Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

LAKE – Eine flexible Datenstromverarbeitungsarchitektur

Corinna Giebler

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Bernhard Mitschang
Betreuer/in:	Dipl.-Inf. Christoph Stach
Beginn am:	24. April 2017
Beendet am:	24. Oktober 2017
CR-Nummer:	D.2.11, H.2.4, H.2.8

Kurzfassung

Forschungsgebiete wie die Industrie 4.0 können nicht auf eine konstante Überwachung verschiedener Komponenten verzichten. Zahlreiche Sensoren sind nötig, um beispielsweise autonome Fabriken sicher betreiben zu können. Über diese Sensoren werden gewaltige Mengen an Daten verfügbar, die sowohl persistent gespeichert, als auch möglichst schnell verarbeitet werden sollen. Die Genauigkeit von Analysen steigt mit der Anzahl verarbeiteter Datensätze, wodurch auch die Verarbeitungszeit wächst. Gleichzeitig müssen aktuelle Informationen in Echtzeit zur Verfügung stehen.

Dieses Problem wird durch zwei Architekturen zur Datenverarbeitung adressiert: Die Lambda– und die Kappa–Architektur verbinden Datenstrom– und Stapelverarbeitung, um sowohl genaue Datenanalysen als auch Echtzeitverarbeitung zu ermöglichen.

Beide Architekturen haben allerdings ihre Schwachstellen. So können beispielsweise Ergebnisse der Stapelverarbeitung in Lambda nicht in der Datenstromverarbeitung verwendet werden oder der Ressourcenverbrauch in Kappa steigt mit den vorhandenen Datensätzen. Diese Arbeit stellt darum LAKE als Verbindung aus beiden Architekturen vor. LAKE adressiert die Schwachstellen und bietet die Möglichkeit, die Art der Verarbeitung jederzeit flexibel auf verschiedene Anwendungsfälle anzupassen.

Zusammen mit dem Konzept beschreibt diese Arbeit auch verschiedene Systeme, die für die Realisierung eines LAKE–Prototypen verwendet werden können. Zwei Implementierungen eines solchen Prototypen werden in dieser Arbeit mit Apache Flink und Apache Spark umgesetzt. Zudem wird am Beispiel einer Klassifikation von Datenobjekten die Flexibilität von LAKE gezeigt.

In der abschließenden Evaluation wird sichtbar, dass die Kombination aus Lambda– und Kappa–Architektur nicht nur die Schwächen der jeweiligen Architektur ausgleicht, sondern zudem weitere Möglichkeiten zur Datenverarbeitung bietet.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

1 Einleitung

Die zunehmende Vernetzung sogenannter *intelligenter Geräte* (engl. Smart Devices, siehe [SBG+15]) bringt neue Herausforderungen mit sich. Neben der Datenübertragung im *Internet der Dinge* (engl. Internet of Things, kurz IoT) steht auch die Datenverarbeitung vor neuen Problemen [SRF15].

In Bereichen wie der *Industrie 4.0* [SDF+13] oder auch der *eHealth* (dt. elektronische Gesundheitsfürsorge) [SBG+15] werden große Mengen verschiedenartiger Daten in kürzester Zeit erfasst [CBMS15]. Durch Messsensorik in Fabrikhallen und beim Patienten werden im Millisekundentakt neue Datensätze verfügbar. So misst beispielsweise ein Temperatursensor die aktuelle Arbeitstemperatur in einer Maschine, oder der Blutdruck eines Patienten wird in regelmäßigen Abständen dokumentiert. Diese Daten können für Langzeitanalysen oder aber für Warnsysteme verwendet werden, die verschiedene Kriterien betrachten und im Notfall reagieren. Bei einer Überhitzung in der Maschine kann diese so zeitnah abgeschaltet, oder bei einem Schlaganfall ein Notruf abgesetzt werden.

Für solche Warnsysteme ist es allerdings nötig, dass Daten in Echtzeit verarbeitet werden, um schnell auf Ereignisse reagieren zu können. Hierfür wird auf die *Datenstromverarbeitung* (engl. Stream Processing) zurückgegriffen [CY15; WGFR16]. Dabei werden die Daten in einem kontinuierlichen Datenstrom transportiert und bei ihrer Ankunft im System direkt verarbeitet [WGFR16]. So werden Wartezeiten reduziert und die Daten können schnell durch die Verarbeitungs–Pipeline hindurchgeführt werden.

Da in der Datenstromverarbeitung jedoch jedes Datum gesondert verarbeitet wird, sind so vorgenommene Analysen weniger exakt als in herkömmlichen Analysesystemen [CY15]. Die Genauigkeit der Analyseergebnisse steigt mit der Menge an einbezogenen Daten. Diese großen Mengen an Daten können durch die *Stapelverarbeitung* (engl. Batch Processing) effektiv verarbeitet werden [CY15]. Hierbei werden die Daten erst gesammelt und anschließend als große, zusammenhängende Menge analysiert, statt als einzelne Datenpakete. Diese Herangehensweise führt zu höheren Latenzzeiten, allerdings können Zusammenhänge (wie z.B. Verläufe) zwischen den Daten entdeckt werden.

In den zuvor genannten Szenarien werden beide Verarbeitungsarten benötigt. Per Datenstromverarbeitung können die Temperaturmessungen auf kritische Werte hin überprüft werden, während eine große Menge historischer Messdaten für eine Leistungsanalyse verwendet werden kann. Auch der Blutdruck eines Patienten kann per Datenstromverarbeitung überwacht werden, um im Notfall einen Arzt zu alarmieren. Die Blutdruckwerte können gleichzeitig an den

Hausarzt weitergeleitet werden, welcher eine Langzeitbetrachtung des Gesundheitszustandes durchführen will.

1.1 Aufgabenstellung

Um diese kombinierte Verarbeitung zu realisieren, gibt es bereits die Lambda- [MW15] und die Kappa–Architektur [Kre14]. Während diese Konzepte das grundsätzliche Problem auf verschiedene Arten lösen, besitzen sie dennoch mehrere Schwachstellen [WGFR16].

In der Lambda–Architektur werden Datenstrom- und Stapelverarbeitung getrennt voneinander durchgeführt. Hierfür gibt es zwei Schichten, die jeweils eine Verarbeitungsart realisieren [MW15]. Dies bedeutet allerdings, dass zwei verschiedene Systeme implementiert und gewartet werden müssen [Kre14].

Die Kappa–Architektur dagegen verarbeitet auch große Datenmengen als Datenstrom [Kre14]. Ändert sich die Verarbeitungslogik, wird der interne Speicher per Datenstromverarbeitung mit der neuen Logik ausgewertet. Dies führt allerdings dazu, dass bei einer solchen Änderung die alte Logik erst nach einiger Zeit vollständig abgelöst wird.

Ziel dieser Arbeit ist es darum, eine Datenstromverarbeitungsarchitektur zu erstellen, welche die beiden Ansätze kombiniert und so ihre Schwächen ausgleicht. Diese Architektur trägt den Namen Lambda–Kappa–Architektur, kurz *LAKE*. In den folgenden Kapiteln wird ein Konzept erarbeitet, welches anschließend als Prototyp implementiert wird. Dabei soll es möglich sein, die Datenverarbeitung flexibel auf verschiedene Anwendungsszenarien anzupassen, beispielsweise für Data Mining, Machine Learning oder Echtzeitverarbeitung. Dies geschieht unter Anderem durch einen Wechsel zwischen der Lambda- und der Kappa–Architektur, ohne dass hierfür das System gewechselt werden muss.

1.2 Gliederung

Die Arbeit ist in folgende Kapitel unterteilt:

Kapitel 2 – Anwendungsszenarien: Es gibt viele verschiedene Szenarien, in denen Datenstrom- und Stapelverarbeitung angewendet werden müssen. Beispielhaft werden darum in diesem Kapitel zwei Szenarien der Bereiche Industrie 4.0 und eHealth vorgestellt, für die eine kombinierte Datenstromverarbeitungsarchitektur benötigt wird.

Kapitel 3 – Anforderungen: Aus den zuvor erstellten Anwendungsszenarien entstehen gewisse Anforderungen an eine Datenstromverarbeitungsarchitektur, welche in diesem Kapitel als Anforderungskatalog vorgestellt werden.

Kapitel 4 – Verwandte Arbeiten: In diesem Kapitel werden zunächst Datenstrom- und Stapelverarbeitung als grundlegende Verarbeitungsarten beschrieben. Anschließend werden die Lambda- und die Kappa–Architektur näher behandelt. Es wird sowohl auf den allgemeinen Aufbau als auch auf die Vor- und Nachteile der Konzepte eingegangen. Zudem werden sie gegen die zuvor gestellten Anforderungen evaluiert.

Kapitel 5 – Konzept für LAKE: Dieses Kapitel beschreibt das Konzept für LAKE, welche die Lambda- und die Kappa–Architektur miteinander kombiniert. Das Konzept wird anhand der zuvor aufgestellten Anforderungen erstellt und kann mit verschiedenen Verarbeitungsmodi verwendet werden.

Kapitel 6 – Allgemeine Umsetzung: Nach dem konzeptionellen Entwurf ist es nötig, die allgemeine Umsetzung zu diskutieren. Hierbei wird festgelegt, wie die einzelnen Komponenten zusammenarbeiten, um das Gesamtkonzept zu realisieren.

Kapitel 7 – Techniken und Systeme: Bevor LAKE praktisch umgesetzt werden kann, müssen konkrete Techniken und Systeme ausgewählt werden, die zur Umsetzung genutzt werden sollen. Dieses Kapitel bietet eine Übersicht über verschiedene Umsetzungsmöglichkeiten. Abschließend werden diese verglichen und schließlich passende Techniken und Systeme für die Implementierung zweier Beispiele ausgewählt.

Kapitel 8 – Anwendungsbeispiele und konkrete Implementierung: Durch die verschiedenen Möglichkeiten, Datenstrom- und Stapelverarbeitung anzuwenden, ist LAKE besonders flexibel. In diesem Kapitel werden zwei Beispiele vorgestellt und implementiert, um zu verdeutlichen, wie die Möglichkeiten von LAKE praktisch angewandt werden können.

Kapitel 9 – Evaluation: Das Konzept von LAKE wird gegen die erstellten Anforderungen evaluiert. Auch die Anwendungsszenarien werden mit LAKE bearbeitet, um zu zeigen, dass die Architektur die gegebenen Probleme lösen kann.

Kapitel 10 – Zusammenfassung und Ausblick: Abschließend folgt eine Zusammenfassung der Arbeit sowie ein Ausblick auf mögliche zukünftige Arbeiten.

2 Anwendungsszenarien

Die Bedeutung des Themenbereiches „Big Data“ nimmt in vielen Unternehmen mehr und mehr zu [LIX17]. Durch das IoT werden die unterschiedlichsten Datenquellen miteinander verknüpft und immense Datenmengen erfasst [CBMS15]. Zudem findet das IoT in vielen verschiedenen Bereichen Anwendung.

Eine wichtige Rolle spielt das IoT in der *Industrie 4.0* [SDF+13]. In dieser neuen Industriegeneration geht es darum, die verschiedenen Teile eines Produktionsprozesses miteinander zu verknüpfen und so eine möglichst weitgehend autonome Fabrik zu schaffen. Dazu wird Sensorik in Fabrikhallen, Maschinen oder Werkzeugen verbaut, die mit Rechenzentren oder auch Smartphones und Tablets kommunizieren. Die Fabrik überwacht und regelt sich zum Großteil selbst.

Doch auch im Bereich der Gesundheitsfürsorge nimmt das IoT eine immer wichtigere Rolle ein [SBG+15]. In der sogenannten *eHealth* werden Ärzte und Patienten durch Smart Devices, also spezielle Messgeräte, die untereinander kommunizieren können, unterstützt. Bei diesen handelt es sich beispielsweise um intelligente Pillendosen, die Rückmeldung über die eingenommenen Medikamente liefern können [SBG+15]. Auch das Smartphone kann Gesundheitsdaten sammeln und diese zur Weiterverarbeitung versenden [GS17]. Da die medizinischen Daten direkt durch den Patienten erfasst werden, kann die Qualität der Versorgung steigen, während die Kosten sinken [SJDN06].

In gerade diesen beiden Bereichen ergeben sich vielfältige Anwendungsszenarien, Abschnitt 2.1 befasst sich dabei mit einem Szenario der Industrie 4.0, während Abschnitt 2.2 einen Anwendungsfall aus dem Bereich der eHealth vorstellt. Die Teilszenarien sind dabei mit „I“ für Industrie 4.0 bzw. „E“ für eHealth gekennzeichnet. Dabei beschreibt der Begriff *historische Daten* die in einem persistenten Datenspeicher vorhandenen Daten, während *Stromdaten* die Daten beschreiben, die zum aktuellen Zeitpunkt per Datenstrom eintreffen.

Diese Szenarien sind nur beispielhaft. Die in dieser Arbeit vorgestellten Konzepte können in beliebigen IoT–Szenarien verwendet werden.

2.1 Industrie 4.0

Dieses Szenario nimmt die Überwachung einer Fertigungsmaschine als Ausgangspunkt (siehe [Jes14; MG14; WSM+17]). Der beispielhafte Anwendungsfall von einem Temperatursensor innerhalb der Maschine aus.

Der Temperatursensor meldet in regelmäßigen, kurzen Abständen die aktuelle Temperatur an eine zentrale Verarbeitungseinheit, wo diese auf kritische Temperaturwerte und -schwankungen analysiert werden muss. Sollte die Temperatur innerhalb der Maschine einen bestimmten Wert erreichen, beispielsweise den Schmelzpunkt des verarbeiteten Materials, muss diese abgeschaltet werden.

Zudem werden Daten aus der Programmierung der Maschine genutzt, um den aktuellen Bearbeitungszustand zu erfassen. Dazu gibt das System, über das der Fertigungsvorgang gesteuert wird, Rückmeldung darüber, wie viele Teile bereits gefertigt wurden. So kann der Fortschritt eines Auftrages überwacht werden.

Die so gesammelten Daten können verschiedenen Zwecken dienen:

I 1
Erreicht die Temperatur innerhalb der Maschine einen kritischen Wert, so muss der wartende Mitarbeiter informiert werden, damit dieser eingreifen kann. Diese Information muss in Echtzeit verfügbar sein, da sonst sowohl die Maschine als auch die darin enthaltenen Werkstoffe Schaden nehmen können.

I 2
Über alle Maschinen der Fabrik hinweg sollen genaue Analysen der gemessenen Temperaturwerte erfolgen. Hierbei muss eine sehr große Menge an Daten akkurat ausgewertet werden, wobei Stromdaten nicht mit einbezogen werden müssen. Für die Analyse werden also nur historische Daten benutzt, welche in einem persistenten Datenspeicher vorliegen.

I 3
Es soll eine Übersicht erstellt werden, wie viele Teile eines bestimmten Typs die Maschine in den letzten zehn Jahren bis zum jetzigen Zeitpunkt gefertigt hat. Hierzu werden Daten alter Aufträge sowie der aktuelle Arbeitsstand der Maschine ausgewertet und anschließend kombiniert.

I 4
In den historischen Daten sind Informationen zu früheren Fertigungsaufträgen und deren Temperaturen enthalten. Ein aktueller Messwert soll mit passenden historischen Daten (z.B. gleiches Material) verglichen werden, um die einzelnen Messwerte besser einschätzen zu können.

I 5
Nach dem Auftreten eines Fehlers soll dessen Ursache gefunden werden. Dafür wird ein kleiner Teil historischer Daten mit einer gesonderten Logik verarbeitet, die sonst im System nicht angewandt wird. Der zu überprüfende Bereich besteht nur aus Daten des letzten Auftrags und der betroffenen Maschine. Aus den historischen Daten muss also ein Auszug erstellt werden, welcher dann gesondert und möglichst schnell verarbeitet werden soll.

I 6
Ein externer Datenspeicher enthält Informationen über die verwendeten Materialien. Dazu gehören auch Temperaturbereiche, in denen die Materialien gefahrlos verarbeitet werden können Die Stromdaten können mit diesen Informationen verknüpft werden, um für jedes Material eigene Grenzwerte für die Temperatur einhalten zu können.

Jedes dieser Teilszenarien nutzt verschiedene Sensoren und Daten als Grundlage. Auch die Art der Verarbeitung unterscheidet sich je nach Szenario. Tabelle 2.1 beinhaltet darum eine Übersicht über die Teilszenarien und deren Eckdaten. Die Häufigkeit gibt an, wie oft eine Messung durchgeführt wird. „Sehr häufig“ bedeutet, dass die Messungen im Untersekundentakt aufeinander folgen, während bei „selten“ Messpausen von mehreren Jahren auftreten können.

Szenario	Verwendete Daten	Datenquelle	Häufigkeit	Verarbeitungsart
I1	Temperatur	Datenstrom	Sehr häufig	Reine Echtzeitverarbeitung
I2	Temperatur	Historische Daten	Selten	Verarbeitung der historischen Date
I3	Teileanzahl	Datenstrom und historische Daten	Häufig	Echtzeitverarbeitung und Verarbeitung der historischen Daten mit anschließender Kombination der Ergebnisse
I4	Temperatur	Datenstrom und historische Daten	Sehr häufig	Erstellung eines Ausschnitts des Datenspeichers, anschließende Analyse des Datenstroms in Echtzeit mit dem Ausschnitt als Analysegrundlage
I5	Temperatur	Ausschnitt des Datenspeichers	Selten	Erstellung eines Ausschnitts des Datenspeichers, anschließende Weiterverarbeitung per Datenstromverarbeitung
I6	Temperatur	Datenstrom und externer Datenspeicher	Sehr häufig	Laden des externen Datenspeichers, anschließende Analyse des Datenstroms in Echtzeit mit den externen Daten als Analysegrundlage

Tabelle 2.1: Die Eckdaten zu den Verarbeitungsszenarien der Industrie 4.0

2.2 eHealth

Auch dieses Szenario basiert auf dem IoT. Gegenstand ist ein Sensor, den Epilepsie-Patienten im Ohr tragen können, um so mögliche Anfälle frühzeitig zu erkennen [Fra16]. Dabei werden Puls und Bewegungen erfasst und an das Smartphone des Patienten übertragen, welches die Daten an einen Computer weiterleitet. Hier werden die Daten auf Auffälligkeiten überprüft. Im Falle eines vermuteten Anfalls kann so schnell Hilfe alarmiert werden.

Auch hier sind verschiedene Verwendungszwecke möglich:

E 1
Sobald Puls und Bewegungsmuster auf einen möglichen Anfall hindeuten, muss ein Arzt verständigt werden. Die Alarmierung muss möglichst in Echtzeit erfolgen, damit der Patient schnellstmöglich Hilfe erhält. Hierbei wird auch der Aufenthaltsort des Patienten übermittelt, um den Arzt zu unterstützen. Da diese Ortsinformationen allerdings privat und für spätere Analysen irrelevant sind, sollen sie nicht persistent gespeichert werden.

E 2
Über mehrere Patienten und einen längeren Zeitraum hinweg sollen Analysen zu Puls- und Bewegungsdaten durchgeführt werden. Das Ziel ist, herauszufinden, zu welchen Zeiten und in welchen Abständen sich Anfälle ereignen. Dazu werden große Mengen historischer Daten ausgewertet, wobei Latenzzeiten zu vernachlässigen sind.

E 3
Für die Festlegung der weiteren Behandlung eines Patienten braucht es eine Auswertung historischer Daten und Stromdaten. Hierfür müssen Daten aus dem Datenspeicher und Stromdaten analysiert und die Ergebnisse kombiniert werden.

E 4
Um Stromdaten besser interpretieren zu können, müssen bestimmte historische Messungen ebenfalls berücksichtigt werden, beispielsweise aus einem angegebenen Zeitintervall. So können Verläufe und Änderungen besser erfasst werden.

E 5
Ein externer Datenspeicher beinhaltet Informationen zu Medikamenten. Diese Informationen können bei einer Analyse historischer Daten miteinbezogen werden, um so die Wirksamkeit verschiedener Medikamente bewerten zu können.

E 6
Auch hier sind Daten zu Medikamenten in einem externen Datenspeicher vorhanden. Über das Smartphone können die Patienten angeben, welche Medikamente eingenommen wurden. Im Ernstfall können die externen Daten mit den Stromdaten verknüpft werden und so einem Arzt die Behandlung zu erleichtern.

Wie bereits in Abschnitt 2.1 werden auch diese Teilszenarien noch einmal mit ihren Eckdaten aufgezeigt. Die Zusammenstellung findet sich in Tabelle 2.2.

Szenario	Verwendete Daten	Datenquelle	Häufigkeit	Verarbeitungsart
E1	Puls, Bewegung	Datenstrom	Sehr häufig	Reine Echtzeitverarbeitung
E2	Puls, Bewegung	Historische Daten	Selten	Verarbeitung der historischen Daten
E3	Puls, Bewegung	Datenstrom und historische Daten	Häufig	Echtzeitverarbeitung und Verarbeitung der historischen Daten mit anschließender Kombination der Ergebnisse
E4	Puls, Bewegung	Datenstrom und historische Daten	Sehr häufig	Erstellung eines Ausschnitt des Datenspeichers, anschließende Analyse des Datenstroms in Echtzeit mit dem Ausschnitt als Analysegrundlage
E5	Puls, Bewegung, Medikamente	Historische und externe Daten	Selten	Laden des externen Datenspeichers, anschließende Analyse der historischen Daten mit den externen Daten als Analysegrundlage
E6	Puls, Bewegung, Medikamente	Datenstrom und externer Datenspeicher	Sehr häufig	Laden des externen Datenspeichers, anschließende Analyse des Datenstroms in Echtzeit mit den externen Daten als Analysegrundlage

Tabelle 2.2: Die Eckdaten zu den Verarbeitungsszenarien der eHealth

3 Anforderungen

Aus den in Kapitel 2 erarbeiteten Anwendungsszenarien ergeben sich bestimmte Anforderungen an eine Datenstromverarbeitungsarchitektur. Die einzelnen Teilszenarien beschreiben dabei verschiedene Eigenschaften der Architektur, welche direkt auf Anforderungen abgebildet werden können. Im Folgenden sind diese aufgelistet. Zudem sind hinter den Anforderungen jeweils die Teilszenarien vermerkt, aus denen sich diese Anforderung ableitet.

- A1 (I1/E1) Daten können in Echtzeit verarbeitet werden, ohne dass (alle) Informationen persistent gespeichert werden müssen.
- A2 (I2/E2) Es sind Analysen möglich, die ausschließlich auf historischen Daten arbeiten.
- A3 (I3/E3) Es ist möglich, sowohl historische als auch Stromdaten zu analysieren und diese Ergebnisse zu einem Ergebnis zusammenzufassen. Hierfür können historische Daten vorverarbeitet und Stromdaten in Echtzeit verarbeitet werden, um möglichst schnell akkurate Ergebnisse zu erhalten.
- A4 (I4/E4) Ergebnisse aus der Verarbeitung historischer Daten können als Grundlage für die Verarbeitung der Stromdaten genutzt werden.
- A5 (I5) Für einen Bereich der historischen Daten ist es möglich, diesen mit einer veränderten Logik auszuwerten. Diese Auswertung geschieht mit nur wenig Latenzzeit. Dafür wird ein Ausschnitt des Datenspeichers erstellt, welcher anschließend erneut verarbeitet wird.
- A6 (E5) Historische Daten können mit beliebigen externen Daten verknüpft werden, um Beziehungen zwischen den Datensätzen zu ermitteln.
- A7 (I6/E6) Stromdaten können ebenfalls mit Daten aus externen Quellen verknüpft werden und mit ihnen zusammen verarbeitet werden.

Um alle Anforderungen erfüllen zu können, werden innerhalb der Datenstromverarbeitungsarchitektur verschiedene Verarbeitungsmodi benötigt. Diese Verarbeitungsmodi unterscheiden sich darin, welche Daten für die Verarbeitung verwendet werden und ob sie im Falle einer Kombination vor oder nach der Verarbeitung zusammengeführt werden. Abbildung 3.1 zeigt die verschiedenen Modi in einer Gesamtübersicht.

Die Pfeile in der Abbildung beschreiben den Datenfluss. Bei den verarbeiteten Daten handelt es sich entweder um historische Daten aus dem persistenten Datenspeicher, oder um Stromdaten. Unabhängig von der verwendeten Datenquelle sind diese Daten zunächst unverarbeitet. Externe

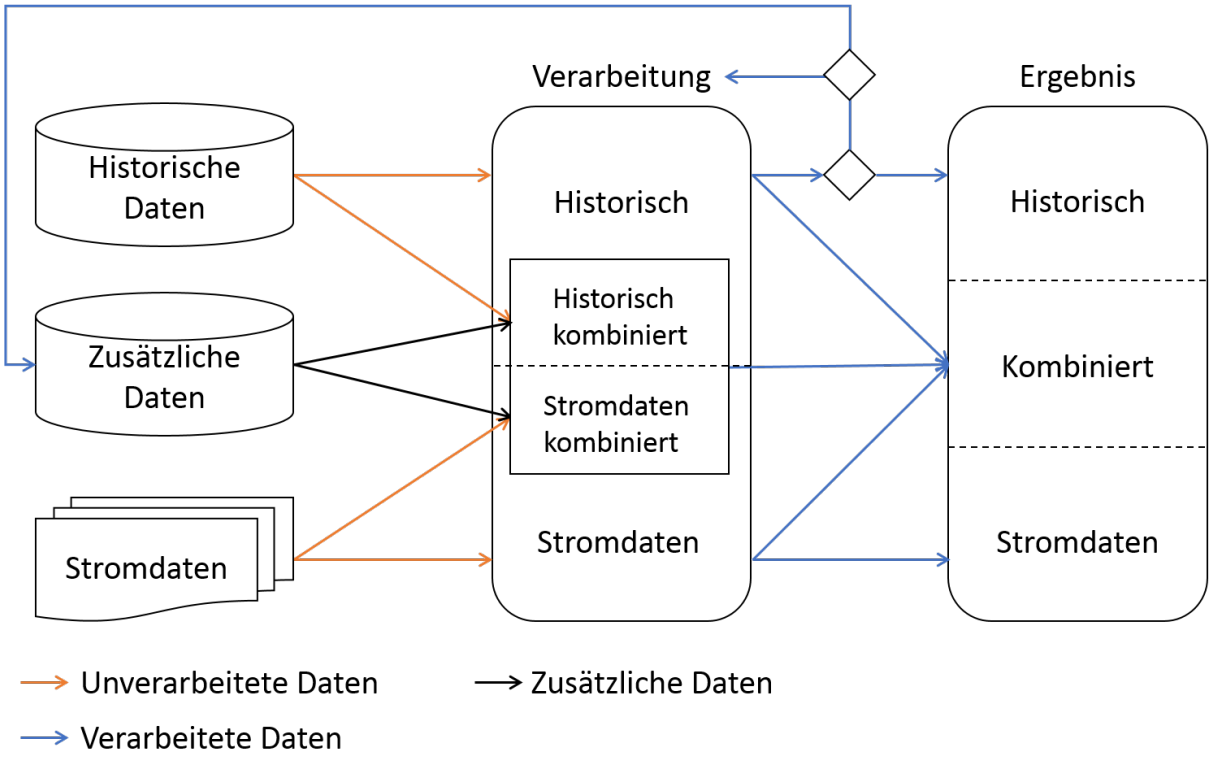

Abbildung 3.1: Die gewünschten Verarbeitungsmodi einer Datenstromverarbeitungsarchitektur

Daten dagegen werden nie direkt verarbeitet, sondern dienen stets nur als Grundlage für die Verarbeitung. Sie gehören damit zu den zusätzlichen Daten.

Die historischen bzw. Stromdaten können nun in beliebigen Kombinationen verarbeitet werden. So ist es für die Anforderungen A1, A3 und A7 nötig, dass die Stromdaten getrennt von den historischen Daten verarbeitet werden können. Die Anforderungen A2, A3, A5 sowie A6 dagegen benötigen eine Verarbeitung rein historischer Daten. Zusätzlich wird für A4 und A5 die Möglichkeit benötigt, eine Ergebnismenge erneut zu verarbeiten. Während dies in A5 auf beliebige Art und Weise geschieht, wird in A4 das Ergebnis mit anderen Daten kombiniert. Hierfür wird die Ergebnismenge als zusätzliche Daten eingebunden. Im Falle von A6 und A7 werden für die Verarbeitung dagegen externe Daten hinzugezogen.

Je nach Verarbeitungsmodus werden auch die Ergebnismengen auf unterschiedliche Weise erstellt. Für A1 und A7 besteht die Ergebnismenge nur aus Stromdaten, wobei im Falle von A7 auch externe Daten das Ergebnis beeinflussen. Eine Ergebnismenge aus historischen Daten dagegen ergibt sich bei der Erfüllung von A2, A5 und A6. Da für A4 historische und Stromdaten bereits vor der Verarbeitung kombiniert wurden, ist hier auch nur eine kombinierte Ergebnismenge möglich. Im Falle von A3 dagegen werden die verschiedenen Daten erst nach der Verarbeitung miteinander kombiniert.

Historische und Stromdaten müssen also sowohl getrennt als auch kombiniert verarbeitet werden können. Auch die Ergebnismengen müssen flexibel und je nach Anwendungsfall anders zusammengestellt werden können. Eine Datenstromverarbeitungsarchitektur kann erst alle Anforderungen erfüllen, wenn alle diese Verarbeitungsmodi unterstützt werden.

4 Verwandte Arbeiten

In diesem Kapitel werden zwei Architekturen vorgestellt, die sowohl die Verarbeitung historischer Daten als auch Stromdaten unterstützen und dabei geringe Latenzzeiten ermöglichen. Um den Aufbau der Architekturen nachvollziehen zu können, werden in Abschnitt 4.1 mögliche Verarbeitungsarten für große Datenmengen beschrieben. Die Architekturen selbst werden in Abschnitt 4.2 behandelt.

4.1 Verarbeitungsarten

Um große Mengen von Daten zu verarbeiten, können verschiedene Datenverarbeitungen verwendet werden [CY15]. Hierbei gibt es drei verschiedene Ansätze: Die Stapelverarbeitung, die Echtzeitverarbeitung und die hybride Verarbeitung.

Stapelverarbeitung Für die Verarbeitung als Stapel (engl. Batch) eignen sich statische Datenmengen [CY15]. Dies bedeutet, dass die Daten über längere Zeit gesammelt wurden und nun vollständig im Datenspeicher vorliegen. Während der Verarbeitung können keine neuen Daten oder sonstige Veränderungen in der Datenmenge nachträglich miteinbezogen werden.

Diese Daten werden anschließend verarbeitet, meist verteilt [CY15]. Um die Verarbeitung besonders effizient auszuführen, können die Daten zudem parallel verarbeitet werden. Allerdings kann sie weder unterbrochen, noch die Logik nach dem Start verändert werden. Zudem benötigt die Verarbeitung aufgrund der großen Datenmengen viel Zeit. Allerdings bietet die Stapelverarbeitung durch die hohe Zahl der verarbeiteten Daten auch eine hohe Genauigkeit der Ergebnisse.

Echtzeit-/Datenstromverarbeitung Um Daten so schnell wie möglich zu verarbeiten, kann darum auf Echtzeitverarbeitung zurückgegriffen werden [CY15]. Hierbei werden nicht die Daten nicht erst gesammelt, stattdessen werden Datenobjekte betrachtet, sobald sie zur Verfügung stehen [Has17]. Die Verarbeitung geschieht dabei auf einzelnen Datenobjekten oder auch auf kleinen Datenmengen, die man als Micro–Batches bezeichnet. Die zu verarbeitenden Datenmengen sind unbeschränkt [Has17], das bedeutet, dass noch während der Verarbeitung die Größe der Datenmenge unbekannt ist.

Durch die einzelne Verarbeitung der Datenobjekte sind Ergebnisse schneller verfügbar, auch wenn durch die unbeschränkten Datenmengen die Verarbeitung nicht von selbst endet [Has17]. Die Einzelverarbeitung führt allerdings dazu, dass die Datenobjekte getrennt voneinander betrachtet werden. So sind Zusammenhänge schwerer zu entdecken.

Tabelle 4.1 zeigt den Vergleich zwischen Stapel- und Echtzeitverarbeitung.

Art	Datenmenge	Verarbeitung	Dauer	Genauigkeit
Stapelverarbeitung	Statisch, unverändert	Große Datenmengen zusammen	Abhängig von Datenmenge	Hoch
Echtzeitverarbeitung	Unbeschränkt	Einzelne Datenobjekte oder Micro–Batches	Millisekundenbereich	Niedrig

Tabelle 4.1: Vergleich zwischen Stapel- und Echtzeit–/Datenstromverarbeitung

Hybride In manchen Fällen reicht die reine Stapel- oder Echtzeitverarbeitung nicht aus, da sowohl schnelle als auch genaue Ergebnisse gefordert sind [CY15]. In der hybriden Verarbeitung können die beiden Verarbeitungsarten auf verschiedene Weisen kombiniert werden. Die in Abschnitt 4.2 stellen zwei solche Kombinationen vor.

4.2 Die Architekturen

Die erste der beiden Architekturen ist die Lambda–Architektur (siehe Abschnitt 4.2.1), die 2011 von Nathan Marz beschrieben wurde [MW15]. Hier werden Datenstrom- und Stapelverarbeitung streng voneinander getrennt. Dagegen steht die Kappa–Architektur (siehe Abschnitt 4.2.2), die Jay Kreps 2014 als Alternative zur Lambda–Architektur vorstellte [Kre14]. Sowohl die Datenstrom- als auch die Stapelverarbeitung werden hier mit demselben System durchgeführt. Damit vermeidet die Kappa–Architektur die Nutzung zweier getrennter Systeme, die implementiert und gewartet werden müssen.

Beide Architekturen werden in den folgenden Abschnitten beschrieben und anhand eines Beispiels vorgeführt. Anschließend werden ihre Vor- und Nachteile betrachtet, bevor sie im Bezug auf die in Kapitel 3 erstellten Anforderungen evaluiert werden. Abschnitt 4.3 bietet ein Fazit zu den beiden Architekturen.

4.2.1 Die Lambda–Architektur

Bei der Lambda–Architektur handelt es sich um ein Konzept zur Umgehung des CAP–Theorems [MW15]. Laut diesem Theorem ist es unmöglich, in einem verteilten System mehr als zwei der folgenden Eigenschaften zu erfüllen [GL02]:

- Consistency – dt. Konsistenz, d.h. alle Knoten beinhalten für jedes Datum die gleichen Werte. Hierbei geht es um starke Konsistenz, die Werte müssen also zu jedem Zeitpunkt gleich sein.
- Availability – dt. Verfügbarkeit, d.h. das System kann zu jedem Zeitpunkt Anfragen erhalten und bearbeiten.
- Partition tolerance – dt. Partitionstoleranz, d.h. auch der vorübergehende Verlust einer Verbindung zu einem Teil des Netzwerks beeinträchtigt die allgemeine Funktion des Systems nicht.

Dazu besteht die Architektur aus zwei getrennten Verarbeitungsschichten (engl. *Layer*), die in Abschnitt 4.2.1 näher beschrieben werden.

Verarbeitung

Abbildung 4.1 zeigt den Aufbau der Lambda–Architektur. Durchgezogene Pfeile zeigen hier den Datenfluss, während der gestrichelte Pfeil eine Anfrage an das System darstellt.

Die Daten erreichen die Lambda–Architektur als Datenstrom. Jedes ankommende Datum wird sowohl an den Batch als auch an den Speed Layer weitergeleitet und dort entsprechend verarbeitet. Wird eine Anfrage an die Architektur gestellt, werden die Ergebnisse aus Batch und Speed Layer kombiniert und daraus die entsprechenden Daten abgeleitet.

Batch Layer Im Batch Layer werden die Daten zunächst in ein Master–Dataset (dt. Originaldatensatz) eingefügt [MW15]. Dabei werden die Daten nicht überschrieben, sondern die Stromdaten einfach an das Master–Dataset angehängt. Das Master–Dataset bleibt dadurch bis auf das Hinzufügen unverändert, weshalb auch Fehler in der Verarbeitungslogik nicht die Integrität der Daten beeinflussen können.

Aus dem Master–Dataset heraus werden die Daten in regelmäßigen Abständen verarbeitet. Als Ergebnis entstehen dabei die *Batch Views*, die Ausschnitte des Master–Dataset beinhalten, welche z.B. bereits aggregiert wurden. Diese Vorverarbeitung geschieht als Stapelverarbeitung in regelmäßigen Abständen.

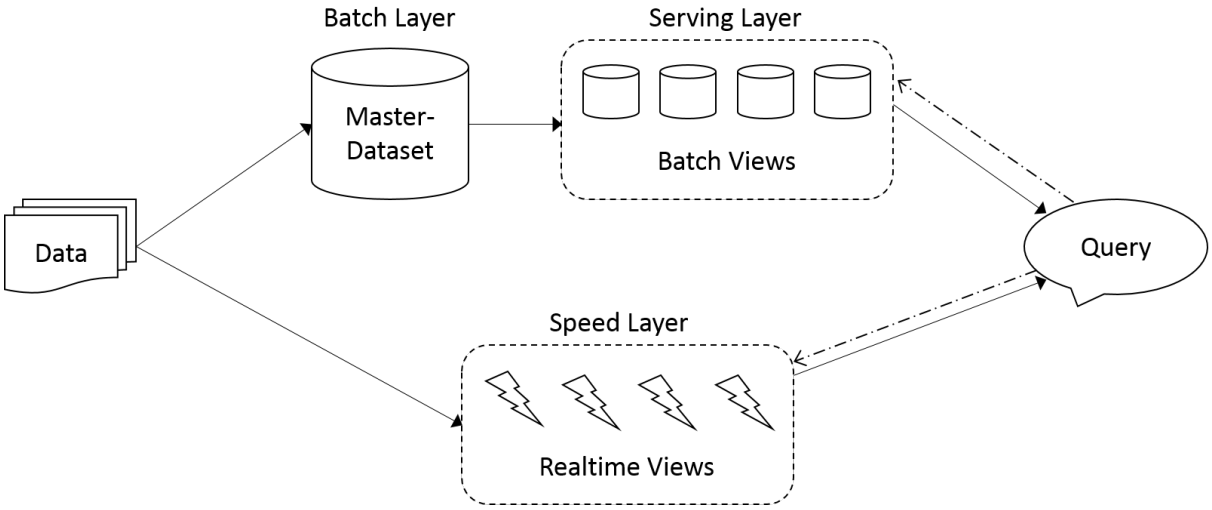

Abbildung 4.1: Die Lambda–Architektur (nach [MW15])

Speed Layer Da die Batch Views nur in bestimmten Abständen erstellt werden, enthalten sie unter Umständen nicht die aktuellsten Daten. Diese werden darum über den Speed Layer verarbeitet [MW15]. Hier werden ankommende Daten per Echtzeitverarbeitung behandelt und die Ergebnisse als Realtime Views verfügbar gemacht. Diese Realtime Views enthalten also die Daten, die noch nicht in einer Batch View repräsentiert werden.

Die Lambda–Architektur kann laut Marz das CAP–Theorem nun wie folgt schlagen [MW15]: Da die Daten im Master–Dataset unverändert bleiben und lediglich neue Daten angefügt werden, entsteht keine Inkonsistenz, da ein Datum entweder korrekt oder nicht verfügbar ist.

Der Batch Layer bietet zudem Verfügbarkeit (engl. Availability) und Partitionstoleranz (engl. Partitiontolerance), aber nur letztendliche Konsistenz (engl. Eventual Consistency), d.h. Daten brauchen einige Zeit, bis sie im Serving Layer als Ergebnismengen vorhanden sind [MW15]. Um dieses Problem zu beheben, verarbeitet der Speed Layer die neuesten Daten und bietet so eine Ergebnismenge, die starke Konsistenz (engl. Strong Consistency) garantieren kann. Durch die kombinierte Verarbeitung der Daten in beiden Schichten kann also das CAP–Theorem ausgehebelt werden.

Ein Beispiel

Als Beispiel dient das Anwendungsszenario aus dem Bereich der Industrie 4.0 (siehe Abschnitt 2.1), allerdings passend vereinfacht. Bei den ankommenden Daten handelt es sich um Informationen der Maschine darüber, wie viele Teile des aktuellen Auftrags bereits gefertigt wurden. Die Daten haben dabei die Form, die in Abbildung 4.2 als UML–Klassendiagramm dargestellt ist. Jedes Datum enthält eine Auftragsnummer, die Art des Teils, die Anzahl der

bereits gefertigten Teile sowie die Gesamtanzahl der zu fertigenden Teile für den Auftrag. Ziel ist es, die Anzahl der bisher gefertigten Teile pro Teileart auszugeben.

In einem Abstand von 24 Stunden werden die Batch Views jeden Abend im System erneuert. Dabei enthält das Master–Dataset die Daten von insgesamt fünf Jahren. Das bedeutet, dass die Batch Views zum Zeitpunkt der Verarbeitung die Zahl aller Teile enthalten, die seit Inbetriebnahme des Systems bis zum Vorabend gefertigt wurden. Die Daten wurden dabei bereits vorverarbeitet und enthalten nun nur noch Key–Value–Paare aus Teileart und der Anzahl gefertigter Teile.

Die Daten seit dem Vorabend dagegen werden durch den Speed Layer beigesteuert. In den Realtime Views befinden sich Daten, die bis zu 24 Stunden alt sind. Sie werden direkt nach ihrer Ankunft im System auf das selbe Datenformat gebracht wie die Daten in den Batch Views.

Um ein Gesamtergebnis zu erhalten, wird nun die passende Batch View mit der passenden Realtime View verknüpft. Diese Ergebnistabelle enthält alle Daten, die seit den letzten fünf Jahren im System eingetroffen sind.

Vor- und Nachteile

Im Gegensatz zu reinen Datenstrom- oder Stapelverarbeitungssystemen kombiniert die Lambda–Architektur beide Ansätze, um in möglichst wenig Zeit Anfragen an das System beantworten zu können. Sie ermöglicht es so, auch große Datenmengen effizient zu verarbeiten.

Ein weiterer positiver Punkt ist, dass Eingabedaten unverändert gespeichert werden [Kre14]. Das Master–Dataset erhält nur neue Daten, die niemals alte Einträge aktualisieren, sondern einfach angehängt werden [MW15]. Zudem werden sie auch nicht durch eine Verarbeitungslogik verändert und bewahren so ihre Integrität.

Durch die Berechnung der Batch Views werden die Daten im Serving Layer immer wieder aktualisiert, wodurch eine Änderung in der Logik für die Vorverarbeitung zu keinen Problemen

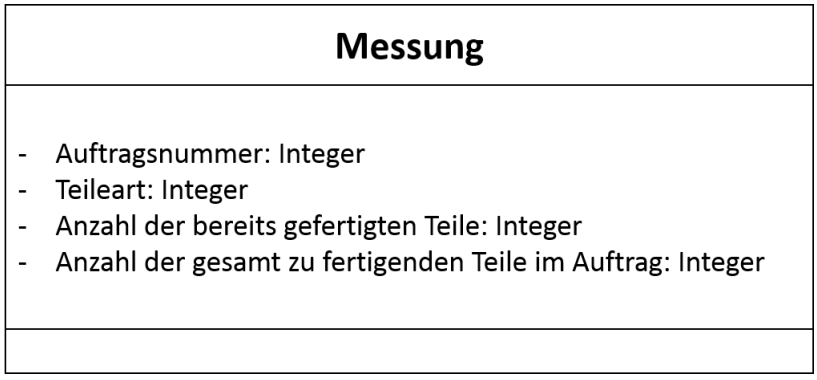

Abbildung 4.2: UML–Diagramm für einen Beispieldatentyp

führt [Kre14]. Stattdessen werden die Batch Views im nächsten Schritt mit der neuen Logik berechnet.

Zudem ist die Lambda–Architektur hoch skalierbar und kann einfach erweitert werden [MW15].

Kritisch zu betrachten ist jedoch die Tatsache, dass die Lambda–Architektur aus zwei verschiedenen Systemen besteht, die nicht miteinander in Kontakt stehen [Kre14]. Die Verarbeitungslogik muss sowohl für den Batch als auch für den Speed Layer separat geschrieben und verwendet werden. Dadurch können in Batch und Speed Layer verschiedene Verarbeitungsarten implementiert werden, falls dies gewünscht ist. Gleichzeitig aber müssen zwei Systeme parallel laufen und auch gewartet werden, was zu einem erhöhten Aufwand führt.

Evaluation

In diesem Abschnitt wird die Lambda–Architektur gegen die gegebenen Anforderungen evaluiert.

A1 ✓ In der Lambda–Architektur können eintreffende Daten in Echtzeit verarbeitet werden. Allerdings werden alle Daten, die eintreffen, im Master–Dataset gespeichert. Dies wäre allerdings durch kleinere Anpassungen zu ändern, darum gilt diese Anforderung als erfüllt.

A2 ✓ Die Verarbeitung rein historischer Daten kann in der Lambda–Architektur ohne weitere Anpassungen durchgeführt werden. Hierzu muss die Verarbeitung lediglich nur auf dem Batch Layer durchgeführt werden.

A3 ✓ Diese Anforderung beschreibt die Kernidee der Lambda–Architektur: Historische Daten werden als Stapel und Stromdaten in Echtzeit verarbeitet. Die Ergebnisse der beiden Verarbeitungen werden anschließend kombiniert. Diese Anforderung ist damit erfüllt.

A4 X In der Lambda–Architektur können die Ergebnisse aus der Stapelverarbeitung keinen Einfluss auf die Datenstromverarbeitung nehmen, da Speed und Batch Layer getrennt voneinander arbeiten. Diese Anforderung ist daher nicht erfüllt.

A5 X In der Lambda–Architektur ist es nicht vorgesehen, Batch Views erneut zu verarbeiten. Zudem ist die Anforderung, dass dies mit wenig Latenzzeit geschieht, nicht erfüllbar, da hierzu Speed und Batch Layer Daten austauschen müssten. Diese Anforderung ist dadurch nicht erfüllt.

A6 X Obwohl die Lambda–Architektur über ein Datenbanksystem im Batch Layer verfügt, ist unklar, ob die historischen Daten mit externen Daten verknüpft werden können. Zwar wäre dies theoretisch möglich, da aber in den Quellen eine solche Verknüpfung nicht thematisiert wird, gilt die Anforderung als nicht erfüllt.

A7 X Die Verknüpfung der Stromdaten mit externen Datenquellen ist in der Lambda–Architektur nicht möglich. Dies ist darauf zurückzuführen, dass die Stromdaten weder in einem Datenbankumfeld verarbeitet werden, noch kann ein zweiter Strom in die Datenstromverarbeitung einfließen. Diese Anforderung ist darum nicht erfüllt.

Die Abbildung 4.3 zeigt die von der Lambda–Architektur unterstützten Verarbeitungsmodi. Da die Daten in Batch und Speed Layer getrennt verarbeitet werden, ist eine kombinierte Verarbeitung nicht möglich. Allerdings können die Ergebnisse anschließend beliebig kombiniert oder auch einzeln betrachtet werden. Im Vergleich mit Abbildung 3.1 stellt die Lambda–Architektur jedoch nicht alle gewünschten Modi zur Verfügung. Sie ist somit für die gegebenen Anwendungsfälle nur bedingt geeignet.

4.2.2 Die Kappa–Architektur

Um das Problem der doppelten Verarbeitungslogik anzugehen, erstellte Jay Kreps 2014 die Kappa–Architektur [Kre14]. Entgegen der Lambda–Architektur werden eingehende Daten hier nicht auf zwei unterschiedliche Pfade, sondern in ein Stream Processing System geleitet, wo sowohl historische als auch Stromdaten verarbeitet werden.

Verarbeitung

Der Aufbau der Kappa–Architektur ist in Abbildung 4.4 dargestellt. Zunächst werden auch hier die Daten in einem unveränderlichen Datenspeicher abgelegt. Anschließend werden sie

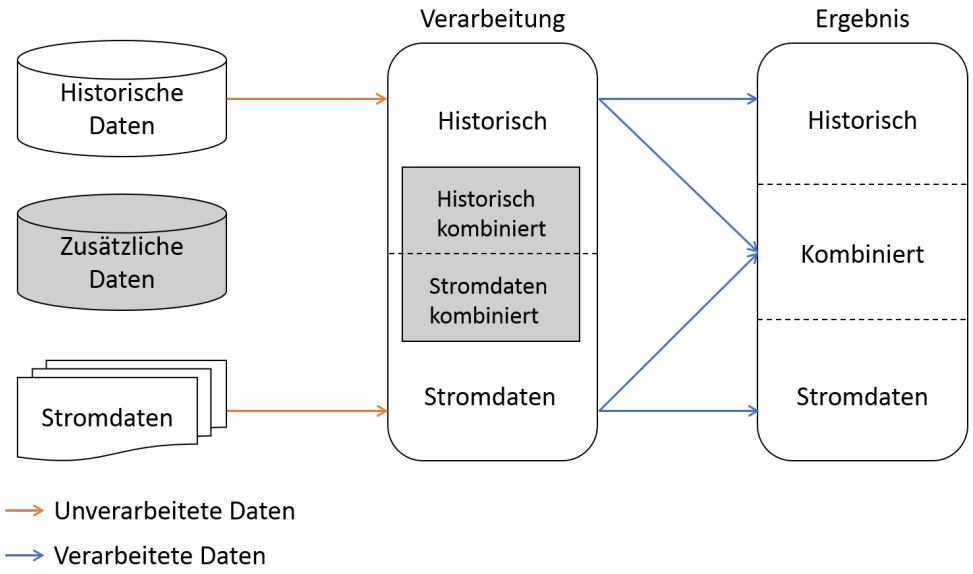

Abbildung 4.3: Die von der Lambda–Architektur unterstützten Verarbeitungsmodi

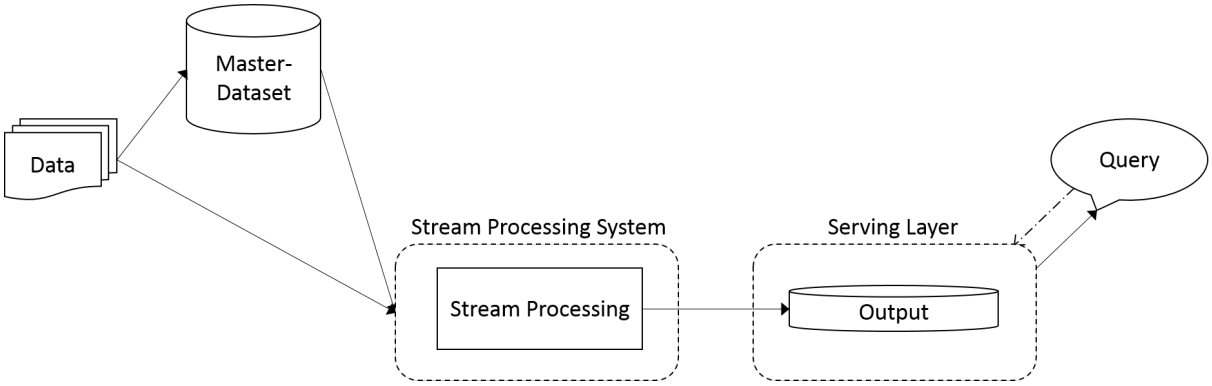

Abbildung 4.4: Die Kappa–Architektur (nach [Kre14])

direkt in das Stream Processing System weitergeleitet. Hier läuft ein Job, d.h. eine bestimmte Verarbeitungslogik, die die Daten behandelt. Die Ergebnisse werden als Ausgabetabelle in den Serving Layer geschrieben. Auf diesem können Anfragen ausgeführt werden.

Wird allerdings die Verarbeitungslogik geändert, müssen die historischen Daten erneut analysiert werden. Dies geschieht parallel zum oben beschriebenen Ablauf. Da es sich um eine begrenzte Datenquelle handelt, kann diese Verarbeitung als Stapelverarbeitung aufgefasst werden.

Die historischen Daten werden dabei aus dem Speicher geladen und als Eingabe für das Stream Processing System verwendet, allerdings mit neuer Verarbeitungslogik. Beide Jobs laufen parallel und schreiben ihre eigene Ausgabe in den Serving Layer. Erst, wenn der neue Job alle historischen Daten verarbeitet hat und auf dem gleichen Stand ist wie der alte, wird der alte Job beendet. Die Daten werden also nur im Fall einer Änderung der Verarbeitungslogik erneut verarbeitet. Diese Verarbeitungsweise ist in Abbildung 4.5 dargestellt.

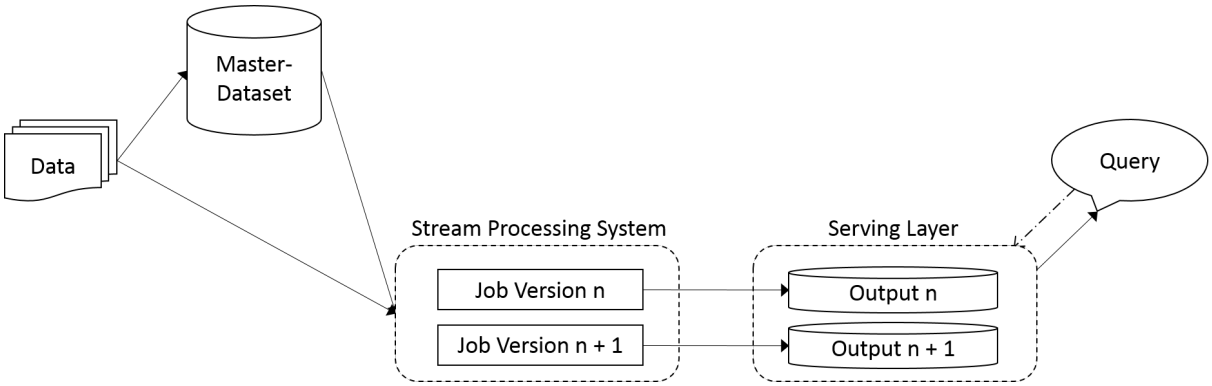

Abbildung 4.5: Die Kappa–Architektur mit doppelter Verarbeitungslogik (nach [Kre14])

Ein Beispiel

Wie in der Lambda–Architektur, kann auch in der Kappa–Architektur das Beispiel aus 4.2.1 genutzt werden. Es wird weiterhin das Datenformat aus Abbildung 4.2 verwendet.

Da in der Kappa–Architektur alle ankommenden Daten direkt verarbeitet und in den Serving Layer weitergegeben werden, entstehen keine Diskrepanzen zwischen den zugänglichen Daten im Serving Layer und den Stromdaten. Stattdessen ist jedes Datum in Echtzeit verarbeitet verfügbar. Es braucht darum keine unterschiedliche Behandlung für die Daten der letzten fünf Jahre oder der vergangenen 24 Stunden.

Vor– und Nachteile

Entgegen der Lambda–Architektur wird hier nur ein einzelnes System benötigt, welches schnell und einfach anpassbar ist [Kre14]. Anstatt kontinuierlich Batch Views zu berechnen, werden historische Daten nur dann erneut verarbeitet, wenn eine Änderung der Verarbeitungslogik das erforderlich macht.

Negativ ist jedoch zu bewerten, dass die Arbeitszeit für eine neue Berechnung linear mit der Zahl der Daten im Datenspeicher ansteigt [WGFR16]. So sind nach einer Änderung der Logik die neu verarbeiteten Daten erst nach einiger Zeit verfügbar. Zudem werden zwei Ausgaben im Serving Layer vorgehalten, während eine neue Logik angewandt wird, was den Speicherbedarf verdoppelt. Die parallele Berechnung des gesamten Datenstroms benötigt ebenfalls zusätzliche Kapazitäten. Die Kappa–Architektur skaliert also nicht für große Datenmengen. Ihre Anwendungsfälle liegen damit in Anwendungen, die nicht allzu viele historische Daten vorhalten und verarbeiten müssen.

Auch dauert es, bis eine Änderung der Logik in der Auswertung der Stromdaten sichtbar wird. Während zudem bei der Lambda–Architektur mehrere Batch Views für verschiedene Fragestellungen erstellt werden können, ist die Kappa–Architektur für die Verwendung einer einzelnen Logik ausgelegt, außer im Falle einer Neuberechnung.

Evaluation

Zuletzt werden die zuvor erarbeiteten Anforderungen für die Kappa–Architektur evaluiert:

A1 ✓ Die Stromdaten können in der Kappa–Architektur in Echtzeit verarbeitet werden. Wie bei der Lambda–Architektur können auch hier Informationen ausgefiltert werden, bevor die Daten im persistenten Speicher abgelegt werden. Die Anforderung ist dadurch erfüllt.

A2 ✓ Auch die reine Verarbeitung historischer Daten ist mit der Kappa–Architektur möglich. Hierfür muss der historische Datenspeicher die einzige Eingabequelle für die Kappa–Architektur sein.

A3 ✓ In der Kappa–Architektur können sowohl historische als auch Stromdaten verarbeitet werden. Die Ergebnisse dieser Verarbeitungen werden automatisch im Serving Layer zusammengefasst. Diese Anforderung gilt darum als erfüllt.

A4 (✓) Durch die Datenstromverarbeitung in der Kappa–Architektur wird jedes Datenobjekt einzeln betrachtet. Somit ist es nicht ohne weiteres möglich, die Ergebnisse bereits verarbeiteter Datenobjekte als Grundlage für die Verarbeitung der Stromdaten zu verwenden. Hierfür muss das Ergebnis der bereits verarbeiteten Daten in dem Stream Processing System zur Verfügung stehen. Da Kreps aber explizit erwähnt, dass Zwischenergebnisse für andere Datenverarbeitungen zur Verfügung stehen können, gilt diese Anforderung als teilweise erfüllt.

A5 X Die Berechnung einzelner Bereiche der historischen Daten ist in der Kappa–Architektur nicht möglich. Hierzu braucht es eine Form der Vorverarbeitung, die durch die Beschreibung der Architektur nicht gegeben ist. Diese Anforderung gilt darum als nicht erfüllt.

A6 X Nutzt man ein Datenbanksystem, so kann die Einbindung externer Quellen für die Verarbeitung historischer Daten möglich sein. Da aus den Quellen jedoch nicht ersichtlich ist, wie und ob diese Einbindung funktionieren würde, gilt diese Anforderung als nicht erfüllt.

A7 X Siehe Anforderung 6. Auch die Einbindung eines zweiten Datenstroms um diesen mit Stromdaten zu kombinieren wird in den Quellen nicht thematisiert. Somit gilt diese Anforderung als nicht erfüllt.

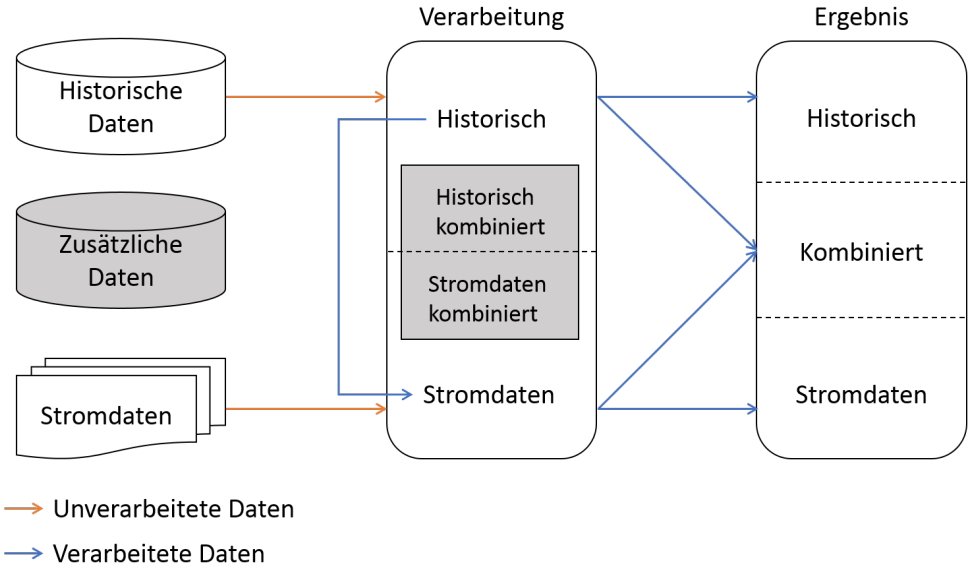

Abbildung 4.6: Die von der Kappa–Architektur unterstützten Verarbeitungsmodi

Wie bei der Lambda–Architektur werden auch die Verarbeitungsmodi der Kappa–Architektur näher betrachtet. Diese sind in Abbildung 4.6 dargestellt. Unter den passenden Voraussetzungen können hier sowohl historische als auch Stromdaten einzeln verarbeitet werden. Die Kombination mit zusätzlichen Daten dagegen ist auch hier nicht möglich. Allerdings kann, mit eventuellen Anpassungen, die Verarbeitung der historischen Daten die der Stromdaten beeinflussen und so bessere Ergebnisse ermöglichen. Trotzdem kann auch die Kappa–Architektur nicht alle benötigten Modi realisieren.

4.3 Fazit

In den vorangegangenen Abschnitten wurde deutlich, dass weder die Lambda– noch die Kappa–Architektur alle gewünschten Eigenschaften aufweisen. Durch die getrennten Verarbeitungsschichten der Lambda–Architektur sind Kombinationen zwischen historischen und Stromdaten nicht möglich. Die Kappa–Architektur dagegen bietet keine differenzierten Verarbeitungsmöglichkeiten für einzelne Abschnitte der gespeicherten Daten. Zudem benötigt sie für eine Änderung der Verarbeitungslogik Zeit und erhöhte Ressourcen. Tabelle 4.2 zeigt, wie die Anforderungen erfüllt wurden.

Anforderung	Lambda	Kappa
A1	✓	✓
A2	✓	✓
A3	✓	✓
A4	X	(✓)
A5	X	X
A6	X	X
A7	X	X

Tabelle 4.2: Auswertung der Lambda– und der Kappa–Architektur

5 Konzept für LAKE

Aus Kapitel 4 geht hervor, dass weder die Lambda- noch die Kappa–Architektur die gegebenen Anwendungsszenarien vollständig erfüllen können. Keine von beiden Architekturen ermöglicht die Kombination mit externen Daten. In der Lambda–Architektur sind keine Verknüpfungen zwischen historischen und Stromdaten vor der Analyse möglich, in der Kappa–Architektur dagegen sind diese beiden Bereiche untrennbar verbunden.

Um alle in Kapitel 3 gestellten Anforderungen erfüllen zu können, braucht es darum eine andere Architektur. Diese muss sowohl die Verarbeitungsmöglichkeiten der Lambda- als auch die der Kappa–Architektur unterstützen und diese erweitern. Dabei dürfen die Vorzüge der beiden Architekturen nicht verloren gehen, während ihre Schwächen ausgeglichen werden müssen. Die Architektur muss Daten also schnell und in großer Menge verarbeiten können, aktuelle Ergebnisse produzieren und darf dabei nicht übermäßig komplex zu implementieren und zu betreiben sein.

Dieses Kapitel beschreibt die Entwicklung dieser Architektur, die den Namen Lambda–Kappa–Architektur, kurz *LAKE*, trägt. Es handelt sich dabei um eine Kombination aus der Lambda- und Kappa–Architektur, welche darüber hinaus über zusätzliche Verarbeitungsmodi verfügt. Wie in der Lambda–Architektur ist es möglich, Daten getrennt als Datenstrom oder Stapel zu verarbeiten. Auch können die Daten, wie in der Kappa–Architektur, unabhängig vom Alter als ein zusammenhängender Datenstrom behandelt werden. Des Weiteren enthält die Architektur ein Datenbanksystem, welches mit externen Quellen verknüpft werden kann, sowie die Möglichkeit, einzelne Ergebnismengen als Eingabe für die Verarbeitung zu nutzen.

Abschnitt 5.1 beschreibt genauer, welche Funktionen LAKE erfüllen soll. Durch die Nutzung der in Kapitel 3 angegebenen Anforderungen und Verarbeitungsmodi wird die gewünschte Funktionsweise der Architektur aufgezeigt. In Abschnitt 5.2 wird anschließend ein Konzept erarbeitet, welches die benötigten Funktionen beinhaltet.

5.1 Geforderte Funktionen

Wie bereits in Kapitel 3 beschrieben wurde, ist die Erfüllung der gestellten Anforderungen direkt an die Verfügbarkeit verschiedener Verarbeitungsmodi gebunden. Werden alle Modi aus Abbildung 3.1 unterstützt, können auch die Anforderungen vollständig erfüllt werden.

Da es sich bei LAKE um eine Kombination aus der Lambda- und der Kappa–Architektur handelt, werden einige Verarbeitungsmodi durch diese Vereinigung unterstützt. Die Möglichkeit zur erneuten Verarbeitung zuvor erstellter Ergebnismengen und die Verwendung externer Datenquellen jedoch kann so noch nicht abgedeckt werden. Abbildung 5.1 stellt dar, welche Modi bereits unterstützt werden und welche es noch umzusetzen gilt.

Um die Lambda- und die Kappa–Architektur zu kombinieren, müssen sich die Verarbeitungskomponenten beider Architekturen in der neuen Architektur wiederfinden. Das bedeutet, dass LAKE sowohl eine Möglichkeit zur Datenstrom- als auch zur Stapelverarbeitung benötigt. Zudem muss es möglich sein, den Datenspeicher als Eingabe für die Datenstromverarbeitung zu nutzen, um so historische und Stromdaten zu kombinieren.

Sind diese Funktionen gegeben, fehlt noch die Wiederverwendung berechneter Ergebnismengen sowie die Einbindung externer Datenquellen. Als wiederverwendbare Ergebnismengen können die Batch Views der Lambda–Architektur genutzt werden, die im Folgenden als Ausschnitte beschrieben werden. Sie umfassen kleinere und spezifischere Datenmengen. Ihre erneute Verarbeitung nimmt also weniger Zeit in Anspruch als die der Gesamtmenge. Zudem können sie als eigene Datenquellen aufgefasst werden, welche als Eingabe für die Datenstromverarbeitung genutzt werden können. Da die Ausschnitte mit beliebiger Logik erstellt werden können, kann ihr Inhalt auf den jeweiligen Anwendungsfall angepasst werden. Um die Ausschnitte erneut zu verarbeiten, muss LAKE über eine Verbindung zwischen Datenstrom- und Stapelverarbeitung verfügen.

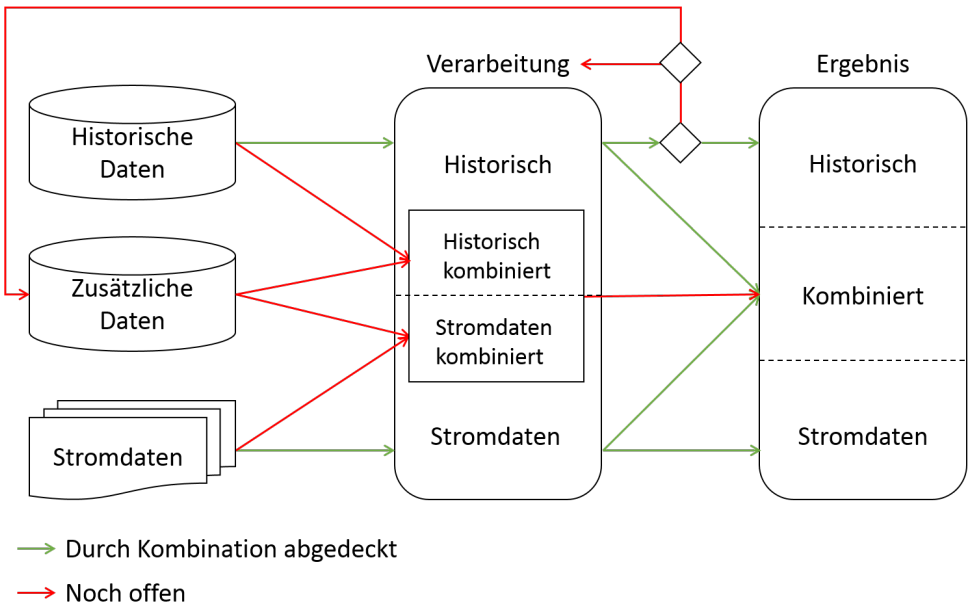

Abbildung 5.1: Die unterstützten Verarbeitungsmodi in einer reinen Kombination aus Lambda- und Kappa–Architektur

Die externen Datenquellen dagegen müssen vor der Verarbeitung eingebunden werden. Nutzt man ein Datenbankmanagementsystem, um die historischen Daten zu speichern, können die externen Daten per Joins hinzugefügt werden.

Zusammengefasst muss LAKE folgende Funktionen bieten:

Reine Verarbeitung historischer Daten – Diese Funktion erhält LAKE durch die Nutzung der Verarbeitungsarten der Lambda–Architektur. Da in der Lambda–Architektur historische und Stromdaten getrennt verarbeitet werden können, kann dies auch in LAKE vorgenommen werden.

Reine Verarbeitung von Stromdaten – Auch diese Funktion wird durch die Lambda–Architektur bereits gegeben. Anstatt historische Daten per Stapelverarbeitung zu Ausschnitten zusammenzufassen, werden hier nur Stromdaten verarbeitet.

Verarbeitung nach Lambda – Da LAKE alle Verarbeitungskomponenten der Lambda–Architektur umfasst, können Daten wie in der Lambda–Architektur selbst verarbeitet werden. Dazu werden die historischen und Stromdaten getrennt voneinander verarbeitet und anschließend zu einer Ergebnismenge zusammengeführt.

Verarbeitung nach Kappa – Die Kappa–Architektur zeichnet sich dadurch aus, dass sie sowohl historische als auch Stromdaten als einen Datenstrom auffasst. Um dies umzusetzen, kann der persistente Datenspeicher als Eingabe für die Datenstromverarbeitung genutzt werden.

Verarbeitung eines Ausschnitts nach Kappa – Anstatt die gesamte Datenmenge als Eingabe für die Datenstromverarbeitung zu nutzen, kann auch ein Ausschnitt als Datenquelle für die Kappa–Architektur verwendet werden. Hierbei endet die Verarbeitung allerdings, sobald alle Daten des Ausschnitts verarbeitet wurden.

Kombination von Stromdaten mit bereits verarbeiteten Daten – Bereits verarbeitete historische Daten können in Form eines Ausschnitts in der Datenstromverarbeitung wiederverwendet werden, um Stromdaten mit ihnen zu kombinieren. Dies kann dazu beitragen, Zusammenhänge zu erkennen, ohne dafür die Gesamtmenge der Daten nutzen zu müssen.

Kombination mit externen Daten – Sowohl historische als auch Stromdaten können mit Daten aus einer externen Datenquelle kombiniert werden. So können weitere Daten als Informationsquellen in der Verarbeitung verwendet werden.

5.2 Aufbau

Dieser Abschnitt beschreibt, wie LAKE aufgebaut ist, um die in Abschnitt 5.1 beschriebenen Funktionen zu unterstützen. Dafür wird zunächst die Gesamtarchitektur kurz vorgestellt, bevor näher auf die einzelnen Komponenten eingegangen wird.

Um Daten wie in der Lambda–Architektur verarbeiten zu können, braucht es zwei getrennte Komponenten. Eine der beiden Komponenten befasst sich mit der Datenstromverarbeitung aus beliebigen Datenquellen, während die andere die Verarbeitung der historischen Daten übernimmt. Damit die historischen Daten wie in der Kappa–Architektur auch als Datenstrom verarbeitet werde können, braucht es zusätzlich eine Verbindung zwischen den beiden Komponenten. Bei ihrer Ankunft werden Stromdaten kopiert und an beide Komponenten weitergeleitet. Abbildung 5.2 zeigt den allgemeinen Aufbau von LAKE, in dem die grundsätzlichen Verarbeitungskomponenten dargestellt sind.

Diese allgemeine Darstellung dient als Grundlage für den Aufbau von LAKE. In den folgenden Abschnitten werden die einzelnen Komponenten und ihre Funktionsweise genauer beschrieben, bevor sie zu einem detaillierten Gesamtkonzept zusammengesetzt werden.

5.2.1 Stapelverarbeitungskomponente

Die Stapelverarbeitungskomponente verwaltet und verarbeitet die historischen Daten. Daten, die an diese Komponente weitergegeben werden, müssen zunächst persistent gespeichert werden. Hierfür verfügt die Stapelverarbeitungskomponente über einen persistenten Datenspeicher, an die neue Daten angehängt werden. Somit bleiben die bereits geschriebenen Daten unverändert und ihre Integrität bleibt erhalten.

Um allerdings alle Anforderungen aus Kapitel 3 erfüllen zu können, verfügt die Stapelverarbeitungskomponente über die Möglichkeit, bestimmte Informationen aus den ankommenden Daten herauszufiltern, bevor sie gespeichert werden. Dies kann genutzt werden, um die Speicherung sensibler Informationen zu verhindern. So können beispielsweise private Daten von Patienten ausgefiltert werden. Da diese Informationen durch die Filterung allerdings unwiderruflich verloren gehen, sollten solche Veränderungen wohlüberlegt sein.

In regelmäßigen Abständen werden per Stapelverarbeitung aus der Gesamtmenge der Daten Ausschnitte erstellt, ähnlich zu den Batch Views der Lambda–Architektur. Diese Ausschnitte

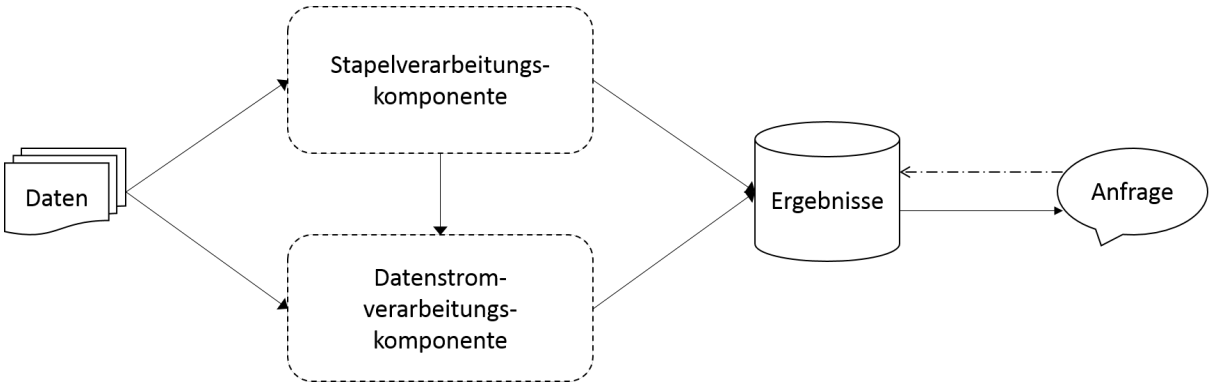

Abbildung 5.2: Der Aufbau von LAKE

repräsentieren beliebige Zwischenergebnisse. Die Logik, die die Ausschnitte bildet, kann jederzeit verändert werden, sodass bei der nächsten Verarbeitung der historischen Daten neue Ausschnitte entstehen. Dies kann über eine neue Version des Codes oder auch durch die Nutzung von Konfigurationsdateien geschehen.

Abbildung 5.3 zeigt, wie die Stapelverarbeitungskomponente aufgebaut ist. Auch ist hier dargestellt, wie die externen Datenquellen in die Verarbeitung miteinbezogen werden. Sie werden vor der Verarbeitung geladen, um direkt in die Ausschnitte einzufließen.

5.2.2 Datenstromverarbeitungskomponente

Im Gegensatz zur Stapelverarbeitungskomponente besteht die Datenstromverarbeitungskomponente lediglich aus der Datenstromverarbeitung. Ankommende Daten aus beliebigen Quellen werden hier als Datenstrom verarbeitet und in die Ergebnismenge eingefügt. Da die Stapelverarbeitungskomponente bereits über einen persistenten Datenspeicher verfügt, müssen Daten in der Datenstromverarbeitungskomponente nicht erneut gespeichert werden.

Die Datenstromverarbeitungskomponente kann zu jedem Zeitpunkt mit Konfigurationen flexibel angepasst werden. So kann beispielsweise festgelegt werden, aus welcher Datenquelle die zu verarbeitenden Daten stammen und wie sie zu verarbeiten sind. Hierfür ist ein Speicherort für Konfigurationsdateien vorhanden, die vor der eigentlichen Verarbeitung geladen werden.

Auch die externen Quellen können hier verwendet werden. Je nach Konfiguration können diese vor der Verarbeitung geladen und mit den Stromdaten verknüpft werden. Abbildung 5.4 zeigt den Aufbau der Datenstromverarbeitungskomponente.

5.2.3 Gesamtkonzept

Betrachtet man die Datenstrom- und Stapelverarbeitungskomponente getrennt, so bieten diese bis auf die Einbindung externer Daten keine weiteren Funktionen, die über die der Lambda–Architektur hinausgehen. Dies liegt daran, dass ein Großteil der Funktionen von

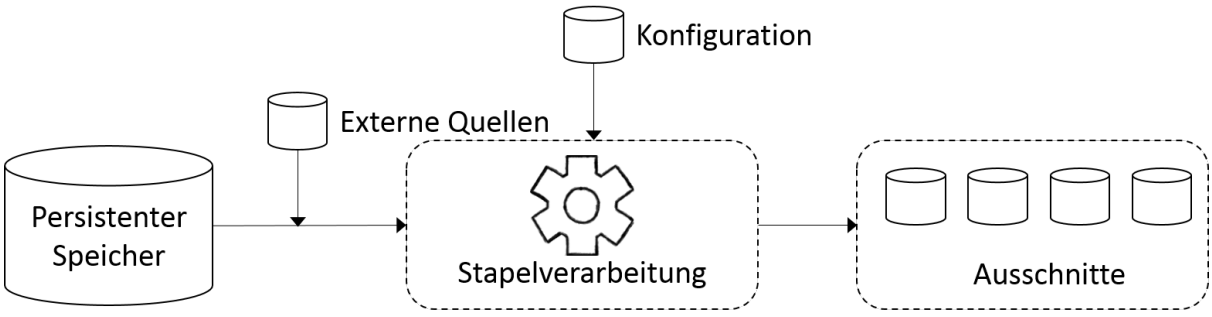

Abbildung 5.3: Die Stapelverarbeitungskomponente von LAKE

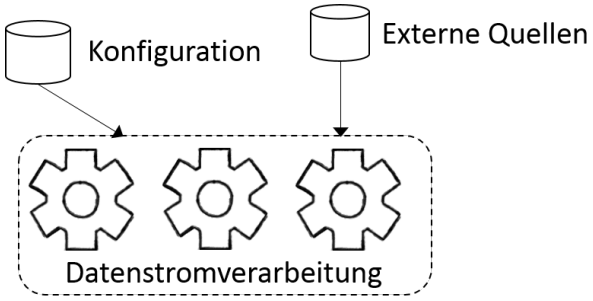

Abbildung 5.4: Die Datenstromverarbeitungskomponente von LAKE

LAKE auf der Verbindung zwischen Datenstrom- und Stapelverarbeitungskomponente beruht. Historische Daten können so nicht nur per Stapelverarbeitung, sondern auch als Datenstrom verarbeitet werden. Ebenso ist es möglich, einen Ausschnitt des Datenspeichers als Datenstrom zu verarbeiten. Durch diese Querverbindungen gewinnt LAKE an zusätzlicher Flexibilität, da sie auf zusätzliche Anwendungsfälle angepasst werden kann. In Abbildung 5.5 ist die gesamte Architektur mit den benötigten Verbindungen zwischen den Komponenten aufgezeigt.

Wichtig ist, dass die in dieser Darstellung gezeigten Datenbewegungen nicht immer in ihrer Gesamtheit verwendet werden. Je nach Anwendungsfall werden bestimmte Verbindungen genutzt oder ignoriert. Um beispielsweise Daten wie in der Lambda–Architektur zu verarbeiten, bleiben alle Verbindungen zwischen Datenstrom- und Stapelverarbeitungskomponente ungenutzt. Auch die Konfiguration sowie die externen Datenquellen werden nicht benötigt. Die Architektur wird wie in Abbildung 5.6 dargestellt verwendet.

Da es sich bei LAKE um eine Kombination aus der Lambda- und der Kappa–Architektur handelt, ist es auch möglich, letztere abzubilden. Hierfür wird die Stapelverarbeitung nicht genutzt und die Daten werden stattdessen von dem Datenspeicher aus in die Datenstromverarbeitungskomponente eingespeist. Abbildung 5.7 zeigt, welche Verbindungen in diesem Fall genutzt werden.

Die Verbindungen zwischen den beiden Komponenten erlauben zudem, verarbeitete historische und externe Daten in beiden Komponenten zu nutzen. LAKE bietet somit alle in Abschnitt 5.1 beschriebenen Funktionen.

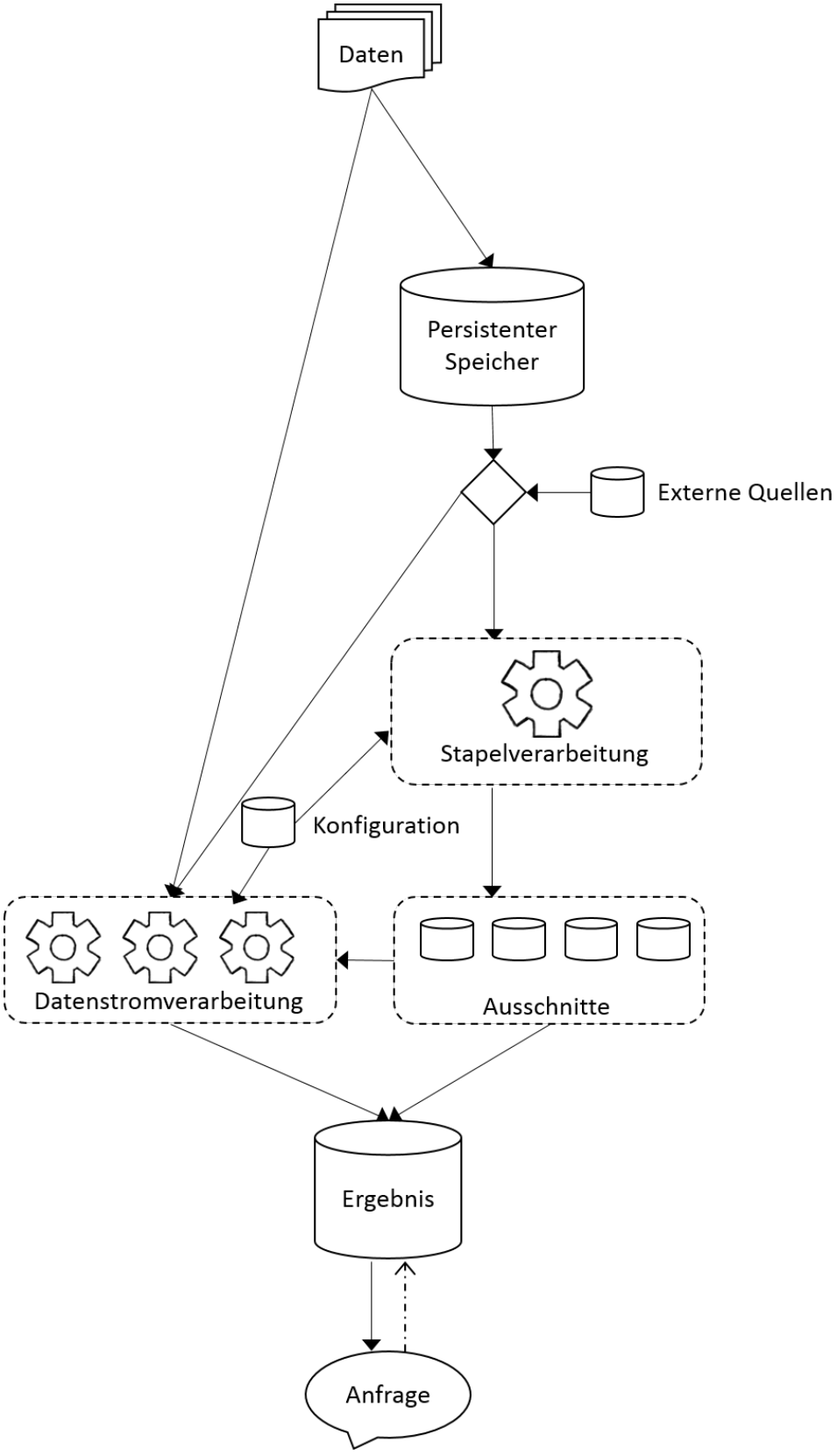

Abbildung 5.5: LAKE im Ganzen

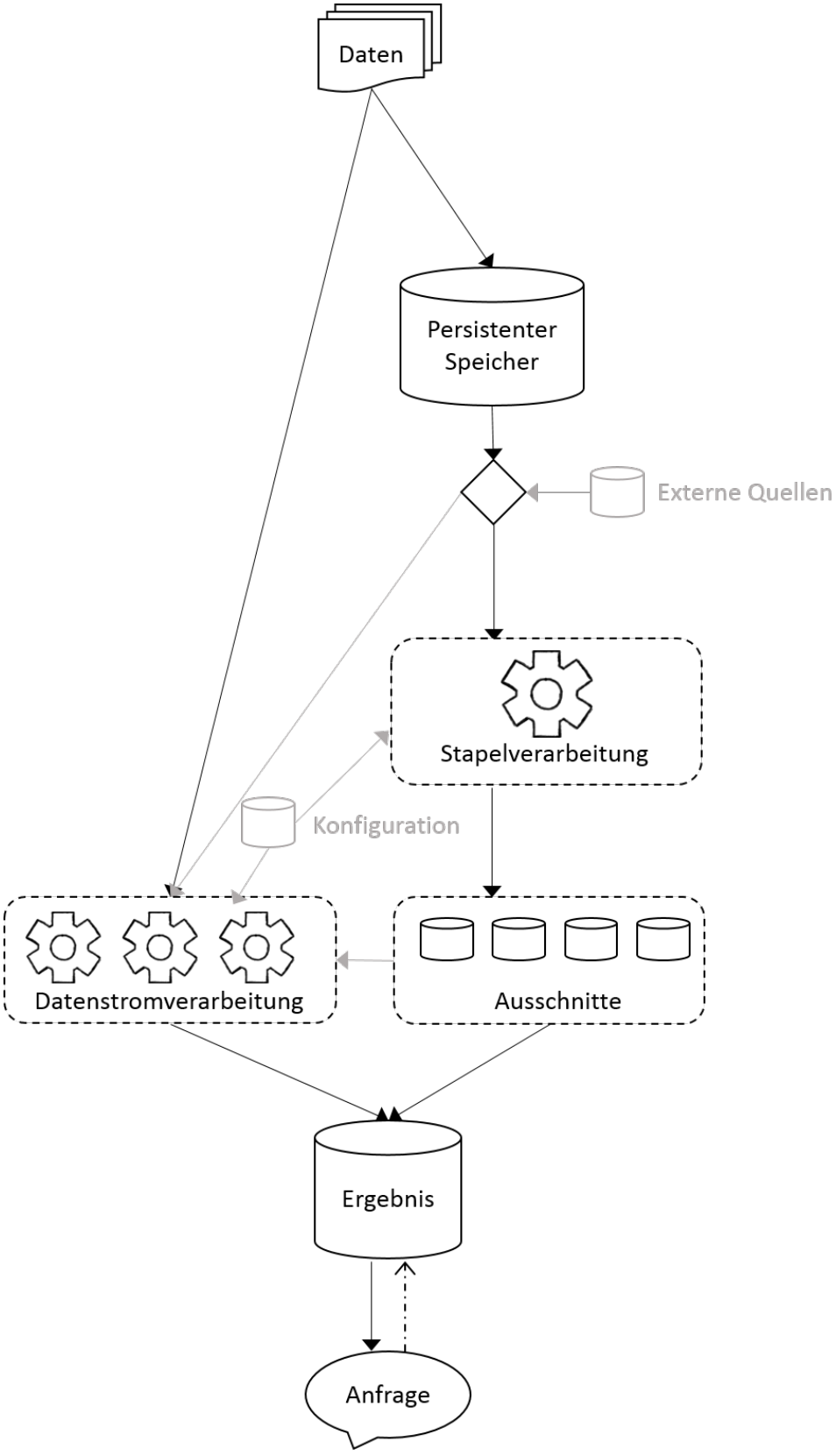

Abbildung 5.6: LAKE verwendet als Lambda–Architektur

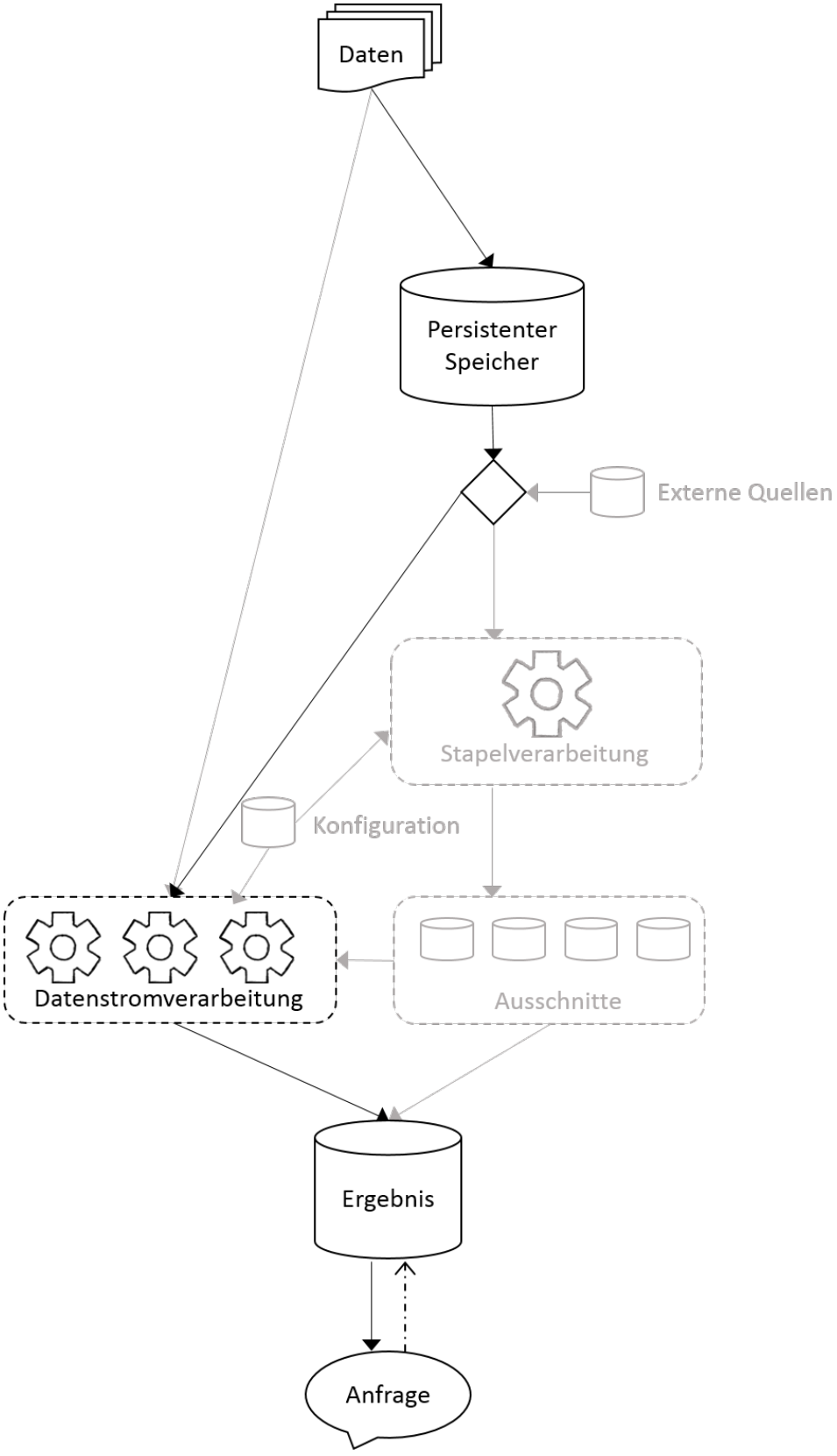

Abbildung 5.7: LAKE verwendet als Kappa–Architektur

6 Allgemeine Umsetzung

Das in Kapitel 5 erarbeitete Konzept kann in beliebigen Szenarien angewandt werden. Da die konkrete Umsetzung von LAKE stark von diesem Szenario abhängt, werden tatsächliche Implementierungen erst in Kapitel 8 anhand zweier Anwendungsbeispiele vorgestellt. Dieses Kapitel dagegen beschreibt die allgemeine Umsetzung des Konzeptes und geht dabei besonders auf die Zusammenarbeit zwischen Datenstrom– und Stapelverarbeitungskomponente ein.

Um eine Abstraktion von einer konkreten Implementierungsart zu erstellen, braucht es definierte Schnittstellen, über die die Komponenten angesteuert werden können. Dadurch kann LAKE flexibel auf den Anwendungsfall angepasst werden. So können die einzelnen Verarbeitungsmethoden in beliebigen Programmiersprachen und Systemen umgesetzt werden. Eine Auswahl möglicher Systeme für die Implementierung wird in Kapitel 7 vorgestellt.

Abbildung 6.1 stellt die Schnittstellen der beiden Hauptkomponenten von LAKE dar. Dabei handelt es sich um die Datenstromverarbeitungskomponente (StreamingExecutor) und die Stapelverarbeitungskomponente (BatchExecutor).

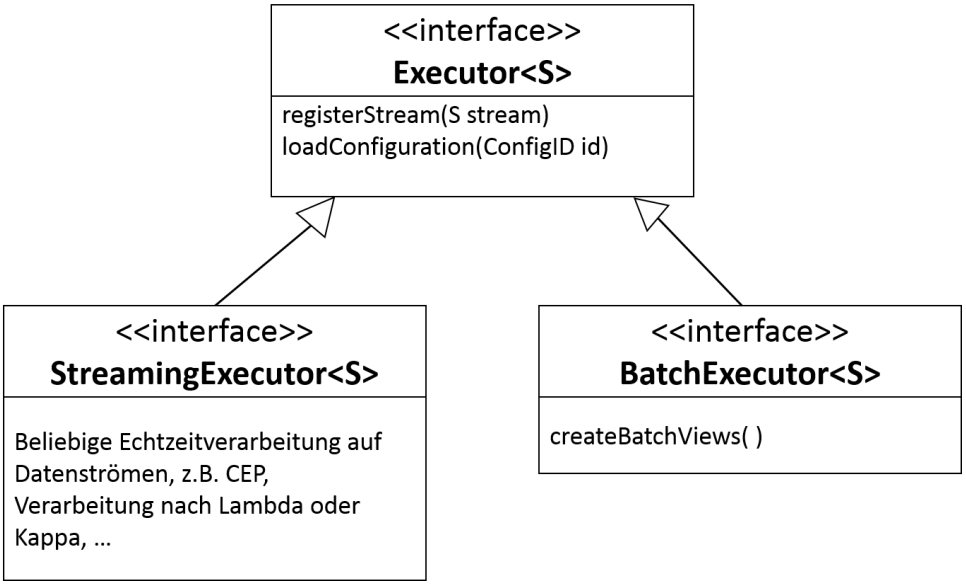

Abbildung 6.1: Die für LAKE verwendeten Schnittstellen

6.1 Abläufe innerhalb der Komponenten

Beide Verarbeitungskomponenten bieten die Möglichkeit, einen Datenstrom anzunehmen. In der Datenstromverarbeitungskomponente kann dieser direkt weiterverarbeitet werden, während die Stapelverarbeitungskomponente die so erhaltenen Daten in den persistenten Speicher schreiben kann. Hierfür dient die Methode `registerStream(S stream)`. Da jedes System einen eigenen Datentyp für Datenströme definiert, wurde hierbei auf generische Typen zurückgegriffen.

Zudem verfügen beide Komponenten über die Möglichkeit, eine Konfiguration zu laden, um damit die Verarbeitung näher zu bestimmen. Während die Datenstromverarbeitungskomponente damit jederzeit flexibel neu konfiguriert werden kann, ist dies in der Stapelverarbeitungskomponente nur zwischen der Erstellung der Ausschnitte möglich. Für das Laden der Konfiguration dient die Methode `public void loadConfiguration(ConfigIDs configID)`.

Während in der Stapelverarbeitung der Datenstrom nur für die Speicherung benötigt wird, wird in der Datenstromverarbeitungskomponente in der Regel direkt auf diesem gearbeitet. Hierfür wird der Datenstrom innerhalb der Komponente an passende Methoden weitergereicht, in denen die durch die Konfiguration festgelegte Verarbeitung stattfindet.

Abbildung 6.2 zeigt ein Sequenzdiagramm für die Datenstromverarbeitung selbst, die innerhalb der Datenstromverarbeitungskomponente ausgeführt wird. Mögliche Konfigurationen werden außerhalb der eigentlichen Verarbeitung geladen und angewandt. Bevor die Datenobjekte im Datenstrom verarbeitet werden, besteht allerdings die Möglichkeit, optionale Daten abzurufen, wie beispielsweise externe Daten oder Ergebnisse der Stapelverarbeitungskomponente. Anschließend wird jedes Datenobjekt entsprechend der Implementierung verarbeitet.

Allerdings besteht in der Datenstromverarbeitungskomponente auch die Möglichkeit, Daten aus dem Datenspeicher einzulesen und diese zu verarbeiten. Dies ist beispielsweise für die Verarbeitung analog zur Kappa–Architektur nötig. In diesem Fall wird die Methode `registerStream(S stream)` nicht benötigt. Stattdessen werden die Daten als Datenstrom aus dem Speicher eingelesen. Das passende Sequenzdiagramm ist in Abbildung 6.3 dargestellt.

Die Stapelverarbeitungskomponente benötigt den Datenstrom für die eigentliche Verarbeitung nicht. Stattdessen kann mit der Methode `createBatchViews( )` die Erzeugung der Ausschnitte angestoßen werden. Wichtig ist dabei, dass diese Erzeugung in regelmäßigen Zeitabständen wiederholt wird, um die Ausschnitte aktuell zu halten.

Ein Sequenzdiagramm für die Stapelverarbeitung selbst ist in Abbildung 6.2 aufgezeigt. Da die Speicherung der Datenstromobjekte unabhängig von der Erstellung der Ausschnitte geschieht, ist sie nicht dargestellt. Auch hier können vor der Verarbeitung selbst optional weitere Daten geladen werden, um diese miteinzubeziehen. Hierbei handelt es sich beispielsweise um externe Daten oder auch die Ergebnisse früherer Verarbeitungen. Dies ermöglicht zudem eine Verarbeitung in mehreren Schritten.

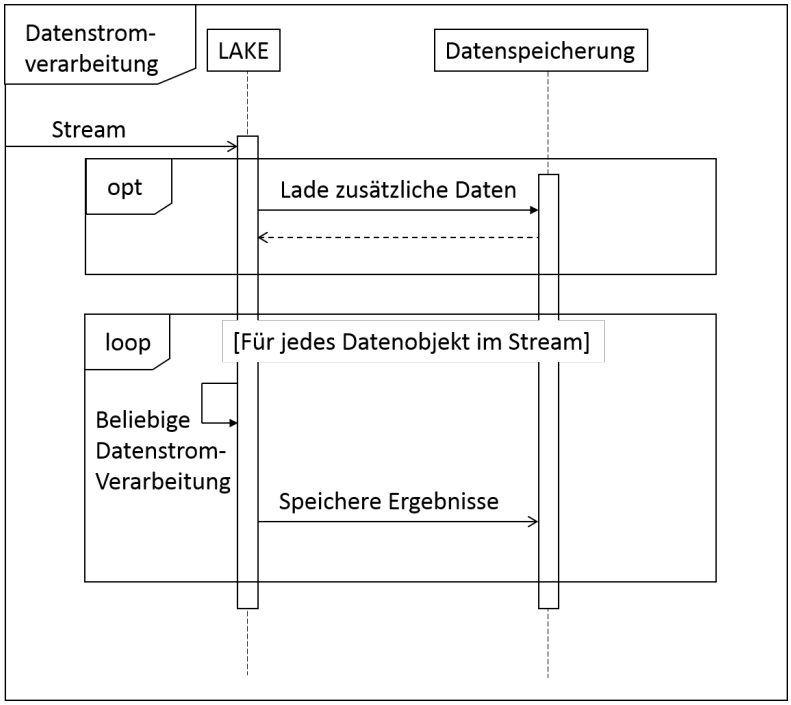

Abbildung 6.2: Sequenzdiagramm für die Datenstromverarbeitung

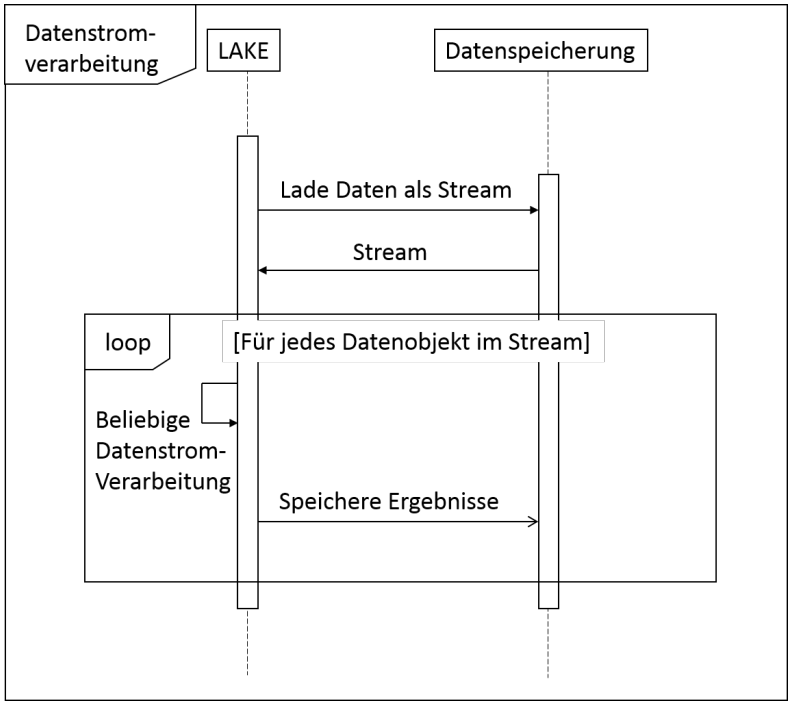

Abbildung 6.3: Sequenzdiagramm für die Datenstromverarbeitung ohne eintreffenden Datenstrom

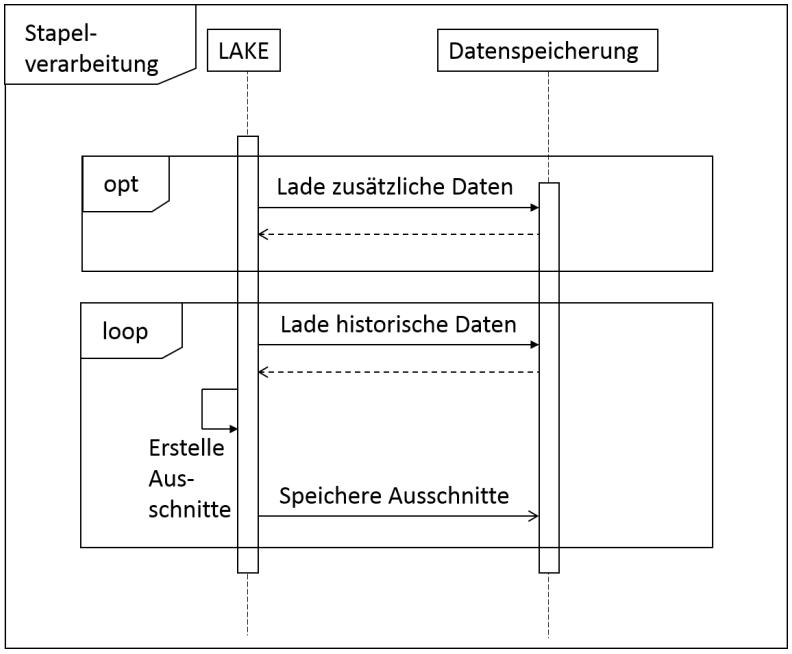

Abbildung 6.4: Sequenzdiagramm für die Stapelverarbeitung

6.2 Interaktion zwischen den Komponenten

Zwar können die Komponenten getrennt voneinander agieren, doch für die Realisierung bestimmter Verarbeitungsmodi sind sie nicht unabhängig voneinander. Dieser Abschnitt beschreibt für verschiedene Verarbeitungsmodi, welche Abhängigkeiten zwischen den Komponenten bestehen. Verarbeitungsmodi, die auf reiner Datenstrom– bzw. Stapelverarbeitung basieren, werden hierbei nicht mit einbezogen.

Den Anfang bildet die Verarbeitung analog zur Lambda–Architektur. Hierfür können die Datenstrom– und die Stapelverarbeitungskomponente unabhängig voneinander agieren. Abbildung 6.5 zeigt, welche Schritte für die Verarbeitung nötig sind. Hierbei wird sichtbar, dass die Verarbeitungen selbst parallel vorgenommen werden können, da keine Abhängigkeiten bestehen. Erst nach der Verarbeitung werden die Ergebnisse zusammengeführt.

Abbildung 6.6 dagegen zeigt die Verarbeitungsschritte für eine Verarbeitung analog zur Kappa–Architektur. Hierbei wird die Stapelverarbeitungskomponente nur für die Speicherung der Daten im persistenten Speicher benötigt. Anschließend werden die Daten als Datenstrom verarbeitet. Hierbei besteht eine Abhängigkeit zwischen der Datenspeicherung und der weiteren Verarbeitung.

Weitere Abhängigkeiten werden sichtbar, wenn aktuelle Daten aus dem Datenstrom mit unverarbeiteten externen oder historischen Daten kombiniert werden sollen. Abbildung 6.7 zeigt, in welcher Reihenfolge die einzelnen Schritte ausgeführt werden, um eine solche Kombination zu erreichen. Die Speicherung des Datenobjektes kann dabei parallel zu allen anderen Schritten

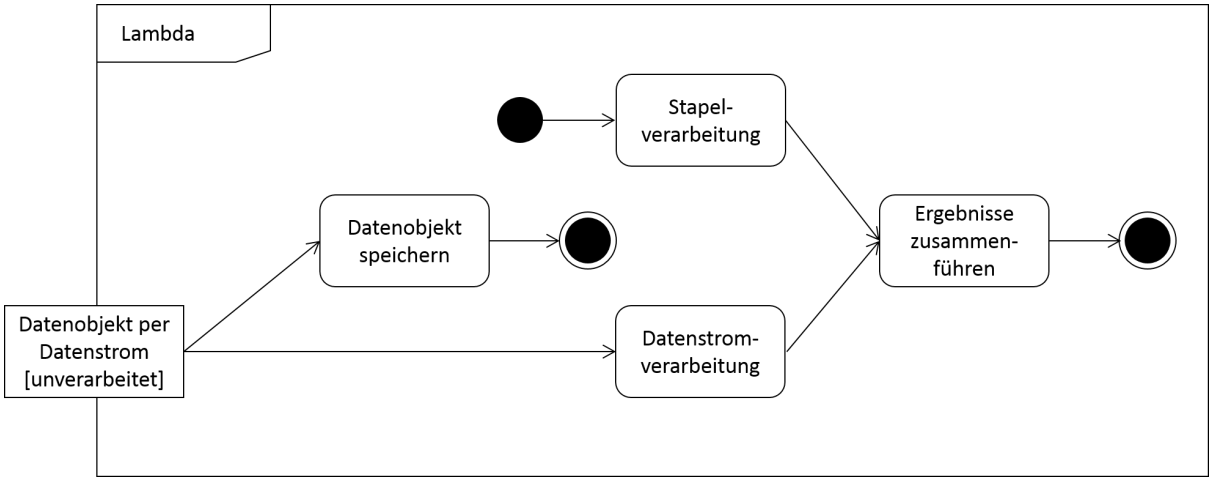

Abbildung 6.5: Aktivitätsdiagramm für die Verarbeitung analog zur Lambda–Architektur

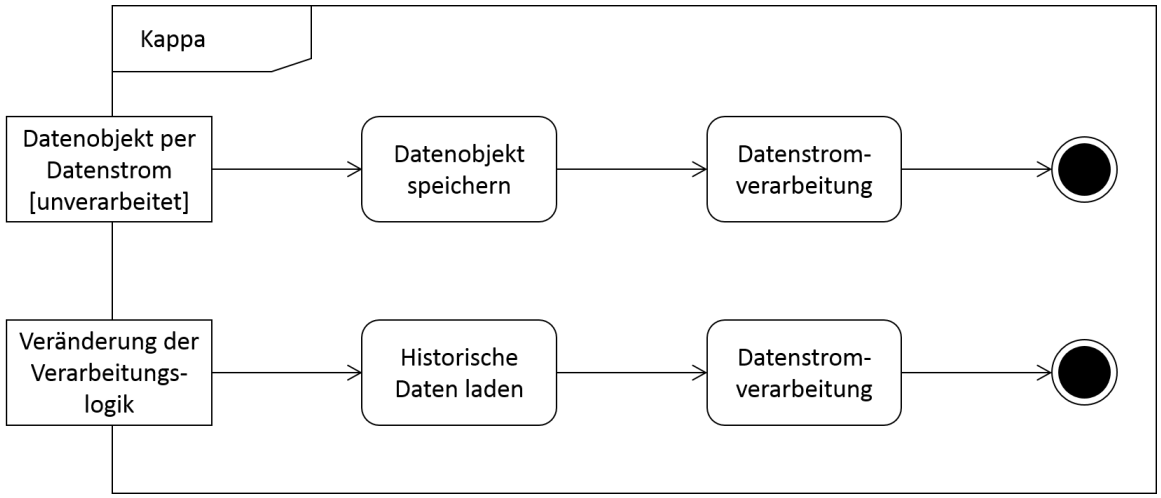

Abbildung 6.6: Aktivitätsdiagramm für die Verarbeitung analog zur Kappa–Architektur

erfolgen. Allerdings müssen vor der Datenstromverarbeitung die zusätzlichen Daten aus dem persistenten Speicher geladen werden. Setzt man vor diesen Schritt die Stapelverarbeitung, können so auch Ausschnitte mit in die Datenstromverarbeitung einbezogen werden.

Zuletzt besteht die Möglichkeit, auch einzelne Ausschnitte als Datenstrom zu verarbeiten. Den Ablauf hierfür stellt Abbildung 6.8 dar. So können Daten, die bereits per Stapelverarbeitung vorverarbeitet wurden, mit geringer Latenzzeit weiterverarbeitet werden. Dafür müssen zunächst die Ausschnitte erstellt werden. Diese können dann, ähnlich zur Kappa–Architektur, als Datenquelle für die Datenstromverarbeitung genutzt werden.

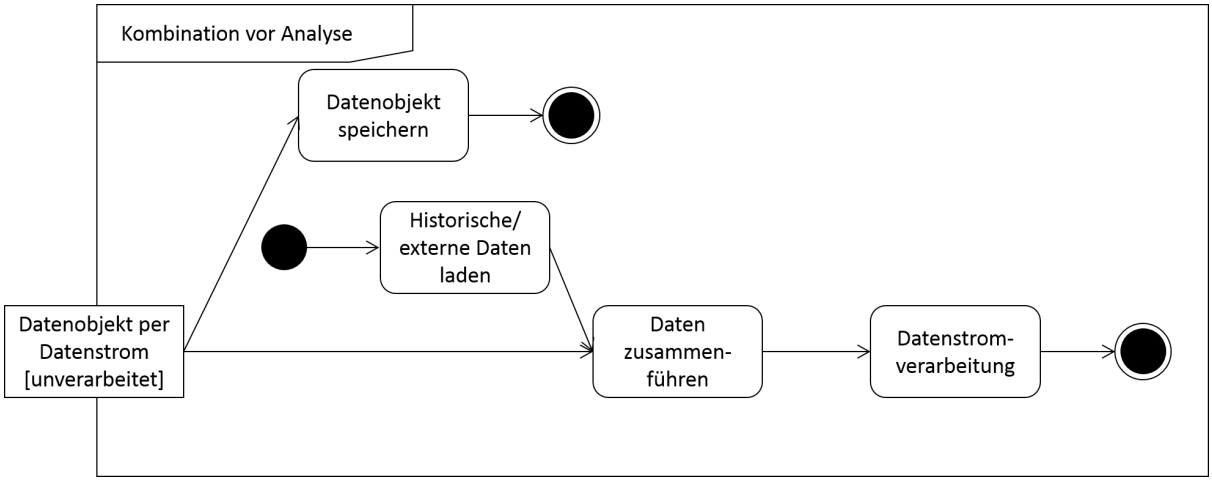

Abbildung 6.7: Aktivitätsdiagramm für die Kombination der Daten vor der Verarbeitung

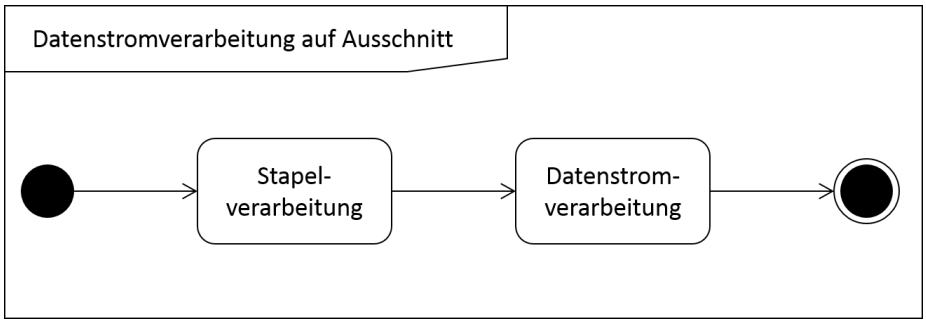

Abbildung 6.8: Aktivitätsdiagramm für erneute Verarbeitung eines Ausschnitts per Datenstromverarbeitung

Diese verallgemeinerten Darstellungen der Funktionsabläufe innerhalb von LAKE können nun anhand eines Beispiels praktisch umgesetzt werden. Diese Umsetzung ist in Kapitel 8 beschrieben. Vorher müssen allerdings passende Systeme ausgewählt werden, mit denen die Umsetzung erfolgen kann. Hierfür bietet Kapitel 7 eine Übersicht über verschiedene Techniken und Systeme zur Datenverarbeitung.

7 Techniken und Systeme

Durch moderne Technologien, wie beispielsweise das IoT, kommt es zu immer größeren Datenmengen, die analysiert werden müssen [WGFR16]. Dabei wird auch im Bereich Big Data noch auf Stapelverarbeitungsarten – wie z.B. MapReduce [DG04] – zurückgegriffen, obwohl bereits 2014 70% der Teilnehmer einer Umfrage die Notwendigkeit für Echtzeitverarbeitung feststellte [LIX17]. Darum entstehen immer mehr Systeme deren Ziel es ist, Daten so schnell wie möglich zu verarbeiten [WGFR16].

Dieses Kapitel beschäftigt sich sowohl mit verschiedenen Techniken zur Datenverarbeitung (Abschnitt 7.1), als auch mit konkreten Systemen, mit denen diese Techniken umgesetzt werden können (Abschnitt 7.2). Da es sich bei LAKE um eine flexible Architektur handelt, die von einer konkreten Technologie abstrahiert, kann jede der vorgestellten Technologien für eine Umsetzung gewählt werden. Um allerdings einen Prototypen umzusetzen, werden in Abschnitt 7.3 einige Technologien ausgewählt.

Der Fokus dieses Kapitels liegt dabei nicht auf Vollständigkeit, da es eine große Vielzahl an Techniken und Systemen gibt. Vielmehr soll es einen Überblick geben und einzelne Aspekte und Instanzen näher beschreiben.

7.1 Techniken

Bei Techniken und Methoden handelt es sich nicht um konkrete Systeme, sondern vielmehr um Konzepte, die zur Datenverarbeitung herangezogen werden können. Da einige der in Abschnitt 7.2 vorgestellten Systeme auf diese Techniken zurückgreifen, werden diese im Folgenden näher vorgestellt. Zudem können sie auch für die Implementierung von LAKE verwendet werden.

Continuous Queries Bei *Continuous Queries* handelt es sich um Anfragen, die auf Datenströmen ausgeführt werden [BBC+09]. Dabei werden die Anfragen statt wie bei einem Datenbankmanagementsystem (DBMS) nicht vom Benutzer, sondern durch ein Datenstrommanagementsystem (DSMS) automatisch ausgeführt [BBC+09; TGNO92]. Die Anfrage wird auf jedem Datenobjekt wiederholt, das per Datenstrom an das System gesendet wird.

Da bei Datenströmen nicht immer gewährleistet werden kann, dass die Daten in der korrekten Reihenfolge im System ankommen [BBC+09], werden eventuelle Zusammenhänge bei der

Analyse nicht berücksichtigt. So kann jedes Datenobjekt getrennt von allen anderen verarbeitet werden, sobald es im System registriert wurde [TGNO92].

Bei Datenströmen handelt es sich allerdings um unendliche Datenquellen, weshalb für die Verwendung von Continuous Queries zudem Zeitfenster definiert werden [BBC+09]. Diese Zeitfenster beschreiben, welche Datenobjekte in die Verarbeitung mit einbezogen werden sollen. So können beispielsweise nur die Datenobjekte betrachtet werden, die in der nächsten Stunde im System eintreffen.

In ihrer Arbeit nutzen Barbieri et al. eine soziale Medienplattform als beispielhafte Datenquelle [BBC+09]. Ähnlich kann man auch Anfragen aus dem Bereich der Industrie 4.0 oder eHealth verwenden. Mögliche Anfragen wären beispielsweise „Welche Temperaturen wurden in der letzten Stunde in meiner Maschine gemessen?“ oder „Wie hoch war der maximale Puls des Patienten in den letzten 15 Minuten?“ (vgl. [BBC+09]).

Um diese Anfragen jederzeit möglichst schnell beantworten zu können, werden eingehende Datenobjekte direkt entsprechend verarbeitet. Das bedeutet, dass für die erste Anfrage nur die Temperaturwerte zurückgeliefert werden, wohingegen für die zweite Anfrage die Pulswerte des Patienten per `MAX` aggregiert werden (vgl. [BBC+09]). Dabei werden nur die Verarbeitungsergebnisse gespeichert, die in das definierte Zeitfenster passen. Für die erste Anfrage umfasst die Ergebnismenge nur Daten der letzten Stunde, während die zweite Anfrage als Ergebnis nur Datenobjekte zurückliefert, die maximal 15 Minuten alt sind. Diese Ergebnismengen können zu beliebigen Zeitpunkten vom Nutzer abgerufen werden.

Complex Event Processing (CEP) Nicht immer ist es notwendig, ein Datenobjekt in einer Datenbank zu speichern [CM12], in manchen Fällen ist eine solche Speicherung sogar unerwünscht. Beispielsweise bei Alarmierungssystemen ist es unnötig, die gesammelten Daten langfristig vorzuhalten, da sie nur in unmittelbarer zeitlicher Nähe von Bedeutung sind. Gleichzeitig aber ist es besonders für Alarmierungssysteme wichtig, in Echtzeit zu reagieren [CM12; Inf10].

Um dies zu ermöglichen, können Datenströme überwacht werden und auf bestimmte Werte hin analysiert werden [CM12]. Diese Werte bilden Ereignisse in der realen Welt ab [CM12], wie beispielsweise eine Temperaturänderung in einer Maschine. Sobald ein solches Ereignis erkannt wurde, kann das System darauf reagieren. Dieses Vorgehen wird als *Event Stream Processing* bezeichnet [Inf10].

Allerdings gibt es Fälle, in denen ein einzelner Wert nicht ausreicht, um einen passenden Rückschluss zu ziehen [Inf10]. Oft müssen erst mehrere einzelne, abhängige Ereignisse festgestellt werden, bevor eine entsprechende Alarmierung erfolgen kann. Ist dies der Fall, spricht man von *Complex Event Processing* [CM12; Inf10].

Um ein solches komplexes Ereignis korrekt klassifizieren zu können, müssen auf einem Ereignis–Datenstrom mehrere zusammenhängende Ereignisse festgestellt werden [Inf10]. Die Ereignisse werden dabei von verschiedenen Quellen registriert und an eine zentrale Stelle

übermittelt. In einer *Complex Event Processing Engine* [CM12] werden anschließend die Ereignisse verarbeitet. Sollte dabei ein komplexes Ereignis gefunden werden, kann entsprechend reagiert werden, beispielsweise durch eine Alarmierung der zuständigen Person.

Auch hier können in der Industrie 4.0 oder der eHealth passende Beispiele gefunden werden, ähnlich zu den vorgestellten Beispielen aus einem Whitepaper der Informatica LLC [Inf10]. Überschreitet beispielsweise die Temperatur einer Maschine über längere Zeit hinweg einen festgelegten Grenzwert, so treffen im System mehrere Datenobjekte mit überhöhten Temperaturwerten ein. Durch das mehrmalige Vorkommen einer Grenzwertüberschreitung kann davon ausgegangen werden, dass die Maschine insgesamt überhitzt ist. So kann eine entsprechende Warnung erfolgen.

Die Nutzung mehrerer Datenquellen ist ebenfalls möglich. So können zum Beispiel in der eHealth bei einem Patienten Bewegungen und Puls über verschiedene Sensoren erfasst werden. Diese Daten werden an die Complex Event Processing Engine weitergeleitet, wo sie kombiniert zur Erkennung eines epileptischen Anfalls genutzt werden können.

MapReduce Bei MapReduce handelt es sich um eine Mechanik für die Stapelverarbeitung [CY15]. Die Daten werden dabei in zwei getrennten Schritten verarbeitet: *Map* (dt. Abbilden) und *Reduce* (dt. Reduzieren) [DG04]. Ein- und Ausgabedaten sind Schlüssel–Werte–Paare.

Im ersten Schritt der Verarbeitung wird eine Map–Funktion auf die einzelnen Paare angewandt [DG04]. Diese wird vom Nutzer erstellt und führt die Daten zu Zwischenergebnissen zusammen, die wiederum Schlüssel–Werte–Paare sind. Anschließend werden unter einem vorübergehenden Schlüssel die passenden Werte gruppiert und an die Reduce–Funktion weitergeleitet.

Auch diese ist vom Nutzer geschrieben [DG04]. Sie erhält zu jedem Schlüssel eine Menge von Werten, die sie weiter verarbeitet. Dabei entstehen kleinere Ergebnismengen, die meist ein oder kein Element enthalten.

Die Map–Funktion wird parallel auf verschiedenen Teilen der Gesamtdatenmenge ausgeführt [DG04]. Aus jedem dieser Teile werden die Zwischenergebnisse berechnet, die anschließend durch die Reduce–Funktion zusammengeführt werden können.

Dean und Ghemawat zeigen in ihrer Arbeit mehrere Beispiele auf, in denen MapReduce angewandt werden kann [DG04]. Analog zu diesen können auch Beispiele in der Industrie 4.0 und der eHealth gefunden werden. So kann beispielsweise in der Industrie 4.0 die Anzahl der bisher gefertigten Teile ermittelt werden. Dazu werden in der Map–Funktion die Art der Teile als Schlüssel und ihre Anzahl als Werte verwendet. In der Reduce–Funktion werden anschließend aus den verschiedenen Teilergebnissen die Gesamtzahlen pro Teileart extrahiert.

Ähnlich dazu können auch durchschnittliche Pulswerte für Patienten errechnet werden. Hierzu werden die Pulswerte pro Patient ermittelt, und anschließend in der Reduce–Funktion der Durchschnitt errechnet.

7.2 Systeme

In diesem Abschnitt werden Systeme vorgestellt, die sich zum Teil die Techniken aus 7.1 zunutze machen. Bei den Systemen handelt es sich um konkrete Implementierungen, die zur Datenverarbeitung verwendet werden können. In Abschnitt 7.2.1 sind die Systeme nach ihrem Veröffentlichungsjahr aufgelistet. Abschließend folgt in Abschnitt 7.2.2 ein Fazit zu den vorgestellten Systemen.

7.2.1 Systemimplementierungen

Aurora Im Jahr 2003 wird das System Aurora vorgestellt [ACÇ+03]. Es handelt sich dabei um ein Datenbankmanagementsystem, welches besonders für Überwachungssysteme entwickelt wurde. Dafür nutzt Aurora Datenströme, die kontinuierlich verarbeitet werden.

Als Überwachungssysteme werden dabei solche Systeme bezeichnet, die bestimmte Objekte betrachten und dem System dessen Zustand übermitteln [ACÇ+03]. Dies kann in regelmäßigen Abständen oder aufgrund eines bestimmten Ereignisses geschehen. Die Besonderheit liegt dabei darin, dass nicht nur der aktuellste Wert, sondern vielmehr eine Historie von Werten für eine korrekte Analyse des aktuellen Zustandes benötigt wird. Auch sind die aus Datenströmen erhaltenen Daten nicht immer vollständig [ACÇ+03].

Um diese Daten korrekt und in Echtzeit verarbeiten zu können, werden sie nach ihrer Ankunft in einen zirkelfreien, gerichteten Graphen geleitet [ACÇ+03]. Abbildung 7.1 stellt das System mit einem solchen Graphen dar. Innerhalb des Graphen werden verschiedene Operatoren auf die Datenobjekte angewandt, bis sie schließlich an Anwendungen weitergeleitet werden können. Hierfür werden Anfragen auf den verarbeiteten Daten ausgeführt, bei denen es sich entweder um Continuous Queries oder benutzerdefinierte Anfragen (Ad–hoc Queries) handelt.

Hadoop Hadoop gilt als das erste Verarbeitungssystem für Big Data und wurde 2006 auf den Markt gebracht [CY15]. Es handelt sich um ein Open–Source Stapelverarbeitungssystem, welches für die Datenverarbeitung auf MapReduce zurückgreift [Apa17b; SKRC10]. Durch die Nutzung des Hadoop Distributed File Systems, kurz *HDFS*, können zudem Daten direkt durch Hadoop gespeichert werden [Apa17b; SKRC10]. Dabei werden die Daten partitioniert gespeichert und können mit hohem Durchsatz wieder abgerufen werden [Apa17b].

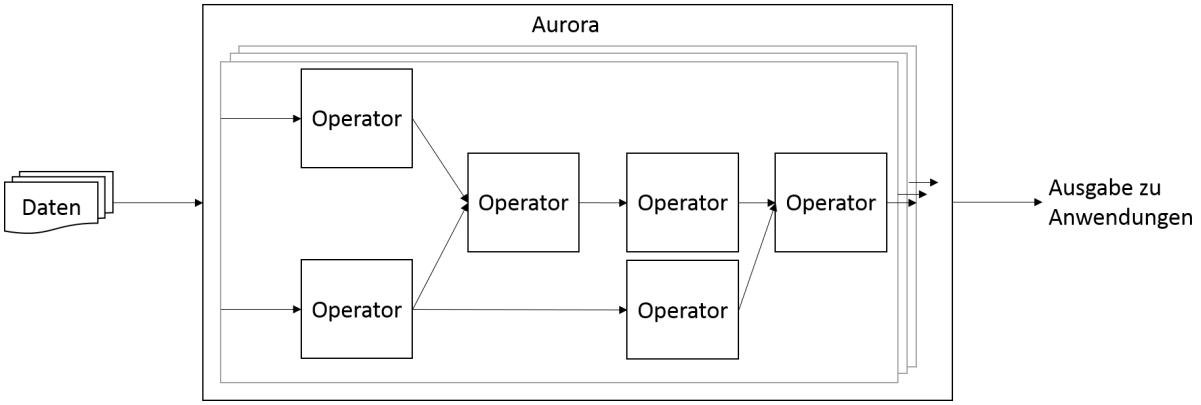

Abbildung 7.1: Die Verarbeitung durch Aurora (nach [ACÇ+03])

Darüber hinaus bietet Hadoop weitere Möglichkeiten, Daten zu speichern und zu verarbeiten [Apa17b; SKRC10]. Mithilfe von HBase können die Daten als Tabellen gespeichert werden, die Erweiterung Hive dagegen bietet Data Warehousing-Optionen. Hadoop ist daher flexibel in verschiedenen Szenarien einsetzbar.

Da es sich bei MapReduce um eine Technik zur Stapelverarbeitung handelt [CY15], ist Hadoop nicht für die Datenstromverarbeitung geeignet. Das HDFS allerdings kann für die Speicherung von Daten aus Datenströmen verwendet werden, da hierfür entsprechende Anpassungen vorgenommen wurden [SKRC10].

Storm Storm entstand 2010 als erstes verteiltes Datenstromverarbeitungssystem [WGFR16]. Es ist mit einer Vielzahl von Programmiersprachen kompatibel und wird als Hadoop für Datenströme bezeichnet [Apa15].

In der Datenstromverarbeitung bietet Storm viele verschiedene Möglichkeiten [Apa15]. So können Daten in Echtzeit analysiert, Machine Learning–Methoden verwendet oder Verarbeitungen immer wieder ausgeführt werden.

Als Eingabe verwendet Storm Datenströme, deren Verarbeitung auf verschiedene Knoten aufgeteilt wird [WGFR16]. Abbildung 7.2 zeigt, wie Daten durch Storm hindurchgeführt werden. Dabei werden die Daten bei Bedarf bei jedem Verarbeitungsschritt neu partitioniert [Apa15]. Die Ausgüsse (engl. Spouts) sind Datenquellen, von denen aus die Daten in Storm hinein geleitet werden, während die Bolzen (engl. Bolts) die Verarbeitung übernehmen.

Spark Ebenfalls 2010 wurde Spark erstmals vorgestellt als Alternative zu MapReduce von Hadoop [ZCF+10]. Im Gegensatz zu Hadoop bietet Spark Machine Learning–Algorithmen sowie interaktive Datenanalysen [ZCF+10]. Dabei bleibt es skalierbar und fehlertolerant.

Implementiert wurde Spark mit Scala, dadurch kann es auch mit Java verwendet werden [Apa17c; ZCF+10]. Zudem kann Spark auch mit Python oder R genutzt werden [Apa17c].

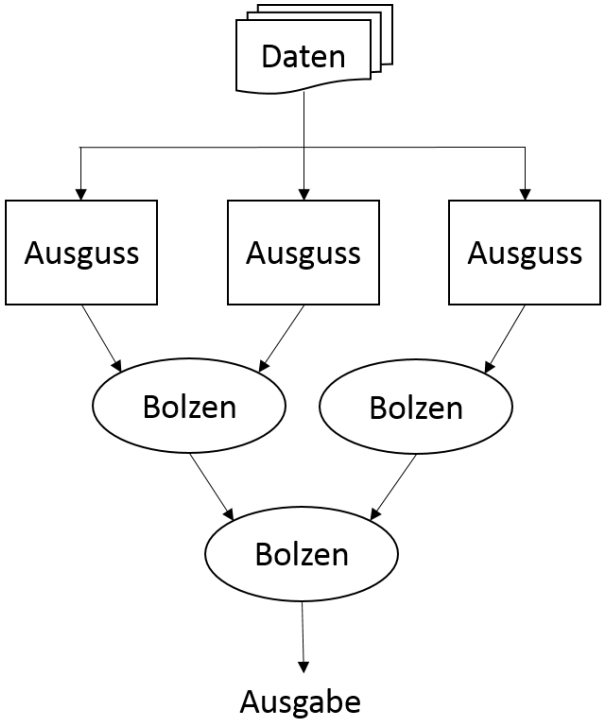

Abbildung 7.2: Darstellung der Datenverarbeitung in Storm (nach [WGFR16])

Die Daten werden für die Verarbeitung mit Spark in so genannte robuste verteilte Datenmengen (engl. Resilient Distributed Dataset, kurz *RDD*) unterteilt [ZCF+10], die auch nach Verlust einer Partition wieder hergestellt werden können. Auf diesen können verschiedene Operationen ausgeführt werden, wie beispielsweise Map-, Transformierungs- und Filterfunktionen [ZCF+10].

Spark selbst ist eigentlich ein System für die Stapelverarbeitung, allerdings bietet es mit der Erweiterung *Spark Streaming* einen Verarbeitungsmodus für Datenströme [WGFR16]. Die Daten werden dabei nicht wie in anderen Systemen tatsächlich als Datenstrom verarbeitet, sondern in Micro–Batches unterteilt [WGFR16].

Spark bietet den Vorteil, dass viele verschiedene Zusatzfunktionen eingebunden werden können [Apa17c]. So unterstützt es beispielsweise SQL, Machine Learning oder auch Graphen [Apa17c]. Zudem bietet Spark Lazy Evaluation für große Anfragen [Apa17c] und eine Integration mit Hadoop [Apa17b]. Spark kann dadurch in Hadoop gespeicherte Daten schnell und auf viele Arten verarbeiten.

Kafka Anders als bei den anderen Systemen handelt es sich bei Kafka weniger um ein System zur Verarbeitung von Daten, sondern vielmehr zur Sammlung und Verteilung von Daten [CY15; KNR11]. Dabei agiert Kafka mithilfe von Nachrichten. Diese Nachrichten werden durch Produzenten (engl. Producer) an so genannte Themen (engl. Topics) weitergegeben.

Die Themen sind auf Servern gespeichert, die Vermittler (engl. Broker) genannt werden. Bei diesen Vermittlern können sich die Verbraucher (engl. Consumer) registrieren, um die Daten zu erhalten. Abbildung 7.3 zeigt, wie Kafka aufgebaut ist.

Mit Kafka können Daten auch aggregiert und überwacht werden [LIX17]. Zudem kann Kafka verteilt und on- und offline arbeiten [CY15; KNR11]

Samza Samza gehört seit 2013 zur Apache Software Foundation [WGFR16]. Wie auch Storm arbeitet Samza direkt auf Datenströmen. Dazu unterteilt das System den Strom in verschiedene Partitionen, die in parallelen Jobs verarbeitet werden können. Somit kann Samza verteilt und skalierbar arbeiten [Apa17e]

Die Daten werden innerhalb von Samza durch Kafka weitergeleitet [WGFR16]. Zudem werden die Samza–Jobs durch den Ressourcenmanager Hadoop YARN [VMD+13] verteilt. Abbildung 7.4 zeigt, wie die Daten durch verschiedene Jobs innerhalb von Samza verarbeitet werden. Dabei ähnelt die Verarbeitung der von Storm, allerdings gibt es nur eine Art von Knoten [WGFR16]. Die Daten werden von einem Job direkt an Kafka weitergegeben, von wo aus der nächste Job die Daten ohne weiteren Aufwand einlesen kann. Dadurch sinkt die Verarbeitungszeit im Vergleich zu Storm.

Flink Flink ist ein System zur Datenstromverarbeitung und existiert seit 2014. Der Grundgedanke der Datenverarbeitung in Flink ist, dass jede Datenmenge als Datenstrom aufgefasst werden kann [Apa17d]. Tatsächliche Datenströme und Datenbanken werden dadurch unterschieden, dass es sich bei Datenbanken um finite Datenströme handelt. So können auch

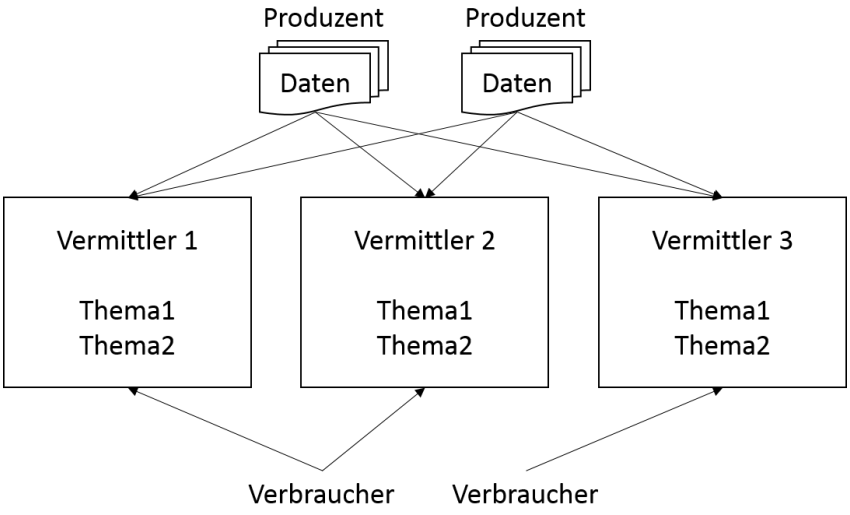

Abbildung 7.3: Die Kafka–Architektur (nach [KNR11])

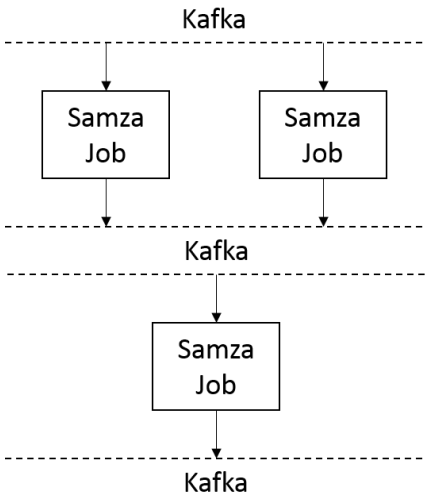

Abbildung 7.4: Darstellung der Datenverarbeitung in Samza (nach [WGFR16])

die Daten in Datenbanksystemen als Datenstrom verarbeitet werden. Dadurch kann es sehr geringe Latenzzeiten garantieren.

Flink kann sowohl mit Java als auch Scala implementiert werden und unterstützt SQL–Anfragen auf Datenstrom- und Stapelebene [Apa17d]. Zudem bietet es durch verschiedene APIs Verarbeitungsmöglichkeiten per CEP, als relationale Tabelle, Graph oder per Machine Learning–Algorithmen.

7.2.2 Übersicht

Während alle der vorgestellten Systeme sich mit der Verarbeitung von Daten befassen, unterscheiden sie sich in der Art der Verarbeitung und den verwendeten Datenquellen. Tabelle 7.1 zeigt die Charakteristika der verschiedenen Systeme. Diese Tabelle dient als Grundlage für die Auswahl passender Systeme zur Umsetzung eines LAKE–Prototypen in Kapitel 6.

System	Systemart	Datenquelle	Verarbeitungsart	Weiteres
Hadoop	Verarbeitung, Speicherung	HDFS	Stapelverarbeitung	–
Aurora	DBMS	Datenstrom	Continuous und Ad–hoc Queries auf Datenströmen	Geschlossenes System
Storm	Datenstromverarbeitung	Datenstrom	Beliebige Datenstromverarbeitung	–
Spark	Stapelverarbeitung	DBMS, HDFS(, Datenstrom)	Stapelverarbeitung, Micro–Batches	Viele Erweiterungen
Kafka	Nachrichtensammler	Producer	Datenstromverarbeitung (Aggregation)	Datensammlung
Samza	Datenstromverarbeitung	Datenstrom	Beliebige Datenstromverarbeitung	Benötigt Kafka und YARN
Flink	Datenverarbeitung allgemein	Datenstrom, DBMS, HDFS	Beliebige Datenstromverarbeitung	Viele Erweiterungen

Tabelle 7.1: Übersicht über die vorgestellten Systeme

7.3 Fazit

Für einen Prototypen von LAKE müssen passende Systeme für dessen Umsetzung ausgewählt werden. Da es sich bei LAKE um ein allgemeines Konzept handelt, kann jedes beliebige System zur Implementierung genutzt werden. Auch können beliebige Techniken für die Datenverarbeitung angewandt werden. Die in Kapitel 8 vorgestellten Implementierungen basieren allerdings auf konkreten Systemen.

Für die Speicherung der Daten ist Hadoop ein geeignetes System. Da viele der Systeme über eine gute Anbindung zum HDFS verfügen, kann Hadoop mit diesen verwendet werden. Zudem handelt es sich beim HDFS um ein Dateisystem, wodurch die Daten in beliebigen Formaten gespeichert werden können. Die Datei- und Ordnerstruktur ermöglicht eine Trennung der verschiedenen Verarbeitungsebenen von LAKE.

Um die Komplexität der Implementierung zu minimieren, sollte sowohl die Datenstrom- als auch die Stapelverwaltung mit dem selben System umzusetzen sein. Auf diese Weise können Analysefunktionen wiederverwendet werden. Die Systeme, die die Verarbeitung selbst umsetzen, müssen also Datenstrom- und Stapelverarbeitung unterstützen und sich zudem flexibel

verwenden lassen. Diese Bedingungen werden lediglich von Spark und Flink erfüllt. Da Spark allerdings für die Datenstromverarbeitung auf Micro–Batches zurückgreift, wird Flink für die Verarbeitung der Datenströme genutzt. Die Implementierung der Stapelverarbeitung dagegen erfolgt sowohl in Spark als auch in Flink.

Innerhalb von LAKE können sowohl CEP als auch Continuous Queries Anwendung finden. Vor allem in der Verarbeitung der Datenströme werden diese Techniken eingesetzt. Da für die Stapelverarbeitung auf Spark und Flink zurückgegriffen wird, ist eine Verwendung von MapReduce allerdings nicht nötig.

8 Anwendungsbeispiele und konkrete Implementierung

Dieses Kapitel beschreibt zwei konkrete Implementierungen von LAKE. Hierfür werden zunächst zwei Anwendungsfälle erarbeitet. Diese geben an, welches Datenformat verwendet und welche Verarbeitungen angewandt werden. Erst danach kann eine Umsetzung erfolgen.

Abschnitt 8.1 stellt ein Szenario aus der Industrie 4.0 vor. Im zugehörigen Prototypen finden sich insgesamt sieben Verarbeitungsmodi, die auf die Daten angewandt werden können. Zwischen ihnen kann jederzeit frei gewechselt werden, zudem können sie über eine Konfigurationsdatei weiter angepasst werden.

Die Grundlage für das Beispiel aus Abschnitt 8.2 dagegen bildet Sensorik aus der eHealth. Um die damit gemessenen Daten klassifizieren zu können, wird hier auf Machine Learning zurückgegriffen. Durch die Verbindung der beiden Verarbeitungskomponenten innerhalb von LAKE können alle Phasen eines Machine Learning–Algorithmus (vgl. [Sha08]) umgesetzt werden.

8.1 Verarbeitung von Maschinendaten

Ähnlich zu dem in Abschnitt 4.2.1 und Abschnitt 4.2.2 verwendeten Beispiel, wird auch hier die Sensorik innerhalb einer Maschine als Datenquelle verwendet. Dabei werden Daten zum Auftrag und zum aktuellen Status in der Maschine erfasst. Das Datenmodell selbst erweitert allerdings das zuvor verwendete Beispiel, um mehr Verarbeitungsmöglichkeiten zu ermöglichen. Die Darstellung des Datenmodells findet sich in Abbildung 8.1. Die Auftragsnummer, die Art der zu fertigenden Teile, das Material und die Anzahl der gesamt zu fertigenden Teile im Auftrag bleiben dabei innerhalb eines Auftrags unverändert. Zur Vereinfachung wird davon ausgegangen, dass pro Auftrag nur eine Art von Teilen hergestellt und nur ein Material verwendet wird. Die Maschine meldet zudem, wie viele Teile sie bereits gefertigt hat und welche Temperatur bei der Fertigung herrscht. Hierfür ist sie mit passender Sensorik ausgestattet, z.B. mit einem Thermometer.

Im Folgenden werden die Verarbeitungsschritte der einzelnen Komponenten näher behandelt.

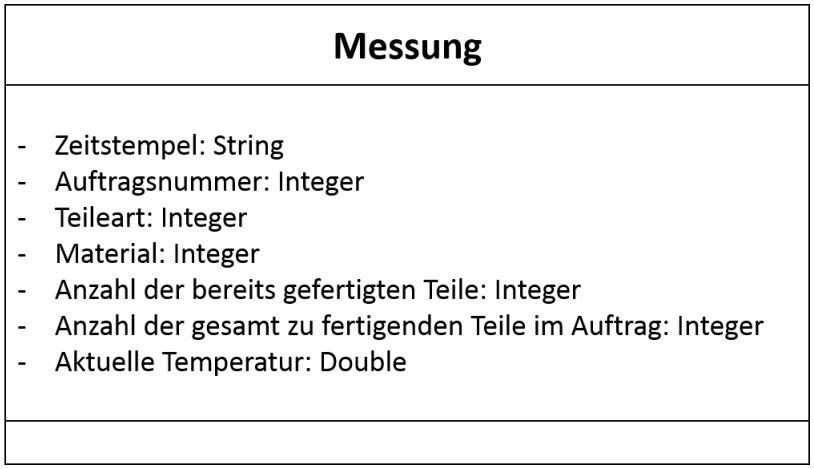

Abbildung 8.1: Das UML–Diagramm für die exemplarischen Messdaten aus der Industrie 4.0

8.1.1 Stapelverarbeitungskomponente

Die Stapelverarbeitungskomponente ist, wie in Abschnitt 7.3 bereits erwähnt, sowohl in Flink als auch in Spark umgesetzt. Zunächst wird jedes eintreffende Datenobjekt im HDFS in einer csv–Datei abgelegt. Sowohl Flink als auch Spark legen hierfür nicht nur eine Datei an, sondern speichern die Daten des Datenstroms in mehreren kleinen Dateien.

Da der historische Datenspeicher bereits mit Daten befüllt wurde, können zu jedem Zeitpunkt Ausschnitte erstellt werden. Während mit Flink die Ergebnisdateien überschrieben werden können, müssen sie in Spark zunächst gelöscht und anschließend neu erzeugt werden. Beide Systeme verfügen aber über die Möglichkeit, csv–Dateien einzulesen und diese direkt auf Objekte zu mappen. Es entstehen so Datenmengen eines bestimmten Objekttyps (`DataSet<T>` in Flink, `Dataset<T>` in Spark).

Da Flink auch in der Stapelverarbeitung auf Datenströme zurückgreift, können in der Stapelverarbeitungskomponente die selben Funktionen verwendet werden, die auch in der Datenstromverarbeitung Einsatz finden. So kann die Komplexität der Implementierung verringert werden, da Code wiederverwendet wird. Damit allerdings Flink und Spark später miteinander verglichen werden können, wurde in beiden Fällen auf SQL–Anfragen zurückgegriffen. Mit Flink müssen die Datenmengen hierfür zunächst in einem `BatchTableEnvironment` registriert werden, während in Spark eine `TempView` erstellt wird. Beides dient dazu, in den SQL–Anfragen auf die Datenmengen verweisen zu können.

Quelltext 8.1 zeigt die hierfür verwendeten SQL–Anfragen. Der erste so erstellte Ausschnitt enthält dabei die Anzahl der gefertigten Teile pro Teileart und Auftrag. Der zweite Ausschnitt dagegen verbindet der historische Datenspeicher mit einer externen Datenquelle und speichert die Messungen, bei denen die maximale Verarbeitungstemperatur des verwendeten Materials überschritten wurde. Dabei können Angaben zur Anzahl der Teile vernachlässigt werden.

Quelltext 8.1 Die zur Erstellung der Ausschnitte verwendeten SQL–Anfragen

```
SELECT part, job, noDone FROM HistDB

SELECT id,job,material,temp,maxTemp FROM HistDB AS h JOIN ExtDB AS e ON h.material =
    e.materialId WHERE h.temp > e.maxTemp
```

Abschließend werden die so entstandenen Ergebnismengen im HDFS abgespeichert. Flink schreibt jedes Objekt dabei als Text, wofür die `toString()`–Methode der Objekte entsprechend angepasst werden muss. Spark dagegen bietet die Möglichkeit, die Ergebnismenge direkt als csv–Dateien in ein Unterverzeichnis zu speichern.

8.1.2 Datenstromverarbeitungskomponente

Die Datenstromverarbeitungskomponente wurde im Rahmen dieser Arbeit mit Flink umgesetzt. Ihre Implementierung umfasst insgesamt sieben verschiedene Verarbeitungsmodi. Diese werden im Folgenden genauer vorgestellt. Zudem kann aus dem HDFS eine Konfigurationsdatei geladen werden. Diese enthält zu jedem Verarbeitungsmodus den Dateinamen eines zu verwendenden Ausschnitts, sowie den Namen der Ergebnisdatei. Wird für die Verarbeitung kein Ausschnitt benötigt, enthält die Konfiguration des Modus an dieser Stelle den Wert null.

CEP In diesem Modus werden die aktuellen Daten per Complex Event Processing (siehe Abschnitt 7.1) auf zu hohe Temperaturen hin analysiert. Hierzu wird ein Grenzwert festgelegt, den die Temperatur des eingehenden Datenobjektes nicht überschreiten darf. Im Falle einer Überschreitung dagegen wird eine passende Warnung generiert.

Flink bietet bereits die Möglichkeit, Muster für die Erkennung solcher Ereignisse zu definieren (siehe [Apa17d]). Diese Muster können die Datenobjekte auf beliebig viele Ereignisse hin untersuchen und so auch Probleme ermitteln, die erst durch das gemeinsame Auftreten mehrerer Ereignisse erkennbar sind.

Der Quelltextausschnitt 8.2 zeigt, wie im Prototypen ein solches Muster erstellt und auf den Datenstrom angewandt wird. In Zeile 2 wird ein Muster definiert, nach welchem die eintreffenden Datenobjekte durchsucht werden können. Die gefundenen Ereignisse, bei denen die Temperatur größer oder gleich dem Grenzwert ist, erhalten den Namen „start“. Auf diese Weise können verschiedene Ereignisse definiert werden. In Zeile 4 wird dieses Muster auf den Datenstrom angewandt. Der Strom `patternStream` enthält nun nur noch die Ereignisse, auf die das definierte Muster zutrifft. Ab Zeile 6 können darum die Datenobjekte aus dem `patternStream` ausgelesen und in passende Warnungen umgewandelt werden.

Für diesen Verarbeitungsmodus ist die Generierung von Ausschnitten unnötig, da hier nur auf den Datenstrom zugegriffen wird.

Quelltext 8.2 CEP in Flink

```
1       // Initialize a pattern for CEP
2       Pattern<GeneralType, ?> tempPattern =
            Pattern.<GeneralType>begin("start").where(data -> data.getTemp() >=
            tempThreshold);
3
4       PatternStream<GeneralType> patternStream = CEP.pattern(stream, tempPattern);
5
6       DataStream<String> warnings = patternStream.select((Map<String, GeneralType>
            pattern) -> {
7             GeneralType gt = (GeneralType) pattern.get("start");
8
9             StringBuilder builder = new StringBuilder();
10            builder.append("[ALERT] - High temperature in job
                  ").append(gt.getJob()).append(", Temperature: ").append(gt.getTemp());
11
12            System.out.println(builder.toString());
13
14            return builder.toString();
15      });
```

Kombination mit historischen Daten In diesem Modus wird der eingehende Datenstrom mit historischen Daten kombiniert, um mögliche Zusammenhänge zu ermitteln. Dabei kann sowohl die Gesamtmenge der historischen Daten als auch ein Ausschnitt verwendet werden. Hierfür muss zunächst in der Stapelverarbeitungskomponente ein Ausschnitt erstellt werden, welcher dann in der Datenstromverarbeitungskomponente geladen werden kann. Es wurde auf den Ausschnitt zurückgegriffen, der zuvor mit den externen Daten verknüpft wurde.

Um die so entstandene Datenmenge mit dem Datenstrom zu kombinieren, wurde im Fall des Prototypen auf die Klasse `RichCoFlatMapFunction<IN1,IN2,OUT>` aus der Flink API [Apa17d] zurückgegriffen. Diese bietet zwei Mapping–Funktionen an, einmal von Typ `IN1` nach Typ `OUT` und einmal von `IN2` nach Typ `OUT`.

Die Implementierung innerhalb des Prototypen ist in Quelltextausschnitt 8.3 dargestellt. Hier wurden diese beiden Funktionen genutzt, um zunächst alle Datenobjekte des Ausschnitts in eine HashMap zu schreiben (siehe Zeile 4ff). Anschließend wurden mit der zweiten Funktion ab Zeile 14 die Temperaturen der Datenobjekte aus dem Datenstrom mittels ihres Materials mit den zuvor gespeicherten Maximaltemperaturen verglichen. Als Ausgabe wurde eine Menge der Datenobjekte zurückgeliefert, deren Temperatur eine zuvor gespeicherte Temperatur überschritt. Dabei wurden Informationen zur Teileanzahl vernachlässigt und stattdessen die Maximaltemperatur zum Objekt hinzugefügt.

Kombination mit einer externen Datenquelle Dieser Verarbeitungsmodus ähnelt der Kombination mit historischen Daten. Allerdings ist hierbei die vorangehende Stapelverarbeitung nicht benötigt. Stattdessen können die Daten der externen Datenquelle direkt gela-

Quelltext 8.3 Die Kombination eines Ausschnittes mit dem Datenstrom in Flink

```
1       Map<Integer, Double> materialTempMap = new HashMap<Integer, Double>();
2
3       @Override
4       public void flatMap1(MaxTempType value, Collector<MaxTempType> out) throws
            Exception {
5             // Add the thing to the Map, but only if it doesn't exist or the new
6             // temperature is lower than the old
7             if (!materialTempMap.containsKey(value.getMaterial())
8             || (materialTempMap.get(value.getMaterial()) > value.getMaxTemp())) {
9                   materialTempMap.put(value.getMaterial(), value.getMaxTemp());
10            }
11      }
12
13      @Override
14      public void flatMap2(GeneralType value, Collector<MaxTempType> out) throws
            Exception {
15            // Map only those values that have a higher temperature than listed in
16            // the map
17            if (materialTempMap.containsKey(value.getMaterial())) {
18                  if (materialTempMap.get(value.getMaterial()) <= value.getTemp()) {
19                        out.collect(new MaxTempType(value.getId(), value.getJob(),
                              value.getMaterial(), value.getTemp(),
20                        materialTempMap.get(value.getMaterial())));
21                  }
22            }
23      }
```

den und mit dem Datenstrom kombiniert werden. Auch hier wird dafür im Prototypen eine `RichCoFlatMapFunction<IN1,IN2,OUT>` verwendet, deren Mapping–Funktionen entsprechend angepasst werden. Durch die hohe Ähnlichkeit zum vorangegangenen Modus, wird auf einen Quelltextausschnitt verzichtet.

Verarbeitung analog zur Lambda–Architektur Da in der Lambda–Architektur Datenstrom- und Stapelverarbeitung stets getrennt sind, ist in diesem Fall eine vorangehende Stapelverarbeitung nicht nötig. Im Prototypen wurde auf den Datenstrom eine `MapFunction<T,O>` aus der Flink API verwendet [Apa17d]. Im Gegensatz zur `RichCoFlatMapFunction<IN1,IN2,OUT>` verfügt diese nur über eine Mapping–Funktion von Typ `T` nach Typ `O`. Diese wird als Continuous Query (siehe Abschnitt 7.1) auf jedes ankommende Datenobjekt angewandt.

Quelltextausschnitt 8.4 zeigt, wie ein Datenobjekt in einen passenden String umgewandelt wird. Dafür werden die Daten werden in das gleiche Format gebracht, wie die Inhalte des ersten Ausschnitts. In Zeile 3 werden dafür nur die Teileart, die Jobnummer und die Anzahl der gefertigten Teile übernommen und als String zurückgegeben.

Quelltext 8.4 Die Mapping–Funktion für die Verarbeitung analog zur Lambda–Architektur in Flink

```
1       @Override
2       public String map(GeneralType value) throws Exception {
3             return value.getPart() + ";" + value.getJob() + ";" + value.getNoDone();
4       }
```

Die so entstandene Menge von Strings kann nun in eine csv–Datei gespeichert werden. Da die Strings die selben Felder enthalten wie der per Stapelverarbeitung berechnete Ausschnitt, können diese nun im Falle einer Anfrage kombiniert werden.

Verarbeitung analog zur Kappa–Architektur Die Kappa–Architektur besitzt keine Stapelverarbeitungskomponente (siehe [Kre14]), wodurch auch keine vorangehende Stapelverarbeitung nötig ist. Stattdessen wird, ähnlich wie bei der Umsetzung der Lambda–Architektur, eine Mapping–Funktion definiert, welche auf den Datenobjekten ausgeführt wird.

Damit im Falle einer erneuten Verarbeitung eine beliebige Mapping–Funktion gewählt werden kann, verfügt die Methode zur Ausführung der Verarbeitung über einen Parameter vom Typ `MapFunction<T,O>` (siehe Quelltextausschnitt 8.5, Zeile 4). So kann flexibel die zu verwendende Verarbeitungsfunktion gewählt werden.

Erneute Verarbeitung aller historischer Daten Die Kappa–Architektur bietet die Möglichkeit, jederzeit die gesamte historische Datenmenge als Datenstrom erneut zu verarbeiten. Auch die Implementierung des Prototypen von LAKE bietet diese Möglichkeit. Hierfür ist keine vorangehende Stapelverarbeitung nötig.

Die Ausführung ist in Quelltextausschnitt 8.6 zu sehen. Zunächst wird die gesamte historische Datenmenge als Datenstrom eingelesen und mithilfe einer `MapFunction<T,O>` verarbeitet (siehe Zeile 6 – 8). Ist dies erledigt, wird in Zeile 14 die Methode `kappa(DataStream<GeneralType> stream, MapFunction<GeneralType, String> mapFunction)` mit dem Datenstrom und der verwendeten Mapping–Funktion aufgerufen.

Quelltext 8.5 Die Methode zur Verarbeitung analog zu Kappa in Flink

```
1       public void kappa(DataStream<GeneralType> stream, MapFunction<GeneralType, String>
            mapFunction) {
2             // Execute the mapper on the incoming stream.
3             // We assume the whole database was done already
4             DataStream<String> kappaResult = stream.map(mapFunction);
5
6             // Write data
7             kappaResult.writeAsText(CStrings.PATH_RESULT + "Flink" +
                  myConfig.getResultFile(), WriteMode.OVERWRITE);
8       }
```

Quelltext 8.6 Die erneute Verarbeitung historischer Daten mit anschließender Anwendung der Kappa–Verarbeitung in Flink

```
1       public void recalcAll(DataStream<GeneralType> stream) {
2             // Kappa reads in the database and checks for updates
3             StreamExecutionEnvironment env =
                  StreamExecutionEnvironment.getExecutionEnvironment();
4
5             // Work with the stream from the DB
6             KappaRecalcMapper kRM = new KappaRecalcMapper();
7             DataStream<GeneralType> input = env.readTextFile(CStrings.getPathDBFlink())
                  .map(new StringToGeneralTypeMapperFlink());
8             DataStream<String> kappaResultDB = input.map(kRM);
9
10            // Write it
11            kappaResultDB.writeAsText(CStrings.PATH_RESULT + "Flink" +
                  myConfig.getResultFile(), WriteMode.OVERWRITE);
12
13            // Now we are done with the db and start the normal Kappa
14            kappa(stream, kRM);
15
16            try {
17                  env.execute("Recalc All");
18            } catch (Exception e) {
19                  e.printStackTrace();
20            }
21      }
```

Erneute Verarbeitung eines Ausschnitts Ähnlich zur erneuten Verarbeitung aller historischer Daten kann auch ein einzelner Ausschnitt als Datenstrom verarbeitet werden. Hierzu muss zunächst per Stapelverarbeitung ein solcher Ausschnitt aus den historischen Daten erzeugt werden. Anschließend kann dieser als Datenstrom eingelesen und mit einer `MapFunction<T,O>` verarbeitet werden. In diesem Fall wird kein Datenstrom als Eingabe für LAKE benötigt, da alle Daten bereits im System gespeichert sind.

8.2 Verarbeitung durch Machine Learning

Die zweite Implementierung von LAKE erhält Daten von einem Sensor aus dem Bereich der eHealth. Als Beispiel wird ein Puls- und Bewegungssensor verwendet, der beispielsweise zur Erkennung von Epilepsieanfällen dienen kann (vgl. [Fra16]). Hierfür werden die Puls- und Bewegungsdaten des Patienten erfasst und zur Verarbeitung weitergeleitet.

Die Verarbeitung soll in diesem Fall Data Mining–Techniken verwenden. Hierbei wurde ein Klassifikationsalgorithmus gewählt. Die erhaltenen Daten sollen anhand des Alters, des Pulses und der Bewegungsmuster des Patienten einer von drei Kategorien zugeordnet werden. Diese sind „OK“, „RISIKO“, falls es sich um einen Anfall handeln könnte und „NOK“, falls es sich

mit sehr hoher Wahrscheinlichkeit um einen Anfall handelt. Abbildung 8.2 zeigt ein UML–Diagramm des verwendeten Datentyps.

Auch für diesen Anwendungsfall wurden die Daten mithilfe eines dafür geschriebenen Programms generiert. Da für die Trainings- und Testphase des Klassifikationsalgorithmus vorklassifizierte Daten benötigt werden, wurden auch diese generiert. Sie werden als Eingabe für die Stapelverarbeitung genutzt, welche in diesem Fall einen Entscheidungsbaum erstellt. Sowohl Flink als auch Spark stellen bereits von sich aus Möglichkeiten zur Nutzung verschiedener Machine Learning–Algorithmen bereit (siehe [Apa17c; Apa17d]). Zum jetzigen Zeitpunkt allerdings ist FlinkML nur mit der Scala API nutzbar [Apa17a], weshalb es in der Java–basierten Implementierung nicht verwendet werden kann. In der in dieser Arbeit erstellten Implementierung wurde der Entscheidungsbaum darum von Hand erstellt.

Nach der Trainingsphase wird der so entstandene Entscheidungsbaum mithilfe von Testdaten validiert. Es ergibt sich der in Abbildung 8.3 dargestellte Baum, der für die Klassifikation genutzt werden kann.

Ist die Testphase abgeschlossen, kann die Klassifikation auf die Datenobjekte aus dem Datenstrom angewandt werden. Dafür wird die Datenstromverarbeitungskomponente entsprechend angepasst. Zunächst wird der Entscheidungsbaum aus dem HDFS geladen, wo er als Text–Datei gespeichert wurde. Da FlinkML noch nicht mit Java nutzbar ist, wurde im Rahmen der Implementierung eine Mapping–Funktion geschrieben, die den Entscheidungsbaum auf die eingehenden Datenobjekte anwendet.

Diese Mapping–Funktion ist in Quelltextausschnitt 8.7 dargestellt. Hierbei wird für jedes Datenobjekt der Baum durchlaufen, bis ein sogenannter `ClassificationNode` gefunden wird. Dabei handelt es sich um einen Blattknoten, der eine Klassifikation enthält. Zunächst wird in Zeile 3 die Baumtraversierung an der Wurzel gestartet. Dabei wird zwischen Entscheidungsknoten und Klassifizierungsknoten unterschieden.

Die Entscheidungsknoten bedeuten, dass hier ein Wert des Datenobjektes verarbeitet werden muss, um den Baum weiter zu traversieren. Jede Ebene des Baumes hat ein `deciscionValue`

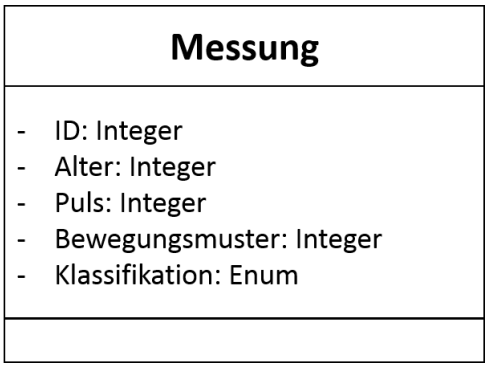

Abbildung 8.2: Das UML–Diagramm für die exemplarischen Messdaten des eHealth–Sensors

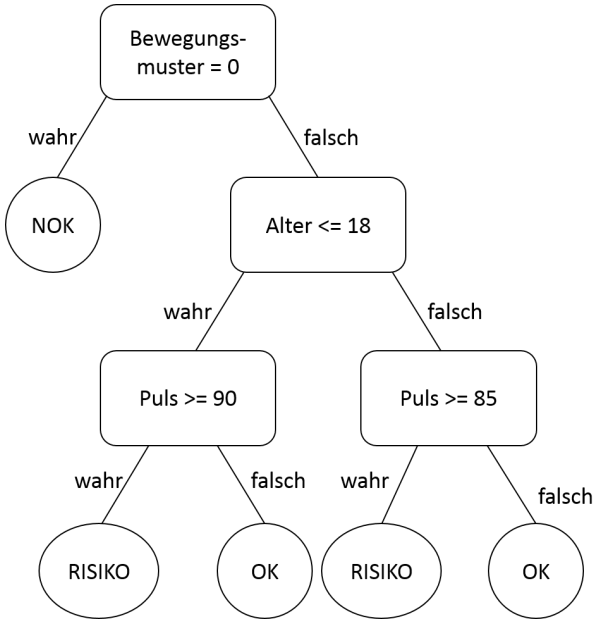

Abbildung 8.3: Der Entscheidungsbaum für die Klassifikation

zugeordnet, dabei handelt es sich um den Wert des Datenobjektes, der betrachtet wird. Dieser wird in den Zeilen 10 bis 18 festgelegt. Anschließend wird der zum Entscheidungsknoten gehörende Operator ausgelesen (Zeile 21 – 31) und direkt errechnet, ob die Bedingung des Knotens für das Datenobjekt erfüllt ist. Abhängig davon wird in Zeile 34 der nächste Knoten ausgewählt.

Sollte der aktuelle Knoten aber ein Klassifizierungsknoten sein, so wird in Zeile 38 die Klassifikation ausgelesen und zu dem Datenobjekt hinzugefügt. Das klassifizierte Datenobjekt wird anschließend zurückgegeben.

Durch die Verbindung zwischen Datenstrom- und Stapelverarbeitungskomponente können alle Phasen des Data Minings mit LAKE ausgeführt werden. Zudem kann im laufenden Betrieb in der Stapelverarbeitungskomponente ein neues Muster erlernt werden, welches gleich anschließend in der Datenstromverarbeitungskomponente Anwendung findet.

Quelltext 8.7 Die Anwendung des Entscheidungsbaumes auf Datenobjekte in Flink

```
1       // Walk through the tree and classify. We only know which variable is needed on
2       // which level (movement, age, pulse)
3       while (i < patternTree.size()) {
4             patternNode = patternTree.get(i);
5             if (patternNode instanceof MLDecisionNodeType) {
6                   // We have to make an decision here
7                   int decisionValue = 0;
8                   boolean isTrue = false;
9
10                  if (i == 0) {
11                        decisionValue = data.getMovement();
12                  } else if (i <= 2) {
13                        // second layer is age
14                        decisionValue = data.getAge();
15                  } else {
16                        // third layer is pulse
17                        decisionValue = data.getPulse();
18                  }
19
20                  // Now check the value against the decisionNode
21                  switch (((MLDecisionNodeType) patternNode).getOperator()) {
22                  case EQ:
23                        isTrue = decisionValue == ((MLDecisionNodeType)
                              patternNode).getValue();
24                        break;
25                  case GET:
26                        isTrue = decisionValue >= ((MLDecisionNodeType)
                              patternNode).getValue();
27                        break;
28                  case LET:
29                        isTrue = decisionValue <= ((MLDecisionNodeType)
                              patternNode).getValue();
30                        break;
31                  }
32
33                  // Traverse the tree further
34                  // ...
35
36            } else {
37                  // patternNode is a ClassificationNode
38                  data.setClassific(((MLClassificationNodeType)
                        patternNode).getClassific());
39                  return data;
40            }
41      }
```

9 Evaluation

Wie bereits die beiden Referenzarchitekturen, Lambda und Kappa, wird auch LAKE in Abschnitt 9.1 anhand der in Kapitel 3 gegebenen Anforderungen evaluiert. In Abschnitt 9.2 wird die Geschwindigkeit der Implementierung betrachtet. Hierzu werden die Verarbeitungszeiten für die Erstellung der Ausschnitte in Flink und Spark, sowie die Dauer der Datenstromverarbeitung in den verschiedenen Verarbeitungsmodi dargestellt.

9.1 Anforderungsevaluation

LAKE wurde dazu entworfen, die Schwächen der Lambda– und der Kappa–Architektur durch eine Kombination dieser beiden auszugleichen. Durch diese Kombination kann LAKE Daten sowohl analog zur Lambda–, als auch zur Kappa–Architektur verarbeiten. Darüber hinaus verfügt LAKE über weitere Möglichkeiten zur Datenverarbeitung, die flexibel während des Betriebs gewechselt werden können. So kann zu Beginn die Kappa–Architektur mit ihrer weniger komplizierten Implementierung verwendet werden, sobald aber die erneute Berechnung der historischen Daten zu viel Zeit benötigt, können die Daten wie in der Lambda–Architektur verarbeitet werden. Je nach Anwendungsfall kann so flexibel die passendste Verarbeitungsart ausgewählt werden.

In der folgenden Auflistung sind die verschiedenen Anforderungen zusammen mit ihrer Umsetzung in LAKE aufgezeigt:

A1 Daten können in Echtzeit verarbeitet werden, ohne dass (alle) Informationen persistent gespeichert werden müssen.

→ In LAKE wird der eingehende Datenstrom zunächst auf die Datenstrom– und die Stapelverarbeitungskomponente aufgeteilt. Somit ist es möglich, im Datenspeicher bestimmte Felder des Datenstroms nicht abzulegen, obwohl diese der Datenstromverarbeitungskomponente zur Verfügung stehen. Damit sind zwar die Daten des persistenten Datenspeichers nicht mehr unverändert, aber Daten, die aus Datenschutzgründen nicht gespeichert werden dürfen, können ausgefiltert werden. Dieser Filter kann in der Stapelverarbeitungskomponente oder bereits beim Weiterleiten an den Datenspeicher realisiert werden.

A2 Es sind Analysen möglich, die ausschließlich auf historischen Daten arbeiten.

→ Um diese Anforderung zu erfüllen, muss lediglich die Stapelverarbeitungskomponente der Architektur verwendet werden. Dieser arbeitet ausschließlich auf historischen Daten und erzeugt aus diesen die Ausschnitte. Auch kann durch die Verwendung des Datenspeichers zuerst ein Ausschnitt erstellt werden, auf welcher dann weitere Verarbeitungen durchgeführt werden können.

A3 Es ist möglich, sowohl historische als auch Stromdaten zu analysieren und diese Ergebnisse zu einem Ergebnis zusammenzufassen. Hierfür können historische Daten vorverarbeitet und Stromdaten in Echtzeit verarbeitet werden, um möglichst schnell akkurate Ergebnisse zu erhalten.

→ Durch die Möglichkeit, die Lambda–Architektur innerhalb von LAKE zu verwenden, ist diese Anforderung erfüllt.

A4 Ergebnisse aus der Verarbeitung historischer Daten können als Grundlage für die Verarbeitung der Stromdaten genutzt werden.

→ Dies ist möglich, indem ein passender Ausschnitt in die Datenstromverarbeitungskomponente geladen und dort mit den Stromdaten kombiniert wird. Da die Ausschnitte mit beliebiger Logik erstellt werden können, stellt die Vorgabe, dass ein Ausschnitt verwendet werden muss, keine Einschränkung dar.

A5 Für einen Bereich der historischen Daten ist es möglich, diesen mit einer veränderten Logik auszuwerten. Diese Auswertung geschieht mit nur wenig Latenzzeit. Dafür wird ein Ausschnitt des Datenspeichers erstellt, welcher anschließend erneut verarbeitet wird.

→ Dies ist durch die Implementierung der Kappa–Architektur innerhalb von LAKE gegeben. Hiermit kann ein Ausschnitt als Eingabe für die Kappa–Architektur verwendet werden.

A6 Historische Daten können mit beliebigen externen Daten verknüpft werden, um Beziehungen zwischen den Datensätzen zu ermitteln.

→ Durch die Nutzung eines Datenbanksystems können die Daten des historischen Datenspeichers mit beliebigen anderen Datenquellen verbunden werden. Die Verarbeitung kann anschließend flexibel in der Datenstrom– oder der Stapelverarbeitungskomponente erfolgen.

A7 Stromdaten können ebenfalls mit Daten aus externen Quellen verknüpft werden und mit ihnen zusammen verarbeitet werden.

→ Auch dies ist gegeben, da die externen Daten als Eingabequelle verwendet werden können. So können sie analog zu den historischen Daten mit dem Datenstrom verbunden werden.

Nachdem in Abschnitt 4.3 bereits die Lambda– und die Kappa–Architektur evaluiert wurden, können diese hier mit LAKE verglichen werden. Wie in Tabelle 9.1 zu sehen, kann LAKE im Gegensatz zu Lambda und Kappa alle gegebenen Anforderungen erfüllen.

Anforderung	Lambda	Kappa	LAKE
A1	✓	✓	✓
A2	✓	✓	✓
A3	✓	✓	✓
A4	X	(✓)	✓
A5	X	X	✓
A6	X	X	✓
A7	X	X	✓

Tabelle 9.1: Der direkte Vergleich zwischen Lambda, Kappa und LAKE (vgl. Tabelle 4.2)

Durch die Kombination aus der Lambda- und der Kappa–Architektur kann LAKE viele der Schwächen der beiden Architekturen ausgleichen. So kann in Anwendungen die Verarbeitung zunächst analog zur Kappa–Architektur stattfinden, solange die Latenzzeit der erneuten Verarbeitung nicht zu hoch ist. Auch, wenn nur wenige Änderungen in der Auswertungslogik stattfinden, kann die Verarbeitung nach Kappa ausgeführt werden. Müssen allerdings mehrere Ausgaben mit verschiedenen Logiken erzeugt werden oder wird die Latenzzeit einer Neuberechnung zu hoch, kann mit LAKE flexibel die Verarbeitungsart zur Lambda–Architektur gewechselt werden. So kann ist die Implementierung der Stapelverarbeitungskomponente nur dann nötig, wenn die Kappa–Architektur selbst an ihre Grenzen stößt. Der Wechsel ist dabei im laufenden Betrieb möglich. Zudem kann durch die Verwendung eines Systems wie z.B. Flink, welches sowohl Datenströme als auch endliche Datenmengen gleich behandelt, der Aufwand der Implementierung reduziert werden.

9.2 Messungen

Ziel der Entwicklung von LAKE ist eine Architektur, die sowohl große Mengen an Daten als auch Daten aus einem Datenstrom in Echtzeit verarbeiten kann. Dies wird durch die separaten Datenstrom– und Stapelverarbeitungskomponenten gewährleistet.

Um LAKE zu testen, wurde im Rahmen dieser Arbeit ein Programm geschrieben, welches csv–Dateien mit beliebigen Einträgen generiert. Aus einer solchen Datei wird mit Flink der Datenstrom als Eingabe für LAKE erzeugt. Eine weitere Datei dient als Datensammlung historischer Daten. Auch kann eine dritte Datei mit weiteren Datensätzen als externe Datenquelle eingebunden werden.

In der Datenstromverarbeitungskomponente können die Daten im Millisekundenbereich verarbeitet werden. Abbildung 9.1 zeigt, wie viel Zeit für die Verarbeitung eines Datenobjektes in den verschiedenen Modi benötigt wird. Weitere Verzögerungen durch beispielsweise das

Einlesen externer Datenquellen werden dabei vernachlässigt, da diese nur in größeren zeitlichen Abständen nötig sind. Aus dem Diagramm wird sichtbar, dass die Verarbeitung in den meisten Fällen zwischen 40 und 50 Millisekunden liegt. Schwankungen, wie beispielsweise bei der Implementierung der Lambda– und der Kappa–Architektur, ergeben sich durch die konkrete Umsetzung im Prototypen.

In der Stapelverarbeitungskomponente wurde die Zeitdauer für die Erstellung eines Ausschnitts ermittelt. Hierfür wurde die Zeit vom Einlesen des Datenspeichers bis zu dem Zeitpunkt, zu dem ein Ergebnis zur Verfügung steht, gemessen. Da die Stapelverarbeitungskomponente sowohl in Flink als auch in Spark implementiert wurde, kann ein direkter Vergleich zwischen beiden Implementierungen gezogen werden.

Beide basieren auf denselben SQL–Anfragen. Die erste SQL–Anfrage wählt aus einer vorhandenen Datenmenge drei Spalten aus, welche die Ergebnismenge darstellen: `SELECT part, job, noDone FROM HistDB`. Die zweite SQL–Anfrage dagegen beinhaltet einen `JOIN`, um die historischen mit externen Daten zu verknüpfen: `SELECT id,job,material,temp,maxTemp FROM HistDB AS h JOIN ExtDB AS e ON h.material = e.materialId WHERE h.temp > e.maxTemp`.

Für die Messungen wurde der persistente Datenspeicher mit 10^4 bis 10^7 Einträgen befüllt. Im Fall des zweiten SQL–Anfrage verfügte der externe Datenspeicher über fünf Einträge. Die

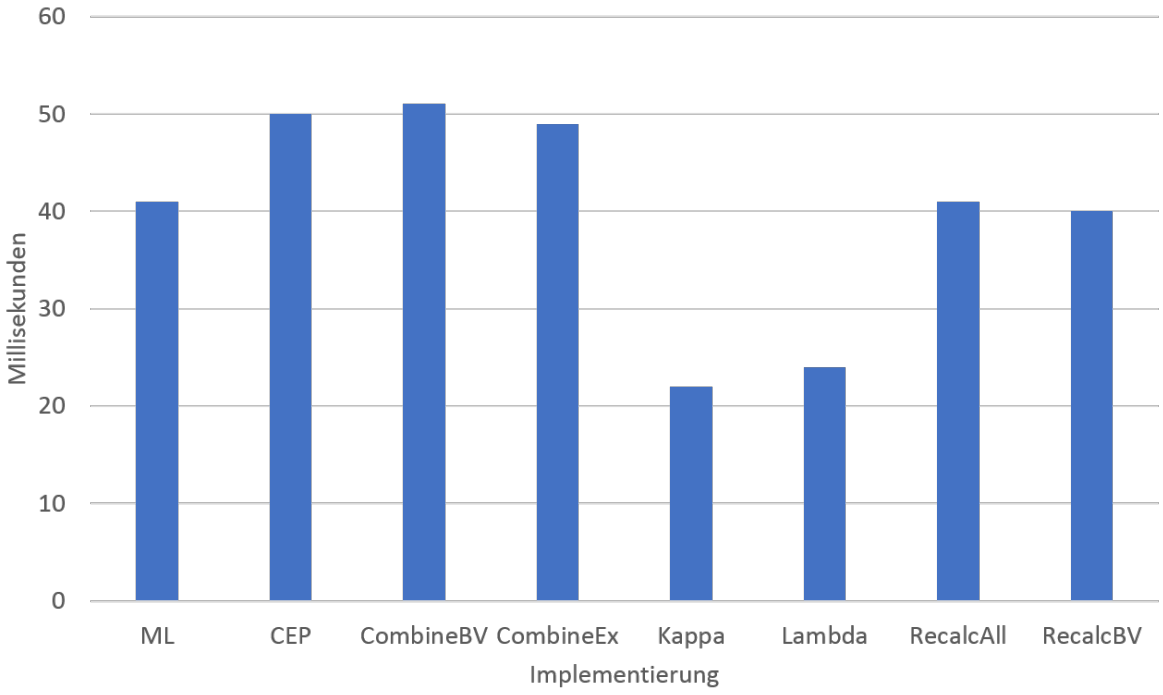

Abbildung 9.1: Die Zeitdauer zur Verarbeitung eines Datenobjekts in der Datenstromkomponente

Zeiten für die Erstellung der Ausschnitte sind in Tabelle 9.2 zu sehen. Eine visuelle Darstellung bietet Abbildung 9.2.

	Flink		Spark	
Anzahl Einträge	Einfaches SQL	Komplexes SQL	Einfaches SQL	Komplexes SQL
10^4	2.534	3.091	3.673	4.635
10^5	2.550	3.236	4.564	6.960
10^6	2.572	3.152	8.878	12.597
10^7	2.534	3.091	3.673	77.558

Tabelle 9.2: Die Messdaten für die Erstellung einer Batch View in Millisekunden

In Tabelle 9.2 und in Abbildung 9.2 ist zu sehen, dass Flink die Ausschnitte schneller erstellt als Spark. Weder die wachsende Anzahl an Einträgen im persistenten Datenspeicher, noch die Komplexität der zweiten SQL–Anfrage verändern die benötigte Zeit erheblich. Spark dagegen benötigt deutlich länger, je mehr Einträge im Datenspeicher vorhanden sind. Auch das komplexere SQL–Statement lässt die Verarbeitungszeit steigen. Dies liegt vermutlich daran, dass Flink auch für die Stapelverarbeitung auf Datenströme zurückgreift. Somit können die ersten Einträge bereits verarbeitet und abgespeichert werden, bevor der gesamte Datenspeicher eingelesen wurde. Spark dagegen setzt auf traditionelle Stapelverarbeitung, für die zuerst große Mengen an Daten geladen werden, bevor die eigentliche Verarbeitung beginnt.

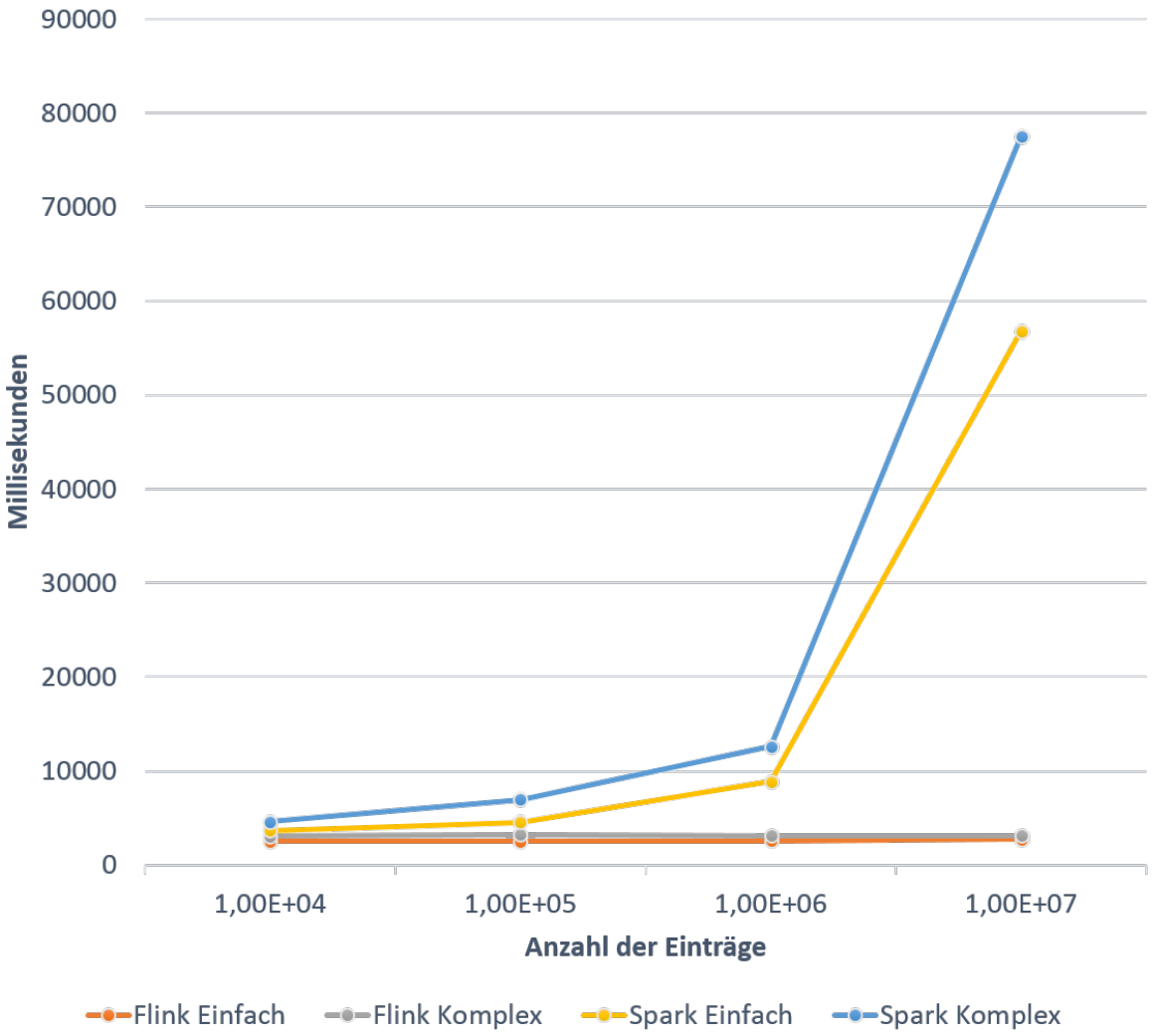

Abbildung 9.2: Die Zeitdauer zur Erstellung einer Batch View

10 Zusammenfassung und Ausblick

Das IoT verändert die Art, wie Daten erfasst und verarbeitet werden und bringt dabei neue Probleme mit sich [SRF15]. Um beispielsweise in der Industrie 4.0 oder in der eHealth schnell auf kritische Messungen reagieren zu können, müssen Daten in Echtzeit verarbeitet werden. Dies kann mithilfe der Datenstromverarbeitung erreicht werden [CY15; WGFR16].

Dabei werden allerdings die Datenobjekte einzeln betrachtet, was zu Ungenauigkeiten führen kann [CY15]. Darum ist eine Verbindung mit einer Verarbeitungsmethode wünschenswert, welche auch große Mengen an Daten verarbeiten kann, wie die Stapelverarbeitung.

Um dies zu erreichen, existieren die Lambda– und die Kappa–Architektur [Kre14; MW15]. Beide bieten die Möglichkeit, sowohl Datenströme als auch den Inhalt eines persistenten Speichers zu verarbeiten. Allerdings besitzen beide auch Schwachstellen: Die Lambda–Architektur trennt streng zwischen Datenstrom– und Stapelverarbeitung, während die Kappa–Architektur nur auf Datenströmen arbeitet. Dagegen steigt die Verarbeitungszeit in der Kappa–Architektur linear mit der Größe der zu verarbeitenden Datenmenge.

Ziel dieser Arbeit ist es, diese Schwachstellen zu umgehen. Hierfür wurden zunächst Anwendungsfälle identifiziert, in denen sowohl einzelne Datenobjekte in Echtzeit als auch große Datenmengen verarbeitet werden müssen. Diese Anwendungsfälle stellten bestimmte Anforderungen an eine Architektur. Sowohl die Lambda–, als auch die Kappa–Architektur wurden gegen die Anforderungen evaluiert. Beide können jedoch nicht alle Anforderungen erfüllen.

Die Kombination beider Architekturen dagegen bietet einen Großteil der benötigten Funktionen. Darum wurde im Rahmen dieser Arbeit ein Konzept für einen Hybriden entwickelt, die so genannte Lambda–Kappa–Architektur, kurz LAKE. Zudem verfügt LAKE über weitere Verwendungsmöglichkeiten: Es können externe Quellen und Konfigurationen eingebunden werden, außerdem können auch Ausschnitte des Datenspeichers als Datenstrom verarbeitet werden.

Um LAKE umsetzen zu können, wurden in dieser Arbeit verschiedene Techniken und Systeme zur Verarbeitung von Daten betrachtet. Anschließend wurden für diese Arbeit zwei Implementierungen von LAKE umgesetzt. Die eine Umsetzung bietet dabei die Möglichkeit, flexibel zwischen Verarbeitungsmodi zu wechseln, während die andere für die Datenverarbeitung auf Data Mining–Algorithmen zurückgreift. LAKE kann also durch eine geeignete Implementierung flexibel auf beliebige Anwendungsfälle angepasst werden. Dies wird durch eindeutig definierte Schnittstellen unterstützt, sodass auch die einzelnen Komponenten flexibel austauschbar sind.

Abschließend wurden Konzept und Implementierungen evaluiert. Dabei ergab sich, dass LAKE die gegebenen Anforderungen erfüllen kann. Zudem können die Daten, die per Datenstrom eintreffen, im Millisekundenbereich verarbeitet werden.

Durch die Implementierung der Stapelverarbeitung mit sowohl Apache Flink als auch Apache Spark konnten zudem diese beiden Systeme miteinander verglichen werden. Hierbei zeigt sich, dass Flink die Daten des persistenten Speichers schneller verarbeitet als Spark.

Ausblick

LAKE bietet verschiedene Möglichkeiten zur Datenverarbeitung. Durch die Möglichkeit zur Kombination von Datenstrom– und Stapelverarbeitung, kann LAKE in verschiedenen Anwendungsfällen eingesetzt werden. Ihr allgemeines Konzept unterstützt dabei beliebige Implementierungen und die Verwendung verschiedenster Verarbeitungssysteme, wie Flink, Spark, Hadoop, Samza, Storm u.v.m. Hierbei können weitere Vergleiche von Implementierungen von LAKE erstellt werden, um die passendsten Systeme für den jeweiligen Anwendungsfall zu finden.

Zudem kann LAKE beliebig erweitert werden. So können beispielsweise weitere Konfigurationsmöglichkeiten für die Verarbeitung eingefügt werden, sowohl in der Datenstrom– als auch in der Stapelverarbeitungskomponente. Des Weiteren kann eine Ergebniskomponente eingefügt werden, welche die Ergebnismengen der beiden anderen Komponenten kombiniert und beliebig weiterverwendet.

Auch im Bereich der Privacy kann LAKE weiterentwickelt werden. Durch Verarbeitungen auf verschiedenen Ebenen kann beispielsweise sicher gestellt werden, dass keine unautorisierten Personen Zugriff auf sensible Daten erhalten. Mit entsprechenden Anpassungen könnte LAKE beispielsweise in Patron [PAT16] Anwendung finden.

Durch die zunehmende Nutzung des IoT und ähnlicher Prinzipien wird die Kombination aus Datenstrom– und Stapelverarbeitung vermutlich an Bedeutung gewinnen. Die Menge der bereits gesammelten Daten nimmt zu, ebenso die Anzahl an Datenobjekten, die per Datenstrom eintreffen. LAKE findet hier ihre Anwendung und kann dazu beitragen, Daten flexibler zu verarbeiten.

Literaturverzeichnis

[ACÇ+03] D. J. Abadi, D. Carney, U. Çetintemel, M. Cheriniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik. „Aurora: a new model and architecture for data stream management“. In: *The VLDB Journal* (2003) (zitiert auf S. 58, 59).

[Apa15] Apache Software Foundation. *Apache Storm*. 2015. URL: https://storm.apache.org/ (zitiert auf S. 59).

[Apa17a] Apache Flink User Mailing List archive. *Using FlinkML from Java?* 2017. URL: http://apache-flink-user-mailing-list-archive.2336050.n4.nabble.com/Using-FlinkML-from-Java-td12696.html (zitiert auf S. 72).

[Apa17b] Apache Software Foundation. *Apache Hadoop*. 2017. URL: http://hadoop.apache.org/ (zitiert auf S. 58–60).

[Apa17c] Apache Software Foundation. *Apache Spark*. 2017. URL: http://spark.apache.org/ (zitiert auf S. 59, 60, 72).

[Apa17d] Apache Software Foundation. *Flink*. 2017. URL: http://flink.apache.org/ (zitiert auf S. 61, 62, 67–69, 72).

[Apa17e] Apache Software Foundation. *Samza*. 2017. URL: http://samza.apache.org/ (zitiert auf S. 61).

[BBC+09] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, M. Grossniklaus. „Continuous Queries and Real-time Analysis of Social Semantic Data with C-SPARQL“. In: *ISWC International Semantic Web Conference, Workshop on Social Data on the Web*. 25. Okt. 2009 (zitiert auf S. 55, 56).

[CBMS15] R. Cortés, X. Bonnaire, O. Marin, P. Sens. „Stream processing of healthcare sensor data: studying user traces to identify challenges from a big data perspective“. In: *Procedia Computer Science* 52 (2015), S. 1004–1009 (zitiert auf S. 13, 17).

[CM12] G. Cugola, A. Margara. „Processing Flows of Information: From Data Stream to Complex Event Processing“. In: *ACM Computing Surveys (CSUR)* 44 (3 Juni 2012) (zitiert auf S. 56, 57).

[CY15] R. Casado, M. Younas. „Emerging trends and technologies in big data processing“. In: *Concurrency and Computation: Practice and Experience* (2015) (zitiert auf S. 13, 27, 28, 57–61, 81).

[DG04] J. Dean, S. Ghemawat. „MapReduce: Simplified Data Processing on Large Clusters“. In: *OSDI Symposium on Operating System Design and Implementation*. Dez. 2004 (zitiert auf S. 55, 57).

[Fra16] Fraunhofer–Institut für Software– und Systemtechnik ISST. *EPITECT – Sensorische Anfallsdetektion*. Forschungsber. Fraunhofer–Institut für Software– und Systemtechnik ISST, März 2016 (zitiert auf S. 21, 71).

[GL02] S. Gilbert, N. Lynch. „Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services“. In: *ACM SIGACT News* 33 (2 Juni 2002), S. 51–59 (zitiert auf S. 29).

[GS17] C. Giebler, C. Stach. „Datenschutzmechanismen für Gesundheitsspiele am Beispiel von Secure Candy Castle“. In: *BTW (Workshops)*. 2017, S. 311–320 (zitiert auf S. 17).

[Has17] W. Hassan. *Big Data Frameworks*. 14. Feb. 2017. URL: https://www.linkedin.com/pulse/big-data-frameworks-dr-waleed-hassan-pmp-cbip-bdscp-togaf-itil (zitiert auf S. 27, 28).

[Inf10] Informatica. *Complex Event Processing – Relevant, Timely and Trustworthy Decisions for End-Users*. Forschungsber. Informatica, 3. Nov. 2010 (zitiert auf S. 56, 57).

[Jes14] K. Jeschke. *Industrie 4.0 Predictive Maintenance*. Software. SAP Deutschland AG & Co. KG, Feb. 2014 (zitiert auf S. 18).

[KNR11] J. Kreps, N. Narkhede, J. Rao. „Kafka: a Distributed Messaging System for Log Processing“. In: *NetDB Networking meets Databases*. 12. Juni 2011 (zitiert auf S. 60, 61).

[Kre14] J. Kreps. *Questioning the Lambda Architecture*. 2. Juli 2014. URL: https://www.oreilly.com/ideas/questioning-the-lambda-architecture (zitiert auf S. 14, 28, 31–35, 70, 81).

[LIX17] X. Liu, N. Iftikhar, X. Xie. „Survey of Real-time Processing Systems for Big Data“. In: *IDEAS '14* (7. Juli 2017) (zitiert auf S. 17, 55, 61).

[MG14] W. MacDougall, Germany Trade & Invest. *Industrie 4.0 - Smart Manifacturing for the Future*. Forschungsber. GTAI, Juli 2014 (zitiert auf S. 18).

[MW15] N. Marz, J. Warren. *Big Data - Principles and best practices of scalable real-time data systems*. Manning Publications Co., 2015 (zitiert auf S. 14, 28–32, 81).

[PAT16] PATRON Project Members. *Patron*. 2016. URL: http://patronresearch.de/ (zitiert auf S. 82).

[SBG+15] V. Stantchev, A. Barnawi, S. Ghulam, J. Schubert, G. Tamm. „Smart Items, Fog and Cloud Computing as Enablers of Servitization in Healthcare“. In: *Sensors & Transducers* 185 (2 Feb. 2015), S. 121–128 (zitiert auf S. 13, 17).

[SDF+13] A.-W. Scheer, W. Dorst, T. Feld, S. Gerlach, M. Hämmerle, M. Hoffmann, H. Kagermann, T. Krause, C. Kurz, J. Schindler, J. Schlick, S. Schlund, R. Schmidt, F. Simon, D. Spath, P. Stephan, W. Wahlster, D. Zühlke. *Industrie 4.0 – Wie sehen Produktionsprozesse im Jahr 2020 aus?* Hrsg. von A.-W. Scheer. Jan. 2013 (zitiert auf S. 13, 17).

[Sha08] R. Shapire. *Machine Learning Algorithms for Classification.* Princeton University, 2008 (zitiert auf S. 65).

[SJDN06] K. Stroetmann, T. Jones, A. Dobrev, V. N Stroetmann. *eHealth is Worth it The economic benefits of implemented eHealth solutions at ten European sites.* Jan. 2006 (zitiert auf S. 17).

[SKRC10] K. Shvachko, H. Kuang, S. Radia, R. Chansler. „The Hadoop Distributed File System". In: *IEEE Symposium on Mass Storage Systems and Technologies.* 3. Mai 2010 (zitiert auf S. 58, 59).

[SRF15] N. Spangenberg, M. Roth, B. Franczyk. „Evaluating New Approaches of Big Data Analytics Frameworks". In: *Business Information Systems.* 24. Juni 2015, S. 28–37 (zitiert auf S. 13, 81).

[TGNO92] D. Terry, D. Goldberg, D. Nichols, B. Oki. „Continuous Queries over Append-Only Databases". In: *SIGMOD 1992.* 2. Juni 1992, S. 321–330 (zitiert auf S. 55, 56).

[VMD+13] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, E. Baldeschwieler. „Apache Hadoop YARN: Yet Another Resource Negotiator". In: *SOCC Symposium on Cloud Computing.* 1. Okt. 2013 (zitiert auf S. 61).

[WGFR16] W. Wingerath, F. Gessert, S. Friedrich, N. Ritter. „Real-time stream processing for Big Data". In: *Methods and Applications of Informatics and Information Technology* 58.4 (2016), S. 186–194 (zitiert auf S. 13, 14, 35, 55, 59–62, 81).

[WSM+17] M. Wieland, F. Steimle, B. Mitschang, D. Lucke, P. Einberger, D. Schel, M. Luckert, T. Bauernhansl. „Rule-Based Integration of Smart Services Using the Manufacturing Service Bus". In: *IEEE International Conference on Ubiquitous Intelligence and Computing.* 2017, S. 0 – 8 (zitiert auf S. 18).

[ZCF+10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica. „Spark: cluster computing with working sets". In: *Hot Cloud Hot topics in cloud computing.* 22. Juni 2010 (zitiert auf S. 59, 60).

Alle URLs wurden zuletzt am 15. 10. 2017 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift