorteil von Apache Spark ist, dass es sowohl für die Umsetzung des Batch-, als auch des Streaming-Layers benutzt werden kann. Für den Streaming-Layer kommt hier das *micro-batched* Modell zum Einsatz. Dabei werden Daten gesammelt, zu einem Batch zusammengefasst und dieser Batch dann in einem Stück berechnet. Diese Art der Berechnung entspricht demselben Schema, das auch beim Batch-Layer verwendet wird. Das vereinfacht die Verständlichkeit und ermöglicht es Codeteile beider Layer untereinander zu verwenden. Damit trotzdem eine hohe Geschwindigkeit erreicht wird, werden die Batchgrößen klein gehalten. Dadurch ist die Berechnungszeit für jeden Batch so kurz, dass dieser wie eine Echtzeitberechnung behandelt werden kann und so das Endergebnis regelmäßig aktualisiert wird. Im Vergleich zum *one-at-a-time* Modell haben die Micro-Batches einen besseren Datendurchsatz, da mehr Daten auf einmal bearbeitet werden, aber eine schlechtere Latenz, weil auf diese Daten gewartet werden muss.

Des Weiteren benutzt Apache Spark für seine Berechnungen auch Arbeitsspeicher, um dort Zwischenergebnisse zu speichern. Dadurch reduziert sich besonders bei Berechnungen mit häufigem Datenzugriff die Berechnungszeit. Um die Zugriffe zu optimieren, nutzt Apache Spark eine verallgemeinerte Variante von MapReduce, die einen *Directed Acyclic Graph (DAG)* einsetzt. Dieser erlaubt eine größere Flexibilität beim Aufbau einzelner MapReduce Schritte. Dabei kann jeder Schritt in einem DAG beliebig viele Aufgaben umfassen, um so Datenverschiebungen zu vermeiden. Auch kann ein DAG mehrere verbundene *MapReduce* Jobs enthalten und so das Zwischenschreiben der Daten auf die Festplatte reduzieren.

Die Implementierung von Apache Spark in der Lambda-Architektur ist in Abbildung 3.1 abgebildet. Für die Umsetzung des Streaming-Layers ist *Spark Streaming* zuständig und befindet sich dafür am Eingang der Lambda-Architektur. Neben der Berechnung von *Realtime-Views* hat Spark Streaming noch die Aufgabe Daten aus Apache Kafka auszulesen und diese unverändert auf dem HDFS zu speichern. Möglichst dicht am HDFS sitzt der mit Apache Spark umgesetzte Batch-Layer und liest dort die Daten für seine Berechnungen aus. Im Anschluss an die Berechnungen werden die Ergebnisse für kurze Zeit zurück auf das HDFS geschrieben, von wo aus sie der Serving-Layer einlesen kann.

3.4 Batch-View Berechnung

Die Komponenten der Lambda-Architektur arbeiten größtenteils unabhängig voneinander und reagieren nur auf neu eingehende Nachrichten. Eine Ausnahme bildet jedoch die Neuberechnung der Batch-Views, da diese nicht durchgehend stattfindet, sondern nur in regelmäßigen Abständen durchgeführt wird. Die Größe des Abstandes kann sehr unterschiedlich sein, abhängig vom Aufwand und wie wichtig die Daten sind. So könnten bei wichtigen Daten die Berechnungen jeden Tag durchgeführt werden, während bei einem sehr großen Datensatz die Berechnung nur einmal im Monat erfolgt.

Unabhängig davon wie häufig die Berechnung durchgeführt wird, muss dies mit dem Streaming- und Serving-Layer abgestimmt werden. Die Lambda-Architektur hat dafür jedoch keine Komponenten vorgesehen und muss daher von extern gesteuert werden.

Zunächst muss festgelegt werden bis zu welchem Timestamp Daten berechnet werden sollen. Da die Apache Druid Datenbank ganze Segmente entfernt und diese nach Stunden aufgeteilt sind, sollen alle Daten bis zur letzten vollen Stunde berechnet werden. Bis zu diesem Timestamp werden die neuen Batch-Views berechnet. Sobald das abgeschlossen ist, werden aus der Apache Druid Datenbank, welche sowohl Batch-, als auch Realtime-Views enthält, alle Segmente bis zum Timestamp entfernt und durch die neuen Batch-Views ersetzt.

3.5 Visualisierung

Für die Umsetzung des Dashboards wird *Grafana* [10] verwendet. Grafana bietet eine Vielzahl an unterschiedlichen Graphen und Plugins, was die Flexibilität erhöht. Eines dieser Plugins ermöglicht den Zugriff auf die Apache Druid Datenbank mit einfachen SQL-Anfragen. Zur Darstellung der vom Loomo erzeugten Daten gehören die zurückgelegte Distanz, die durchschnittliche Wi-Fi Stärke und die gefahrene Strecke. Beim Design des Dashboards wurden einige einfache Regeln beachtet. Dazu gehört, dass das Erscheinungsbild einheitlich wirkt und alle Graphen auf einen Blick sichtbar sind. Außerdem muss sofort erkennbar sein, was der jeweilige Graph darstellt. Zusätzlich werden wichtige Werte durch Farben hervorgehoben.

4 Evaluationsumgebung

In den vorangegangenen Kapiteln wurde das Konzept der Lambda-Architektur erläutert. Die Frameworks Apache Spark, Apache Hadoop, Apache Kafka und Apache Druid erlauben die Umsetzung aller Architektur-Layer. Mithilfe eines Prototyps wird die Leistungsfähigkeit einer solchen Umsetzung gezeigt. Aufgrund des komplexen Aufbaus wird hier nicht die vollständige Architektur beleuchtet. Der Fokus wird auf den Streaming-Layer und Apache Spark gelegt. Apache Spark ist das Herzstück der Umsetzung und verantwortlich für die Datenverarbeitung im Batch- und Streaming-Layer.

Für die Evaluation werden unterschiedliche Arten von Aggregationen verwendet, welche im Streaming-Teil von Apache Spark ausgeführt werden. Dabei wird der Batch-Layer nicht einzeln beleuchtet, da dieser dieselben Berechnungen durchführt und geringere Anforderungen hat. Die Aggregationen stehen deshalb im Mittelpunkt der Aufmerksamkeit, da sie den Zugriff auf alle gespeicherten Daten benötigen. Je nach deren Implementierung kann dies zu einer steigenden Berechnungszeit führen. Andere Operationen, wie das Abflachen der Datenstruktur oder das Ausführen von einfachen Berechnungen, betreffen nur einzelne Einträge und sollten zu keinem Anstieg führen.

Damit die Leistungsfähigkeit der Architektur bei steigendem Datenvolumen gewährleistet ist, darf die Berechnungszeit der Aggregationen nicht zu schnell ansteigen. Das erwünschte Ziel der Versuche ist es höchstens einen linearen Anstieg zu erhalten. Sollte dies nicht der Fall sein und sich ein höherer Anstieg ergeben, so ist die Aggregation nicht für den Streaming-Layer geeignet.

CPU	Intel® Core™ i9-9980XE CPU @ 3.00GHZ
RAM	Corsair VENGEANCE LPX 128 GB (4 × 32 GB) DDR4 2666MHz
HDD	Samsung SSD 970 Evo Plus 2 TB

Tabelle 4.1: Hardwarespezifikationen

Für die Versuche wird die komplette Architektur auf einem einzelnen Server ausgeführt. Das reduziert den Prototyp auf seinen einfachsten Zustand ohne eine Skalierung. Darüber hinaus liegt der Fokus auf dem Streaming-Layer, wodurch Apache Hadoop keinen Einfluss auf die Versuche hat. Die Hardwarespezifikationen des Servers können der Liste 4.1 entnommen werden.

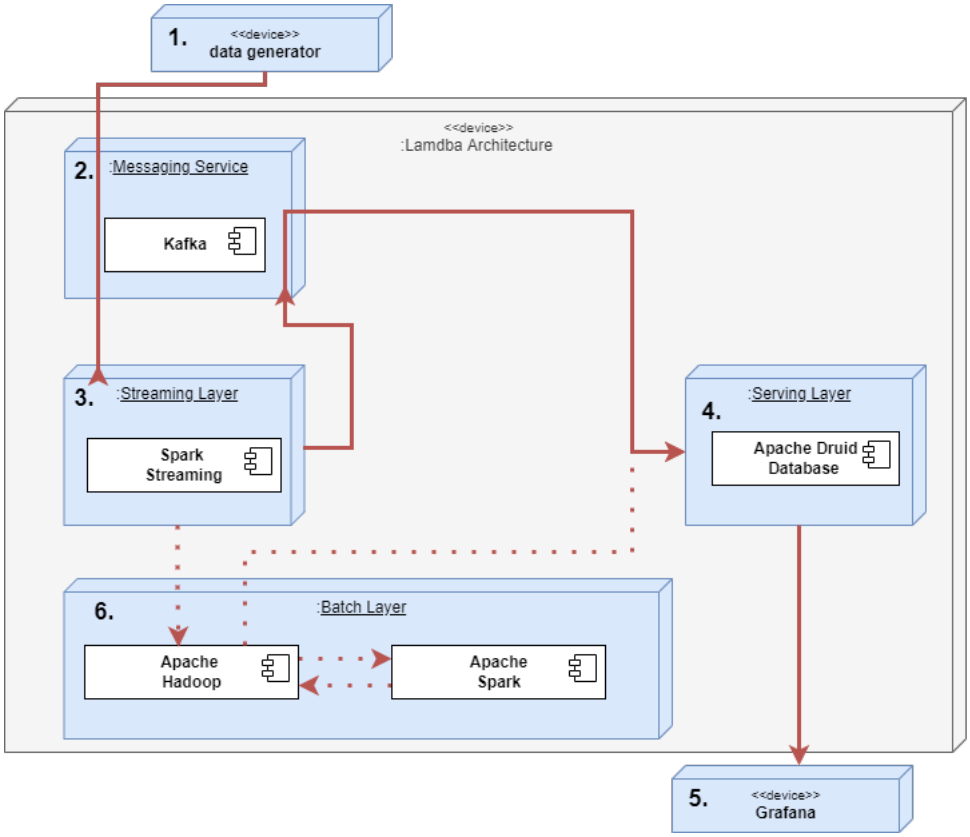

Abbildung 4.1: Versuchsumgebung

4.1 Gesamtaufbau

Die durchgeführten Versuche fokussieren sich zwar auf den Streaming-Layer, jedoch werden die anderen Komponenten weiterhin benötigt. Dieses Kapitel zeigt einen detaillierten Aufbau des Gesamtsystems. Die für die Umsetzung verwendete Software, wie zum Beispiel Apache Kafka, besteht selbst in der kleinesten ausführbaren Version aus mehreren Instanzen. Um das Starten, Verwalten und Konfigurieren dieser zu vereinfachen, wird

jede Komponente in ihrer eigenen Instanz gestartet. Hierfür wird die Container Virtualisierungssoftware *Docker* verwendet. So besteht fast jede Komponente der technischen Umsetzung, welche in Kapitel 3 beschrieben und in Abbildung 4.1 zu sehen ist, aus mehreren Containern.

Nachfolgend ist eine Beschreibung der einzelnen Komponenten, welche in der Abbildung 4.1 zu sehen sind. Jeder Komponente ist hierfür einer Zahl zugeordnet.

1. Der Datengenerator sitzt vor der Lambda-Architektur. Er ist dafür zuständig die Daten, welche später in der Architektur verarbeitet werden, zu erzeugen. Um die Daten zu versenden, wird Apache Kafka verwendet. Zur Datenübertragung werden die Nachrichten zunächst in einen Byte-Stream umgewandelt. Dies geschieht mit dem Serialisierungs-Framework *Apache Avro*. Damit wird die Größe der Nachrichten verringert und das Framework erlaubt die Prüfung auf korrekte Datentypen. Dadurch kann möglichst früh festgestellt werden, ob der Nachrichtenaufbau wie erwartet ist.

2. Apache Kafka ist eine Event-Streaming-Plattform. Es wird für die Nachrichtenübertragung im Streaming-Layer verwendet. Dazu gehört das Einlesen der Nachrichten vom Datengenerator sowie die Verbindung zwischen Apache Spark und der Datenbank im Serving-Layer.

   Diese Komponente besteht aus insgesamt drei Containern. Der Hauptinstanz von Apache Kafka, welche die Datenübertragung regelt. Einer Instanz von Apache *ZooKeeper*, die Apache Kafka Instanzen koordiniert, falls mehrere vorhanden sein sollten. Zuletzt noch eine Instanz von Kafdrop, welches die Entwicklung und das Debuggen von Apache Kafka Programmen unterstützt. Kafdrop besitzt eine Weboberfläche, die das Einsehen von Apache Kafka Nachrichten vereinfacht.

3. Apache Spark ist der Kern des Versuchsaufbaus und kann in zwei Teile aufgeteilt werden. Der erste Teil von Apache Spark ist für die im Batch-Layer durchgeführten Berechnungen zuständig. *Spark Streaming*, der zweite Teil von Apache Spark, ist für die Berechnungen im Streaming-Layer zuständig und der Fokus der Versuche.

   Dabei führt Spark Streaming die Berechnungen nicht für jede Nachricht einzeln aus, so wie es bei einem Stream üblich wäre, sondern verarbeitet mehrere Nachrichten auf einmal in einem Batch. Die Größe eines Batches wird hierbei so klein gehalten, dass die Verarbeitung wie ein Stream wirkt. Dadurch wird zum einen der Durchsatz

erhöht und zum anderen Berechnungen aus dem Batch-Layer im Streaming-Layer ermöglicht.

Die Daten für die Nachrichten werden aus Apache Kafka ausgelesen und der Byte-Stream in ein für Apache Spark lesbares Format umgewandelt. Nach der Verarbeitung werden die Daten wieder zurück zu Apache Kafka gesendet, wo sie vom Serving-Layer ausgelesen werden können.

Apache Spark besteht in der kleinsten Form aus zwei Instanzen. Einem *Master*, welcher die Arbeit verteilt und einem *Worker*, welcher die Berechnungen durchführt. Die eigentlichen Berechnungen werden von einem *Scala* Programm durchgeführt, welches beim Start Instanzen zur Verfügung gestellt bekommt.

4. Apache Druid ist eine spaltenorientierte Datenbank, welche für die Umsetzung des Serving-Layers und das Speichern der Daten aus dem Streaming-Layer eingesetzt wird. Dabei werden die Daten sowohl aus Apache Kafka, als auch aus Apache Hadoop, dem Speicherort des Batch-Layers, eingelesen und in derselben Tabelle abgespeichert. Die Speicherung erfolgt in Zeitsegmenten, die entfernt werden können, um sie durch neu berechnete Daten aus dem Batch-Layer zu ersetzen.

5. Am Ende der Lambda-Architektur steht ein Dashboard, welches mit *Grafana* umgesetzt ist und der Visualisierung der Daten dient. Mithilfe eines Plugins ist es direkt mit Apache Druid verbunden und hat so Zugriff auf die dort gespeicherten Daten. Diese werden in Form von unterschiedlichen Diagrammen auf einer Weboberfläche dargestellt.

6. Der Batch-Layer bleibt von den Versuchen unberührt, existiert unter normalen Umständen aber parallel zum Streaming-Layer. Er besteht aus Apache Hadoop, welches als *Master-Dataset* dient und aus Apache Spark. Neue Daten erhält der Batch-Layer vom Streaming-Layer und speichert diese zunächst separat von den bereits in Apache Hadoop archivierten Daten ab. Startet eine Neuberechnung des Batch-Layers, so verschiebt Apache Spark diese Daten ebenfalls zu den archivierten Daten. Auf diesem kompletten Datensatz wird dann die Berechnung durchgeführt und das Endergebnis in Apache Hadoop gespeichert. Von hier kann der Serving-Layer die Daten auslesen.

4.2 Versuchsaufbau

Die Evaluation besteht aus vier unterschiedlichen Versuchen. Für deren Vergleich werden gleichbleibende Daten benötigt. Um einen identischen Datensatz mehrmals zu erzeugen, wird ein Java Programm eingesetzt. Dieses generiert Nachrichten (siehe Tabelle **??**), die an Apache Spark gesendet werden, welches dann die Berechnungen durchführt. Die Nachrichten folgen dem Vorbild der vom Loomo gesendeten Nachrichten und haben den gleichen Aufbau. Von besonderer Bedeutung für die Aggregationen sind die *wifiData* und *odomData* Felder. Das *wifiData* Feld enthält die Wi-Fi-Signalstärke in decibel-milliwatts (dBm) und wird im zweiten Versuch benutzt. Es enthält achtzehn Integerwerte, welche einen zufälligen Wert zwischen -30 und -90 annehmen können. Damit die Werte konstant zufällig sind, wird der Zufallsgenerator beim Start mit einem festen *Seed* initialisiert. Das *odomData* Feld wird im dritten und vierten Versuch benutzt, um die zurückgelegte Distanz zu berechnen. Die Felder *senderName* und *location* sind für die Versuche nicht von Bedeutung, da nur ein Gerät zum Versenden von Nachrichten verwendet wird. Anhand des *findTimestamp* Feldes wird im vierten Versuch die Reihenfolge der Nachrichten überprüft.

senderName	Ein String, welcher zur Identifikation dient.
location	Ein String, welcher die geschätzte Raumposition angibt.
findTimestamp	Absendezeitpunkt der Nachricht, generiert beim Absender.
odomData	Ein Array aus X, Y, Z Positionskoordinaten.
wifiData	Eine Map aus Integerzahlen, welche die Verbindung zu achtzehn Wi-Fi-Hotspots beschreibt.

Während eines Versuchsdurchlaufs werden Nachrichten mit einem Abstand von acht Sekunden an die Lambda-Architektur gesendet. Der Versuch wird beendet, wenn 40000 Jobs von Apache Spark brechnet wurden. Die Gesamtlaufzeit beträgt vier Tage. Insgesamt wird dies für die folgenden vier Versuche wiederholt. Bei den Ergebnissen der Versuche wird grundsätzlich eine Streuung der Berechnungszeit erwartet. Diese entsteht durch das Betriebssystem und andere Programme, welche CPU-Zeit benötigen sowie Netzwerkverzögerungen. Für die Auswertung wird daher die Trendlinie des Graphen verwendet, sofern die Tendenz im Graph nicht eindeutig zu erkennen ist.

4.2.1 Versuch 1: Einlesen von Daten und Normalisierung

Im ersten Versuch geht es darum sicherzustellen, dass das Einlesen der Daten keine unerwartete Last erzeugt. Dies ist von besonderer Bedeutung, da dieser Schritt in den Aggregationen der Folgeversuche ein Bestandteil der Berechnung ist. Sollte der Versuch lange Berechnungszeiten hervorbringen, würden diese auch die Ergebnisse der anderen Versuche beeinflussen. Es werden die drei Arbeitsschritte aus der Liste **??** ausgeführt.

- Einlesen der Daten aus Apache Kafka.
- Vereinfachen der Datenstruktur.
- Durchschnitt aus den achtzehn Einträgen aus dem *wifiData* Feld erstellen.

Jede dieser Aufgaben ist auf genau eine Nachricht beschränkt und keine der vorab gesendeten Nachrichten beeinflusst die aktuelle Nachricht. Da die Anzahl der Nachrichten die Berechnungszeit nicht beeinflusst, sind die Berechnungen der entscheidende Faktor. Das Abflachen der Datenstruktur ist eine einfache Zuordnung der Variablen und hat eine Komplexität von $\mathcal{O}(1)$. Für die Durchschnittsberechnung muss eine Schleife durchlaufen werden, was eine Komplexität von $\mathcal{O}(n)$ hat. Aufgrund der konstanten Größe des Feldes ist diese aber für alle Jobs konstant. Das Einlesen von einem Event aus Apache Kafka hat eine theoretische Komplexität von $\mathcal{O}(1)$, aber aufgrund des Netzwerkes und dem Auslesen der Daten wird hier mit Variationen gerechnet. Erwartungsgemäß sollte die Berechnungszeit über alle 40000 Jobs trotzdem konstant ausfallen. Dieser Versuch gilt als erfolgreich, wenn das Ergebnis sowohl eine konstante Berechnungszeit über alle Jobs aufweist, als auch deutlich kürzer ausfällt, als im zweiten und dritten Versuch.

4.2.2 Versuch 2: Spark Aggregation

Der zweite Versuch dreht sich um die Spark eigenen Aggregationen. Dabei handelt es sich um Operationen, die im Apache Spark Framework implementiert sind. Diese decken bereits eine Vielzahl an einfachen Problemen, wie zum Beispiel den größten Wert finden, ab. Für eine Umsetzung der Lambda-Architektur ist es wichtig, dass diese effizient berechnet werden können, da eine ansteigende Berechnungszeit einen Großteil der Berechnungen betreffen würde und die benötigten Ressourcen erhöht.

Wie bereits erwähnt, werden die Arbeitsschritte aus dem ersten Versuch vor der eigentlichen Aggregation durchgeführt. Die Aggregationen des zweiten Versuches sind in der Liste ?? zu sehen.

- Finden des größten Wertes.
- Finden des kleinsten Wertes.
- Durchschnitt der Wi-Fi-Signalstärke berechnen. Hierfür wird das neu berechnete Feld aus dem ersten Versuch benutzt.

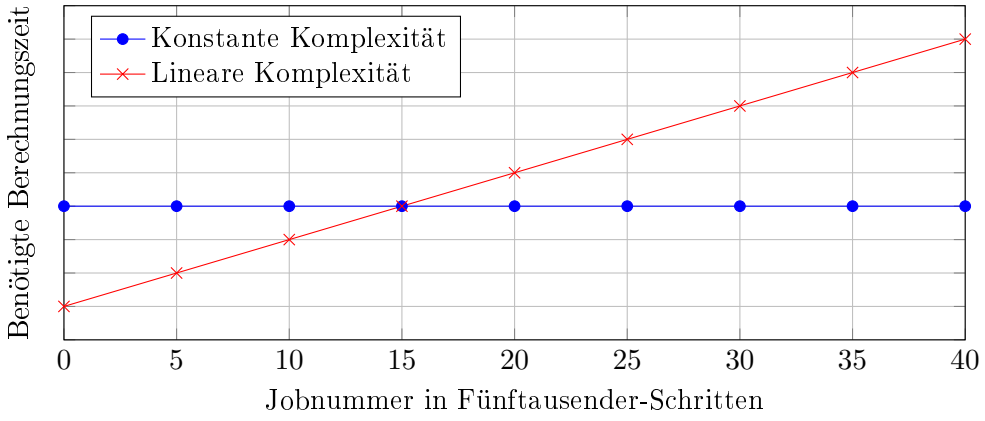

Abbildung 4.2: Erwartete Berechnungszeit für Spark Aggregationen

Die erste Erwartungshaltung an die Berechnungszeit ist, dass diese linear ansteigend verläuft. Die Notwendigkeit alle Einträge in einer Schleife zu durchlaufen, entspricht einer Komplexität von $\mathcal{O}(n)$. Eine einfache Umsetzung einer Durchschnittsberechnung würde alle Einträge aufsummieren und durch die Anzahl der Einträge dividieren. Bei einer solchen Implementierung würde der Großteil der Berechnungszeit für das Aufsummieren der Einträge verwendet werden. Mit jeder neuen Nachricht wächst die Anzahl der Einträge linear. Dadurch sollte auch der Aufwand und die Berechnungszeit linear ansteigen (siehe Abbildung 4.2).

Jedoch verwendet Apache Spark den Arbeitsspeicher, um mithilfe von Zwischenergebnissen die Berechnungen zu optimieren. Damit die Größe des verwendeten Arbeitsspeichers gering gehalten wird, speichert Apache Spark nicht alle Einträge, sondern nur Zwischenergebnisse. Für das Beispiel der Durchschnittsberechnung wird die Summe sowie die Anzahl der bereits aufsummierten Einträge gespeichert. So wird für einen neuen Eintrag die Summe aufaddiert, die Anzahl erhöht und anschließend der Durchschnitt neu

berechnet. Dadruch wird die Schleife entfernt und es bleiben nur Operationen mit einer $\mathcal{O}(1)$ Komplexität übrig. Dieser Ansatz ist für alle drei Aggregationen anwendbar und die erwartete Berechnungszeit ist daher nicht mehr linear ansteigend, sondern konstant. Der Versuch gilt als erfolgreich, wenn diese konstante Berechnungszeit erreicht wird.

4.2.3 Versuch 3: Akkurate, benutzerdefinierte Aggregation

Neben den vordefinierten Aggregationen, ist es auch möglich benutzerdefinierte zu erstellen. Dies ist häufig nötig, um komplexere Probleme zu lösen. In diesem Versuch wird die zurückgelegte Distanz mithilfe eines Algorithmus berechnet, welcher eine ansteigende Berechnungszeit hat, je höher die Anzahl der vorhanden Nachrichten ist. Diese Art der Berechnung ist häufig im Batch-Layer der Lambda-Architektur zu finden, kann aber auch im Streaming-Layer eingesetzt werden. Das ist aber nur möglich, wenn der Anstieg der Berechnungszeit nicht im quadratischen Bereich liegt. In einem solchen Fall können die zeitkritischen Anforderungen des Streaming-Layers nicht erfüllt werden.

Zur Berechnung der Distanz zwischen zwei dreidimensionalen Punkten wird folgende Formel benutzt:

$$d = \sqrt{(x2 - x1)^2 + (y2 - y1)^2 + (z2 - z1)^2}$$

Die Daten hierfür werden dem *odomData* Feld entnommen.

Für jeden neuen Eintrag wird die Distanz erneut berechnet, indem über die komplette Liste der Positionskoordinaten iteriert wird. Da der hierfür notwendige Zugriff auf die vollständige Liste gewährleistet ist, fallen die Berechnungen einfach aus und entsprechen der Art und Weise, wie Daten auch im Batch-Layer berechnet werden. Zusätzlich wird der Timestamps sichergestellt, dass die Einträge in der Liste die richtige Reihenfolge haben. Anderenfalls könnten falsch sortierte Einträge, welche durch die verteilte Natur der Daten und Berechnungen entstehen können, dass Ergebnis der Distanz verfälschen. Da der Prototyp nur auf einem einzelnen Server läuft, ist die falsche Reihenfolge der Daten selten. Jedoch ist für die Skalierung des Systems wichtig, dass diese Art der Berechnungen möglich ist. Das führt zwar zu einem akkuraten Ergebnis, benötigt aber mehr Arbeitsspeicher, um das Zwischenergebnis zu speichern.

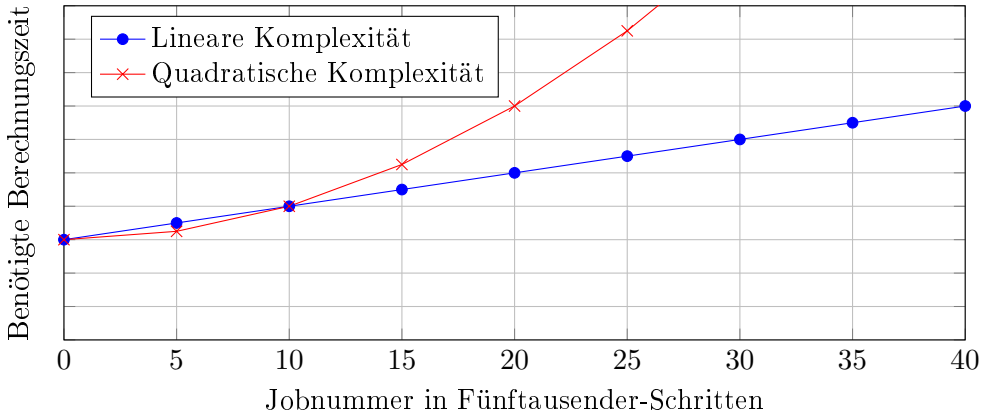

Abbildung 4.3: Erwartete Berechnungszeit für akkurate Aggregationen

Das Iterieren über die Liste aller Einträge führt erwartungsgemäß zu der Annahme einer $\mathcal{O}(n)$ Komplexität (siehe Abbildung 4.3). Hinzu kommt das Sortieren der Liste mit *Timsort*, welches im *Worst-Case-Szenario* eine Komplexität von $\mathcal{O}(nlogn)$ hat. Allerdings wird nicht für jeden Eintrag die komplette Liste sortiert, sondern die sortierte Liste als Zwischenergebnis übergeben. Deshalb muss nur der neueste Eintrag einsortiert werden, was die Komplexität auf $\mathcal{O}(n)$ reduziert. Für den Versuch wird daher eine linear ansteigende Berechnungszeit erwartet.

4.2.4 Versuch 4: Inakkurate, benutzerdefinierte Aggregation

Der vierte Versuch hat dieselbe Aufgabe wie der dritte und berechnet die zurückgelegte Distanz zwischen zwei Punkten. Jedoch werden in diesem Fall nicht alle Einträge gespeichert, sondern lediglich der letzte sowie die Summe der berechneten Distanzen. Durch diese Reduzierung kann nicht mehr sichergestellt werden, ob die Einträge in der richtigen Reihenfolge sind. Dadurch kann es zu inakkuraten Ergebnissen kommen, wenn Werte nicht in der korrekten Reihenfolge auf einem Server vorliegen. Da über keine Listen iteriert werden muss, sondern nur einfache Operationen durchgeführt werden, ist die Komplexität $\mathcal{O}(1)$. Der Gewinn an Geschwindigkeit durch den Verlust von Genauigkeit ist in der Lambda-Architektur vorgesehen, um schnellere Ergebnisse zu erhalten. Dieses Mittel wird häufig eingesetzt, um langsame Berechnungen, wie in Versuch drei, im Streaming-Layer zu ersetzten. Wichtig dabei ist, dass die Berechnungen eine konstant schnelle Berechnungszeit haben.

5 Evaluation des Designs

In diesem Kapitel werden die Ergebnisse der durchgeführten Versuche evaluiert. Dazu werden diese zunächst dargestellt und ausgewertet. Anschließend wird diskutiert was die Ergebnisse für die Umsetzung der Lambda-Architektur bedeuten. Zur Veranschaulichung der Ergebnisse werden diese als Graph dargestellt. Diese haben auf der x-Achse die Job-ID, welche sich mit jeder neuen Berechnung erhöht. Je höher die Job-ID, desto später wurde diese ausgeführt und umso mehr Nachrichten sind bereits in der Lambda-Architektur vorhanden. Die höhere Anzahl an gespeicherten Nachrichten hat einen größeren Einfluss auf Aggregationen, die alle Nachrichten für ihr Ergebnis benötigen. Das führt zu einer steigenden Berechnungszeit. Auf der y-Achse wird die Berechnungszeit des dazugehörigen Jobs in Millisekunden dargestellt. Die Messung startet mit dem Einlesen der Nachricht und endet mit dem Schreiben der Ergebnisse.

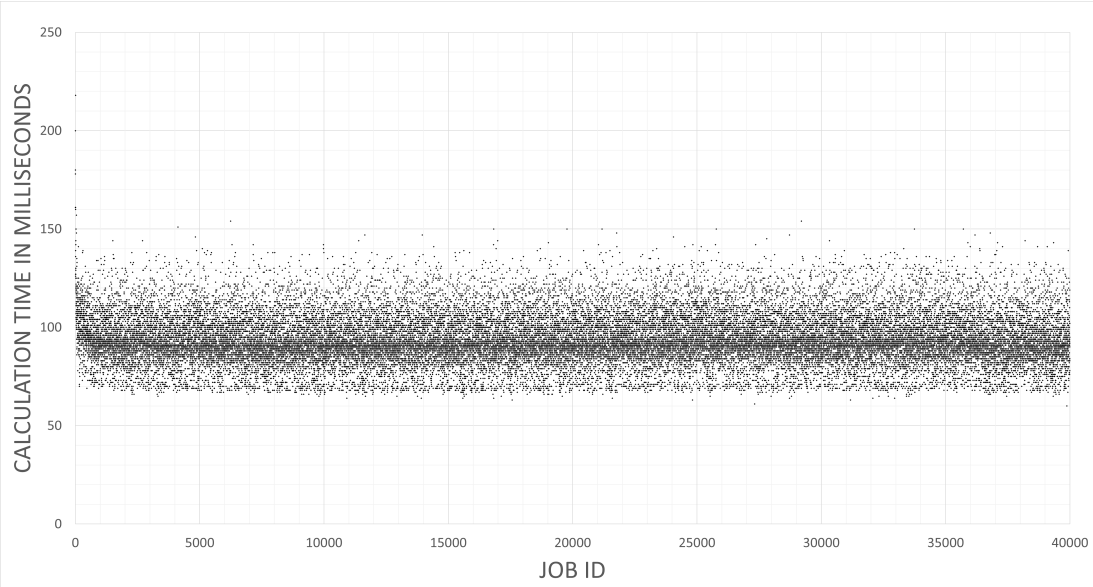

Abbildung 5.1: Berechnungszeit zum Einlesen und Normalisieren der Daten

5.1 Versuchsergebnisse

Die folgenden Werte wurden aus dem Apache Spark Monitoring Werkzeug entnommen. Zur besseren Lesbarkeit der Daten variiert die Größe der y-Achse.

5.1.1 Versuch 1: Einlesen von Daten und Normalisierung

Die Werte zum ersten Versuch (beschrieben in Kapitel 4.2.1) zeigen ein sehr konstantes Ergebnis und entsprechen damit den Erwartungen (siehe Abbildung 5.1). Der Großteil der Werte liegt zwischen 50 und 150 Millisekunden und hat nur eine geringe Streuung. Dabei ist kein Anstieg oder Abfall der Berechnungszeit bei späteren Jobs zu erkennen und es gibt keine Jobs mit auffällig hoher Berechnungszeit. Die größte Anhäufung an Messwerten ist auf der Höhe von 90 Millisekunden zu finden. Im Vergleich mit den anderen Versuchen weist der erste Versuch eine besonders niedrige Berechnungszeit auf. Selbst zum nächst schnelleren, vierten Versuch, beträgt die Differenz mehrere Tausend Millisekunden.

5.1.2 Versuch 2: Spark Aggregationen

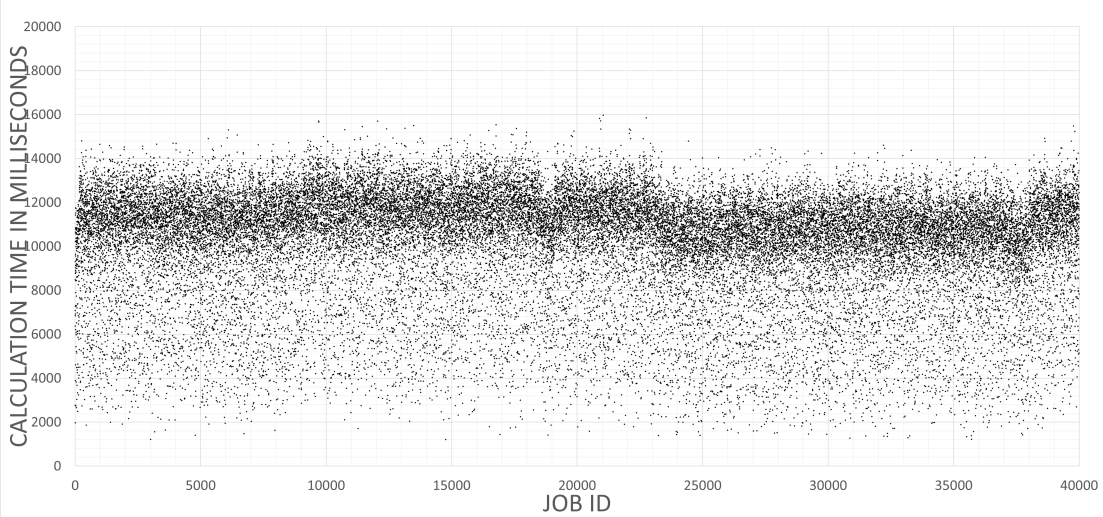

Abbildung 5.2: Berechnungszeit der Spark Aggregationen

Beim zweiten Versuch (siehe Kapitel 4.2.2) liegen die Minimal- und Maximalwerte sehr weit auseinander. Während die kürzesten Zeiten bei etwa 2000 Millisekunden liegen, benötigen die längsten Berechnungen 16000 Millisekunden (siehe Abbildung 5.2). Damit

hat der zweite Versuch die längsten Berechnungszeiten von allen Versuchen. Obwohl es einen weit gefächerten Wertebereich von 12000 Millisekunden gibt, liegen die meisten Werte bei etwa 11500 Millisekunden. Bei genauerer Betrachtung ist ein leichter Abfall der Berechnungszeit ab Job-ID 23000 zu sehen. Trotz der großen Variation in der Berechnungszeit entspricht das Ergebnis der Erwartung einer $\mathcal{O}(1)$ Komplexität. Jedoch ist bei diesem Versuch die Berechnungszeit konstant länger, als der Abstand zwischen gesendeten Nachrichten. Die gesamte Ausführungszeit war allerdings nicht erkennbar länger, als bei den anderen Versuchen und hat das viertägige Zeitfenster nicht überschritten.

5.1.3 Versuch 3: Akkurate, benutzerdefinierte Aggregationen

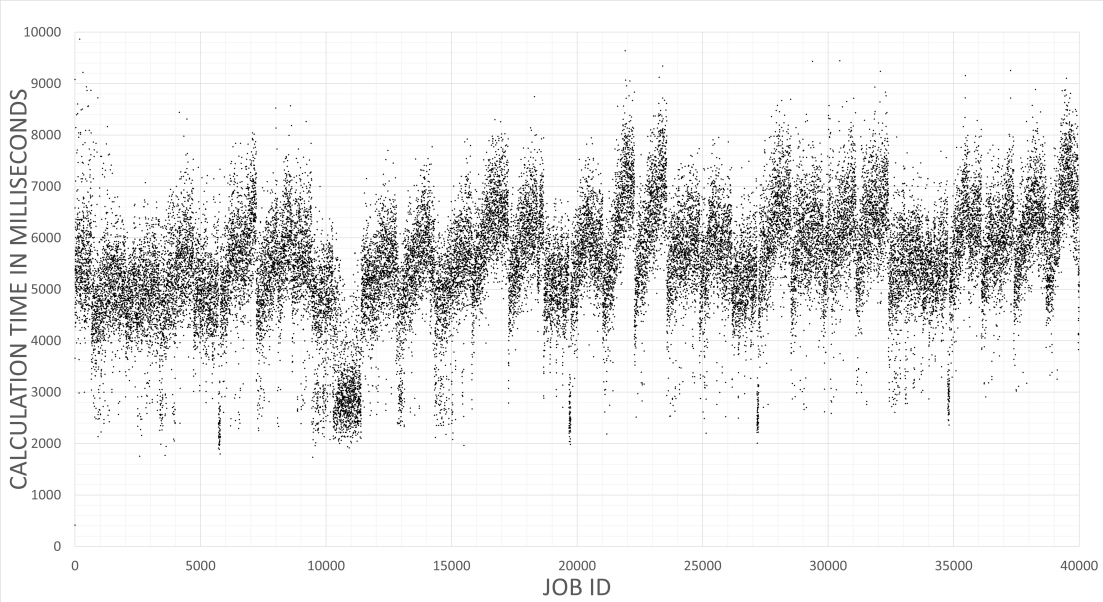

Abbildung 5.3: Berechnungszeit der akkuraten, benutzerdefinierten Aggregationen

Für den dritten Versuch (siehe Kapitel 4.2.3) wurde eine linear ansteigende Berechnungszeit erwartet und diese findet sich auch im Ergebnis wieder. Bei den ersten Jobs liegt das Mittel der Werte um die 4900 Millisekunden, während es zum Ende hin 6200 und mehr Millisekunden beträgt (siehe Abbildung 5.3). Das ist ein Anstieg von ungefähr 1300 Millisekunden für 40000 Nachrichten. Die Trendlinie zeigt dies noch einmal deutlicher (siehe Abbildung 5.4). Der komplette Wertebereich erstreckt sich zwischen 2000 und 9000 Millisekunden, wobei Berechnungszeiten von unter 4000 Millisekunden selten sind. Bei diesem Versuch ist neben der Berechnungszeit auch der benötigte Festplattenspeicher von Be-

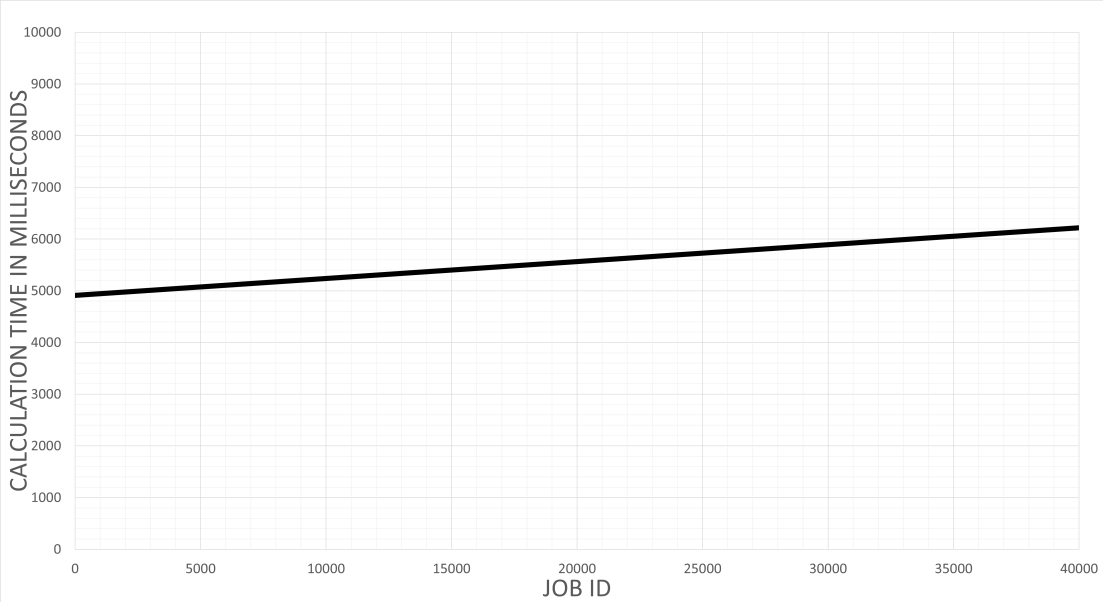

Abbildung 5.4: Trendlinie der akkuraten, benutzerdefinierten Aggregationen

deutung, welcher die Erwartungen überschritten hat. Gegen Ende des Versuches wurden zwei Terabyte an Speicher verwendet.

5.1.4 Versuch 4: Inakkurate, benutzerdefinierte Aggregationen

Der vierte Versuch (siehe Kapitel 4.2.4) entspricht einer schnelleren Variante des dritten Versuches. Ein Anstieg der Berechnungszeit im späteren Verlauf ist hier nicht festzustellen, ansonsten weisen beide Versuche aber gemeinsame Merkmale auf. Der Wertebereich erstreckt sich hier zwischen 1500 und 10000 Millisekunden, während das Mittel bei etwa 5000 Millisekunden liegt (siehe Abbildung 5.5).

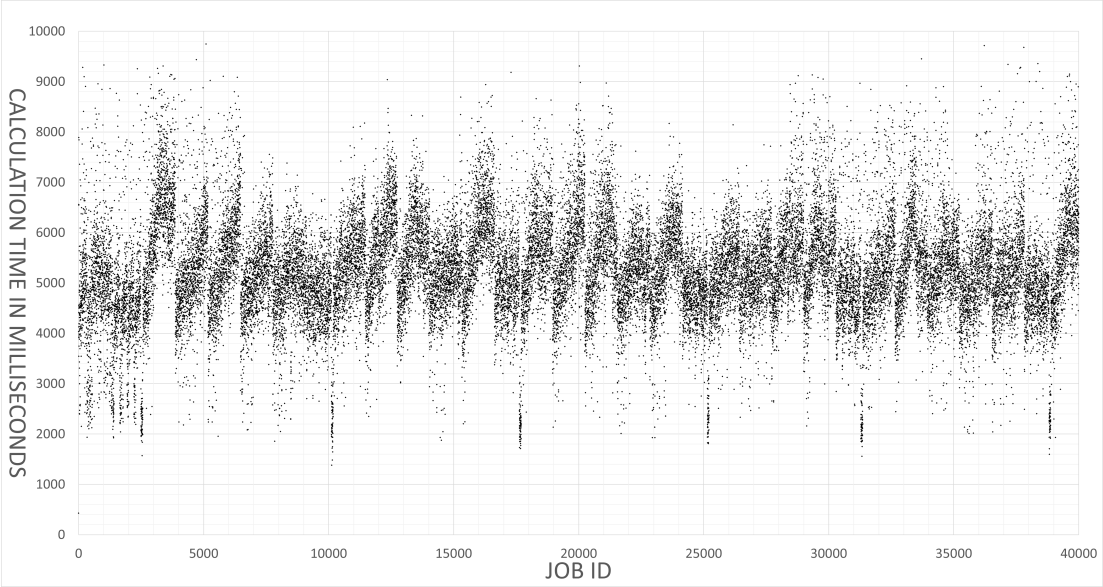

Abbildung 5.5: Berechnungszeit der inakkuraten, benutzerdefinierten Aggregationen

5.2 Diskussion der Ergebnisse

Das Ziel der Versuche war es die Leistungsfähigkeit einer Lambda-Architektur zu zeigen. Dabei wurden Aufgabenbereiche abgedeckt, die üblicherweise häufig bewältigt werden müssen. Der erste Versuch deckt die Notwendigkeit ab Daten einlesen zu müssen. Die Auswertung des Versuches zeigt, dass diese Aufgabe gut bewältigt wird und nur sehr geringe Berechnungszeiten benötigt werden. Mit einem Durchschnittswert von 90 Millisekunden ist der Aufwand so gering, dass er die Aggregationen kaum beeinflusst. Da der Versuch nur auf einem einzelnen Server durchgeführt wurde, gibt es keine Netzwerklatenz und der Großteil der Zeit wurde von Apache Spark verwendet. Wenn eine Skalierung des Systems durchgeführt wird, ist damit zu rechnen, dass sich die Netzwerklatenz deutlich erhöht. Dies konnte im Rahmen der durchgeführten Versuche jedoch nicht gezeigt werden.

Im zweiten Versuch werden die integrierten Aggregationen von Apache Spark untersucht. Trotz der insgesamt höchsten Berechnungszeit werden die Aufgaben effizient erledigt. Es ist kein Anstieg in der Berechnungszeit zu sehen ist. Damit sind die eingebauten Aggregationen von Spark, welche viele einfache Probleme lösen, für den Einsatz im Streaming-Layer geeignet. Die hohe Berechnungszeit lässt sich auf die drei Aggregationen zurückführen, während im dritten und vierten Versuch nur eine Aggregation durchgeführt wird. Dies könnte auch zu den großen Unterschieden zwischen der minimalen und maximalen Berechnungszeit beigetragen haben. Denn statt einem werden hier gleich drei Ergebnisse in einer Zeit von durchschnittlich 11500 Millisekunden berechnet. Wird zum Vergleich die Berechnungszeit des vierten Versuches auf das Dreifache hochgerechnet, so ergibt sich im Mittel eine Zeit von 15000 Millisekunden. Das sind 3500 Millisekunden mehr als die Berechnungszeit des zweiten Versuches. Die Berechnungszeit des zweiten Versuches liegt zwar über der Sendegeschwindigkeit von 8 Sekunden, jedoch ist Apache Spark in der Lage mehrere Nachrichten in einem Arbeitsschritt zu verarbeiten, was den Datendurchsatz erhöht. Dadurch entsteht kein Backlog und die Aufgabe kann gut bewältigt werden. Weiterhin ist beim zweiten Versuch ein leichter Abwärtstrend in der Berechnungszeit zu erkennen. Einerseits könnte es an den sich weniger verändernden Minimal- und Maximalwerten liegen. Andererseits könnte es aber auch nur ein kurzfristiger Trend sein, da es im Bereich um Job-ID 23000 (siehe Abbildung 5.2) einen sehr auffälligen Abfall gibt. Dieser könnte durch eine Veränderung der Serverlast entstanden sein.

Der dritte und vierte Versuch sind zwei Extrembeispiele für benutzerdefinierte Aggregationen. Häufig reichen Standard-Aggregationen zur Lösung komplexer Probleme nicht aus weshalb benutzerdefinierte Aggregationen zum Einsatz kommen. Im dritten Versuch wird bei jeder Berechnung eine Liste von Daten vollständig durchlaufen, was zu einer ansteigenden Trendlinie führt (siehe Abbildung 5.4). Der Anstieg ist der wachsenden Anzahl an gespeicherten Daten zuzuschreiben, welche bei jedem neuen Job höher ist. Die große Zahl an gespeicherten Zwischenergebnissen führt auch zu einem extremen Anstieg des benötigten Festplattenspeichers im Vergleich zu den restlichen Versuchen. Der große Speicherbedarf entsteht durch alte Zwischenergebnisse und Checkpoints, welche zur Wiederherstellung bei Fehlern verwendet werden. Um Extremfälle wie bei diesem Versuch zu vermeiden, können Bereinigungsfunktionen von Apache Spark aktiviert werden.

Apache Spark Streaming, das zur Umsetzung des Streaming-Layers benutzt wird, ist nicht in der Lage den kompletten Datensatz im Speicher zu halten. Um Funktionen, wie beispielsweise das Sortieren zu verwenden, ist es notwendig die Datenmenge zunächst durch eine Gruppierung zu verringern und dann zu sortieren. Dadurch wird der hohe

Speicherbedarf, welcher im dritten Versuch zum Problem wurde (siehe Kapitel 5.1.3), vermieden. Dieser Vorfall zeigt, dass sich Spark Streaming für Operationen auf kompletten Datensätzen nicht gut eignet. Das bedeutet jedoch nicht, dass der Einsatz von ihnen unmöglich ist. Da der Streaming-Layer die Daten nicht dauerhaft hält, sondern nur bis zur nächsten Neuberechnung des Batch-Layers, muss nur diese kurze Zeit überbrückt werden. Die Reduzierung des Datensatzes, durch den Batch-Layer, macht die Umsetzung möglich. Ein Vorteil dabei ist, dass der gleiche Algorithmus sowohl für den Streaming-, als auch für den Batch-Layer verwendet werden kann, was die Umsetzung vereinfacht. Außerdem sind Algorithmen, die den kompletten Datensatz zur Verfügung haben in der Regel einfacher umzusetzen.

Der vierte Versuch nutzt den umgekehrten Ansatz und rechnet zugunsten von schnelleren Ergebnissen ungenauer und ohne die korrekte Reihenfolge. Die Ergebnisse haben zwar eine konstante Berechnungszeit, allerdings hat ein auffällig großer Teil der Jobs etwa 5000 Millisekunden benötigt. Dies ist, in Anbetracht des Aufwands der Berechnungen, ein hoher Wert und zeigt, dass Spark Streaming hier einen Großteil der Zeit dafür benutzt die Durchführung der Berechnungen zu planen. Für den Prototyp ist diese Berechnungszeit jedoch kein Problem, da die Daten auf einem Dashboard dargestellt werden, welches sich maximal alle fünf Sekunden aktualisiert. Eine Möglichkeit, um die Berechnungszeit zu verringern und dabei die Lambda-Architektur beizubehalten, wäre der Einsatz von Apache Storm. Dieses ist auf die Verarbeitung von Realtime-Daten (*one-at-a-time*) ausgelegt, während Apache Spark das *Micro-Batch* Verfahren benutzt und dadurch schlechtere Ergebnisse erreicht.

6 Zusammenfassung

Die Lambda-Architektur ist Nathan Marz Antwort auf die steigende Anzahl an Daten und die damit verbundene Komplexität. Sie soll besonders skalierbar, fehlertolerant und schnell sein. Um das *CAP-Theorem* zu umgehen, hat die Lambda-Architektur zwei Pfade für Daten. Dies ist zum einen der Streaming-Layer, welcher für aktuelle Daten zuständig ist und dabei hohen Wert auf *Availability* legt. Zum anderen gibt es den Batch-Layer, welcher ältere Daten verwaltet und diese über den Serving-Layer nach außen anbietet. Dabei fokussiert sich dieser Pfad auf die *Consistency* der Daten. Zusammen speichern und verarbeiten die beiden Pfade Daten und ermöglichen einen schnellen Zugriff auf diese.

Zur Umsetzung der Lambda-Architektur war eine Vielzahl an unterschiedlichen Technologien nötig. Für den Datentransfer wird Apache Kafka benutzt. Die Langzeitspeicherung der Daten wird mit dem verteilten Dateisystem Apache Hadoop realisiert. Die Verarbeitung der Daten erledigt Apache Spark und wird dabei sowohl im Batch-, als auch im Streaming-Layer verwendet. Um die verarbeiteten Daten letztlich auch anzubieten, wird Apache Druid eingesetzt. All diese Programme und Frameworks ermöglichten die Umsetzung der Lambda-Architektur.

6.1 Fazit

Die Umsetzung einer Lambda-Architektur ist mit derzeitigen Technologien sehr gut möglich. Apache Kafka und Apache Hadoop wurden schon von Nathan Marz eingesetzt und erlauben große Freiheiten beim Einlesen und Speichern von Daten. Der Einsatz von Apache Spark für Batch- und Streaming-Layer reduziert die benötigten Frameworks. Dadurch rücken die beiden Layer näher zusammen und es entfällt die Notwendigkeit für zwei unterschiedliche Projekte, was ein großer Kritikpunkt an der Lambda-Architektur ist.

Trotz dieser Vereinfachung gibt es bei der Umsetzung immer noch eine Vielzahl an Konzepten und Technologien, die zuerst verstanden werden müssen und damit für die hohe Komplexität der Lambda-Architektur sorgen. Ist dieses Hindernis aber erst überwunden, dann liefert die Lambda-Architektur gute Ergebnisse. Alle Ergebnisse der vier Versuche entsprachen den Erwartungen und hatten maximal einen linearen Anstieg in der Berechnungszeit, was die Leistungsfähigkeit der Lambda-Architecture zeigt.

Die Ergebnisse der Evaluation lagen alle im erwarteten Bereich, auch wenn die Aggregationen eine lange (5000 Millisekunden) Laufzeit unabhängig von der eigentlichen Aufgabe hatten. Für einen Großteil der zu bewältigenden Aufgaben, wie Berechnungen für Dashboards, sind die Laufzeiten trotzdem schnell genug.

Durch den kurzen Lebenszyklus des Streaming-Layers sind selbst unoptimierte Aggregationen, wie die des dritten Versuches, immer noch nutzbar. Das reduziert den Arbeitsaufwand, um die Aggregationen zu programmieren, da sie zum einen einfacher sind und zum anderen sowohl im Batch-, als auch im Streaming-Layer benutzt werden können.

6.2 Ausblick

Die Lambda-Architektur konnte mithilfe des Prototyps erfolgreich umgesetzt werden. Allerdings lief die Umsetzung auf nur einem einzelnen Server. Das bedeutet, dass der wichtige Aspekt der Skalierung im Rahmen dieser Arbeit nicht ausführlich beleuchtet werden konnte. Außerdem wurden die Technologien zum Speichern der Daten in der Evaluation ausgelassen. Somit ergeben sich in diesen Themenfeldern noch offene Fragen. Dazu gehört wie gut die Lambda-Architektur auf multiplen Servern skaliert und wie sehr die Skalierung ungenaue Werte, wie die des vierten Versuches, beeinflusst.

Literaturverzeichnis

[1] Cerezo, Felipe: Deconstructing the Lambda architecture: an experience report. In: *IEEE International Conference on Software Architecture Companion (ICSA-C)* (2019)

[2] Eric A. Brewer: *Towards robust Distributed Systems.* Juli 2000

[3] Foundation, Apache S.: *Apache Hadoop.* https://hadoop.apache.org/. 2006 - 2021

[4] Foundation, Apache S.: *Apache Kafka.* https://kafka.apache.org/. 2011 - 2021

[5] Foundation, Apache S.: *Apache Druid.* https://druid.apache.org/. 2012 - 2021

[6] Jae Hyeon Bae, und Sudhir T.: *Announcing Suro: Backbone of Netflixs Data Pipeline.* Dezember 2013. – URL https://netflixtechblog.com/announcing-suro-backbone-of-netflixs-data-pipeline-5c660ca917b6

[7] Jeffrey Dean, Sanjay G.: MapReduce: Simplified Data Processing on Large Clusters. (Oktober 2004). – URL http://static.googleusercontent.com/media/research.google.com/es/us/archive/mapreduce-osdi04.pdf

[8] Kreps, Jay: *Questioning the Lambda Architecture.* July 2014. – URL https://www.oreilly.com/radar/questioning-the-lambda-architecture/

[9] Kyumars Sheykh Esmaili, Philippe D. und: *Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper.* 8 Juni 2017

[10] Labs, Grafana: *Grafana.* https://hadoop.apache.org/. 2014 - 2022

[11] Lampson, Butler W.: Hints for Computer System Design. In: *Proceedings of the ninth ACM symposium on Operating systems principless* (Oktober 1983), S. 33–48

[12] Lin, Jimmy: The Lambda and the Kappa. In: *IEEE Internet Computing* 21 (Oktober 2017), S. 60–66

[13] Marz, Nathan: *ElephantDB*. https://github.com/nathanmarz/elephantdb. 2013

[14] Marz, Nathan: *How to beat the CAP theorem*. Oktober 2011. – URL http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html

[15] Nathan Marz, James W.: *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., April 2015. – URL https://www.manning.com/books/big-data. – ISBN 9781617290343

[16] Seth Gilbert und Nancy Lynch: Brewers Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. In: *ACM SIGACT News* Volume 33, Issue 2 (Juni 2002), S. 51–59

[17] Software, Pivotal: *RabbitMQ*. https://www.rabbitmq.com/. November 2021

[18] Spark, Apache: *Apache Spark*. https://spark.apache.org/. 2014 - 2021

[19] Twitter: *Investor Fact Sheet*. 2020. – URL https://s22.q4cdn.com/826641620/files/doc_financials/2020/q4/Q4_20__InvestorFactSheet.pdf

[20] Twitter, Backtype: *Apache Storm*. https://storm.apache.org/. 2011 - 2021

Glossar

Apache Spark Eine Engine für großflächige Datenanalysen.

conflict-free replicated data types Eine Datenstruktur, die es erlaubt Daten verteilt zu speichern und Inkonsistenzen mathematisch zu lösen.

Druid Datasource Entspricht im Allgemeinen einer Tabelle aus einer relationalen Datenbank.

Geo-Replication Das persistente Speichern von Daten über mehrere Cluster.

immutable Ein Objekt, das nicht mehr verändert werden kann, nachdem es erstellt worden ist.

Jimmy Lin Professor an der Universität von Waterloo.

Loomo Ein von Segway Inc. entwickelter selbstbalancierender Roboter.

monetarisierbaren, täglich aktiven Nutzer Die monetarisierbaren, täglich aktiven Nutzer, im Englischen auch monetizable daily active users, sind Nutzer, denen Werbung geschaltet werden konnte.

mutable Ein Objekt, das verändert werden kann, nachdem es erstellt worden ist.

Nathan Marz Erschaffer von Apache Storm und Erfinder der Lambda-Architektur.

Normalisierung Die Vereinheitlichung von Daten in ein gleichmäßiges Schema.

query Abfrage von Daten aus einer Datenbank.

queue Eine Queue, zu Deutsch Warteschlange, speichert kurzzeitig Daten, bis sie weiterverarbeitet werden können.

raw Daten, die sich in ihrem ursprünglichen Zustand befinden und nicht durch Algorithmen verändert worden sind.

replica Ein Replica ist das Duplikat eines Dienstes, welches beim Auftreten eines Fehlers das Original ersetzt.

shard Ein Shard ist ein Fragment, das einen Teil der Datenbank enthält.

Tiered Storage Eine Funktion, die es erlaubt Daten für die Langzeitspeicherung auszulagern.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original