Entwurf und Implementierung einer generischen Ingestion-Schnittstelle mit Versionierung für Data Lake Systeme

Masterarbeit
zur Erlangung des Grades *Master of Science*

an der
Hochschule Niederrhein
Fachbereich Elektrotechnik und Informatik
Studiengang *Informatik*

vorgelegt von
Alexander Martin
1018332

Datum: 28. November 2021

Prüfer: Prof. Dr. rer. nat. Christoph Quix
Zweitprüfer: M.Sc. Sayed Hoseini

Eidesstattliche Erklärung

Name: Alexander Martin
Matrikelnr.: 1018332
Titel: Entwurf und Implementierung einer generischen Ingestion-Schnittstelle

Ich versichere durch meine Unterschrift, dass die vorliegende Arbeit ausschließlich von mir verfasst wurde. Es wurden keine anderen als die von mir angegebenen Quellen und Hilfsmittel benutzt.

Die Arbeit besteht aus _____ Seiten.

_____ _____

Ort, Datum Unterschrift

Zusammenfassung

Abstract

Inhaltsverzeichnis

Kapitel 1

Einleitung

Heutzutage spielen Daten und deren Verarbeitung in vielen Bereichen eine immer wichtigere Rolle. Im *Rethink Data Report 2020* SEAGATE TECHNOLOGY (2020) wurde eine Studie durchgeführt, die eine Steigerung von 42% der Menge an anfallenden Daten pro Jahr prognostiziert. Dies wird unter anderem auf den vermehrten Einsatz von IoT-Geräten, immer ausführlichere Datenanalysen und die einfachere werdende Anwendung von Cloud-Speichern zurückgeführt. Dabei besteht die Herausforderung, Daten in verschiedensten Formaten in großen Mengen zu verwalten und zu verwenden.

Als eine Lösung für das Problem haben sich Data-Lake-Systeme hervorgetan. Ein Data Lake ist eine Methodik, die die Erfassung, Verfeinerung, Archivierung und Erkundung von Daten verbessert (Fang 2015). Große Datenmengen sollen möglichst kostensparend gespeichert werden. Verschiedenste Formate können in einem Data Lake abgelegt und verarbeitet werden. Darunter zählen zum Beispiel strukturierte Daten aus traditionellen Datenbanksystemen, semistrukturierte Daten wie JSON-Dateien, bei denen nicht alle Attribute festgelegt sind und unstrukturierte Daten wie Bilder oder Videos (Singh und Ahmad 2019).

1.1 Zielsetzung der Arbeit

In einer Reihe von Projekten und Masterarbeiten wird an der Hochschule Niederrhein ein generelles Data-Lake-System entwickelt. Das heißt es ist überall einsetzbar und alle nötigen Komponenten sind im System enthalten. Ein Prototyp wurde bereits während eines Masterprojekts entwickelt (Martin, Thiel und Kuller 2020/21).

Diese Masterarbeit befasst sich mit der Entwicklung, Implementierung und Integration einer neuen Schnittstelle für die Aufnahme von Daten (Ingestion). Die wichtigsten Ziele sind dabei die Generalität, Kontinuität und Datenversionierung bei einer möglichst einfachen Verwendung für den Benutzer. Die Generalität bezeichnet die Funktionalität unabhängig von der Art der Datenquelle oder der Struktur der Daten. Unter Kontinuität ist das erneute und optional auch regelmäßige Nachladen von Daten aus einer Quelle gemeint. Für diese Daten soll es auch möglich sein eine Versionierung durch zu führen um Änderungen auswerten zu können.

Der Aufbau der Arbeit gliedert sich in fünf Kapitel. In diesem Kapitel werden wichtige Grundlagen für die Arbeit vermittelt und verwandte Arbeiten erläutert. Danach wird auf die Ziele und die daraus folgenden Anforderungen an die Schnittstelle eingegangen. Das nächste Kapitel befasst sich mit der Entwicklung einer Architektur und des Designs der Ingestion. Auf diesem Design baut die darauf folgende Umsetzung auf. Zuletzt wird das System durch funktionale Tests und Benchmarks evaluiert.

1.2 Grundlagen und Verwandte Arbeiten

1.2.1 Apache Spark

Apache Spark ist eine Datenverabeitungs-Engine. Das ist Ziel ist die Vereinheitlichung von Arbeitsabläufen. Darunter fallen zum Beispiel Arbeiten mit SQL, Datenströmen, maschinellem Lernen oder Graphen-Daten.

Es gibt Bibliotheken für diese oder andere Arbeitsabläufe, die durch ihre Optimierung in Spark ähnliche Performance, wie spezialisierte Engines erreichen. Spark kann entweder lokal auf einem Computer oder auf einem Spark-Cluster nach dem Master-Worker-Modell ausgeführt werden.

Ein Kernprinzip ist die Abstraktion der Daten in RDDs (Resilient Distributed Datasets, deutsch: Robuste Verteilte Datensätze). RDDs sind fehlertolerante Sammlungen von Objekten, die auf den Workern verteilt werden und parallel bearbeitet werden können. Diese werden flüchtig im Speicher gehalten, können aber für einen schnelleren Zugriff zwischengespeichert werden. Die Erstellung und Bearbeitung von RDDs geschieht über sogenannte Transformationen. Die Transformationen werden in einem Herkunftsgraphen gespeichert, wodurch eine Wiederherstellung bei Fehler an jedem Punkt möglich ist.

Für die Verarbeitung von strukturierten oder semi-strukturierten Daten gibt es zusätzlich die eigene Abfragesprache SparkSQL. Spark biete APIs für die Sprachen Scala, Java, Python und R. Auf den RDDs gibt es noch eine weitere Abstraktionsebene. Mit DataFrames, die eine Sammlung RDDs von Datensätzen mit einem bekannten Schema sind, kann eine API benutzt werden, bei der die Bearbeitung der Daten über Funktionsaufrufe statt SparkSQL möglich ist (ZAHARIA u. a. 2016).

Die Interaktion mit einem Spark Cluster kann über eine interaktive Shell oder als fertige Anwendung geschehen. Informationen über die Anwendung werden im SparkContext gespeichert. Beim Lesen und Schreiben wird das Format der Daten angegeben. Spark unterstützt standardmäßig einige Formate, aber durch die Konfiguration von extra Bibliotheken im SparkContext, können beliebige Formate hinzugefügt werden. Von der verwendet Bibliotheken sind auch die Optionen abhängig, die beim Lesen und Schreiben gesetzt werden müssen. Die Optionen sind immer Schlüssel-Wert-Paare und enthalten zum Beispiel Verbindungsinformationen zu einer Datenbank oder einen Dateispeicherort (Apache Spark 2021a).

1.2.2 Hadoop Distributed File System

Das Hadoop Distributed File System (HDFS) ist ein verteiltes und fehlertolerantes Dateisystem. Es wurde entwickelt um auf Hardware mit geringen Kosten zu laufen und große Datenmengen zu verarbeiten. Im HDFS gespeicherte Dateien können von einem Gigabyte bis mehrere Terabyte groß sein. Die Fehlertoleranz wird dabei durch die Möglichkeit der Replikation mit einem beliebigen Faktor gegeben.

Das Dateisystem ist ähnlich zu anderen bekannten Dateisystemen. Dateien und Ordner können im Namensraum hierarchisch organisiert werden. Es unterstützt jedoch keine Zurgiffsberechtigung oder Hard- und Soft-Links. Um einfach und effektiv kohärent zu bleiben, werden Dateien nur einmal geschrieben, können aber mehrfach gelesen werden. Dateien werden zur Speicherung in einzelne Blöcke aufgeteilt. Dabei sind für eine Datei alle, bis auf der letzte, Blöcke gleich groß.

Ein HDFS-Cluster funktioniert nach dem Master-Worker-Prinzip und besteht aus einem NameNode und vielen DataNodes. Der NameNode übernimmt die Verwaltung des Namensraum und Verteilung der einzelnen Blöcke einer Datei. Er reguliert dazu noch den Zugriff durch Clients und führt Operationen auf dem Dateisystem, wie das Öffnen, Schließen oder Umbenennen von Ordnern und Dateien aus. Die DataNodes speichern die einzelnen Blöcke der Dateien. Auf die Anweisung des NameNodes werden Blöcke erstellt, gelöscht oder repliziert. Außerdem bearbeiten sie Anfragen zum Lesen und Schreiben von Dateien (Borthakur 2010).

Mit Apache Parquet steht ein Format zur Verfügung, mit dem Daten im HDFS effizient gespeichert werden können. Parquet ist ein spaltenorientiertes Speicherformat, das auch die Kompression und Kodierung der Daten unterstützt. Auch die Speicherung von verschachtelten Daten ist möglich.

1.2.3 Apache Kafka

Apache Kafka ist ein verteiltes Event-Streaming-System. Die Vermittlung von Events nach dem Publish-Subscribe-Muster läuft dabei in Echtzeit. Events können von Produzenten veröffentlicht werden und Konsumenten können auf diese Events abonnieren. Durch seine Verteilung kann Kafka den Ausfall einzelner Server ausgleichen. Zusätzlich können Ströme von Events für einen beliebigen Zeitraum persistiert werden.

Kafka besteht aus einem Cluster von Servern und verschiedenen Clients. Es gibt zwei Arten von Servern. Die sogenannten Broaker sind für die Verteilung und Verwaltung von Events zuständig. Andere verwenden Kafka Connect[1] um existierende Systeme, zum Beispiel eine Datenbank, in das Kafka Cluster zu integrieren. Die Clients sind Anwendungen die entweder Events produzieren oder konsumieren.

In diesem System repräsentiert ein Event den Fakt, dass etwas „passiert“ ist und besteht aus einem Schlüssel, einem Wert, einem Zeitstempel und optionalen Metadaten. Dabei werden die Werte nicht interpretiert sonder einfach als Block versendet und können so beliebige Struktur haben. Events werden in sogenannte Topics unterteilt. Es kann immer mehrere Produzenten oder Konsumenten auf einer Topic geben. Events in einer Topic können mehrfach gelesen werden und werden nicht nach dem Konsumieren gelöscht. Es kann aber für jede Topic einzeln eine Dauer festgelegt werden, nach der die Events verworfen werden. Um eine Topic fehlertolerant zu machen, kann diese repliziert werden.

Topics werden in Partitionen über verschiedene Broaker aufgeteilt, so dass das ganze System gut skalierbar wird. Ein Produzenten kann zum Beispiel Events auf mehreren Brokern gleichzeitig veröffentlichen. Wenn ein Event in einer Topic veröffentlicht wird, wird dieses an eine der Partitionen angehängt. Events, die den gleichen Schlüssel haben werden immer der gleichen Partition zugeordnet und Events einer Partition kommen

[1] https://kafka.apache.org/documentation/#connect

garantiert in der Reihenfolge des Schreibens bei dem Konsumenten der Partition an (Apache Kafka 2021).

1.2.4 Delta Lake

Delta Lake ist eine extra Speicher-Ebene, die auf dem HDFS angewendet werden kann. Das Ziel ist es diesen Speichern ACID Transaktionen, schnelles Arbeiten mit Metadaten der Tabelle und eine Versionierung der Daten hinzuzufügen. Daten werden in sogenannten Delta Tabellen mit Metadaten und Logs gespeichert.

Eine Delta Tabelle ist ein Verzeichnis im Speicher. Die tatsächlichen Daten werden hier in Apache Parquet Dateien abgelegt. Parquet ist ein spalten-orientiertes Format, zum effizienten Speichern von Daten (Apache Parquet 2021). Dabei können die Daten auch noch in Unterverzeichnisse aufgeteilt sein, zum Beispiel für jedes Datum ein Verzeichnis. Neben den Datenverzeichnissen gibt es eines für die Logs in Form von JSON Dateien mit aufsteigender Nummerierung. Metadaten werden sowohl innerhalb der Parquet als auch der Log Dateien passend gespeichert.

Im Delta Lake wird ein Protokoll für den Zugriff verwendet, dass es ermöglicht, dass mehrere Clients gleichzeitig Lesen und immer nur einer Schreiben kann. Dabei werden beim Schreiben immer erst neue Datensätze, die zur Tabelle hinzugefügt werden sollen in das korrekte Verzeichnis geschrieben. Danach wird eine neu Log Datei erstellt.

Beim Lesen werden die Log Dateien als Grundlage verwendet um daraus zusammen mit den gespeicherten Daten den Tabellen stand zu erzeugen. Man kann beim Lesen auch eine bestimmte Version angeben. Um den Aufwand bei der Verarbeitung der Logs zu verringern wird periodisch ein Kontrollpunkt erzeugt, bei dem alle vorherigen Logs zusammengefügt und komprimiert werden. Das bedeutet, dass zum Beispiel das Operationen die sich gegenseitig aufheben nicht gespeichert werden. Damit reicht es aus nur den letzten Checkpoint vor der zu lesenden Version und alle

darauf folgende Logs zu lesen.

Durch das Design werden keine eignen Server für die Pflege der Delta Tabellen benötigt, sondern dies kann alles über den Client gemacht werden. Der Delta Lake unterstützt sowohl die Batch-Verarbeitung von Daten als auch Datenströme und bietet volle Integration in Spark (Armbrust u. a. 2020).

1.2.5 Der Data-Lake-Prototyp

Ein Data Lake basiert darauf, dass zuerst alle Daten in ihrem Rohformat gespeichert werden. Dadruch fällt der Aufwand einer Transformation bei der Integration von neuen Daten weg. Gleichzeitig bleiben auch alle Informationen bleiben für Analysen erhalten (Jarke und Quix 2017). Neben dem Speichern und Bereitstellen von Daten ist eine weitere Aufgabe eines Data-Lake-Systems die Verwaltung von Metadaten.

Eine Architektur für einen Data Lake ist nicht vorgegeben und variiert mit den Anforderungen. Dabei gibt es jedoch einige Stufen, die die Daten im Data Lake durchlaufen. Die erste ist die Aufnahme, wobei die Daten aus unterschiedlichen Quellen geladen und gespeichert werden. Der nächste Schritt ist die Extraktion von Metadaten. Zum Schluss werden die Daten wieder für die Verwendung aus dem internen Speicher geladen (Fang 2015).

Im Masterprojekt *Development of a Data Lake System* (Martin, Thiel und Kuller 2020/21) an der Hochschule Niederrhein wurde ein Prototyp für ein Data-Lake-System entwickelt. In Abbildung 1.1auf der nächsten Seite ist ein Überblick über dessen Architektur zu sehen. Es handelt sich hierbei um eine Client-Server-Anwendung. Der Client besteht aus einer Web-Anwendung über die Benutzer mit dem Data-Lake-System interagieren. Er kommuniziert mit dem Server über eine REST-API, die auch durch andere Clients verwendet werden könnte. Die Datenverarbeitung wird über ein Apache Spark Cluster gelöst. Zum Speichern der Daten

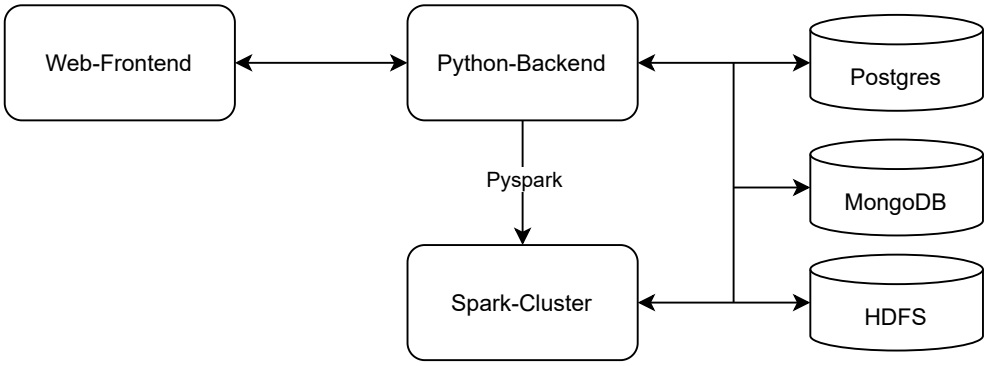


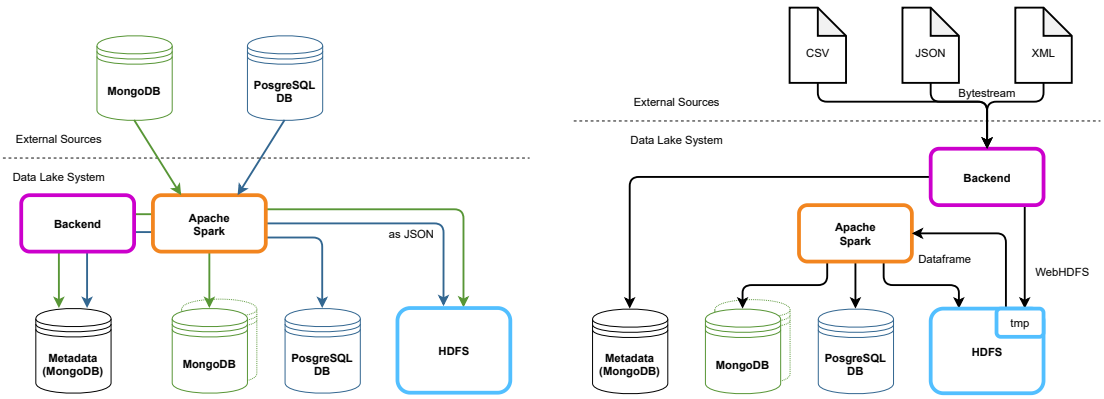
Abbildung 1.1: Architektur des Prototypen, Quelle: *Development of a Data Lake System*, S. 2

stehen drei verschieden System zu Verfügung. Eine Postgres Datenbank für strukturierte, eine MongoDB für semistrukturierte und ein HDFS für unstrukturierte Daten.

Die Verarbeitung der Ingestion ist im Prototyp Abhängig von der Datenquelle und dem ausgewählten Zielspeicher. In Abbildung 1.2auf der nächsten Seite sind die verschiedenen Wege zu sehen. Diese Verarbeitungsweise hat zwei Probleme, die in der neuen Ingestion gelöst werden müssen. Dadurch, dass der Benutzer aus den verschiedenen Speichern ein Ziel auswählt, können hier leicht Probleme entstehen, falls die Datenquelle nicht mit dem Format des Speichers kompatibel ist. Außerdem sind die Verarbeitungsweisen der Quellen zu den Speichern fest im Code des Servers festgehalten. So ist es nicht möglich während der Laufzeit neue Datenquellen zu integrieren.

1.2.6 Change Data Capture (CDC)

Um Änderungen an Daten in Datenquellen in das Data-Lake-System einpflegen zu können, müssen diese erst erfasst werden. Diesen Prozess nennt man Change Data Capture (CDC). Das Ziel beim CDC ist es, die Änderungen an den Daten nur an einer Stelle zu erfassen und dann an an-



(a) Datenbank-Ingestion (b) Datei-Ingestion

Abbildung 1.2: Ingestion-Verarbeitung des Prototypen, Quelle: *Development of a Data Lake System*, S. 3

dere Systeme weiter zu schicken. Dafür gibt es verschiedene Ansätze, die in den Arbeiten „Delta view generation for incremental loading of large dimensions in a data warehouse“(Mekterović 2015), „Change data capture in NoSQL databases: A functional and performance comparison“(Schmidt u. a. 2015) und „Extracting delta for incremental data warehouse maintenance“(Ram 2000) erläutert werden.

Datenbank-Trigger basiert

Datenbank-Trigger sind Events, die bei verschiedenen Aktionen auf den Daten in einer Datenbank ausgelöst werden. Über diese Trigger lassen sich CDC-Programme realisieren, die Änderungen genau dann festhalten, wenn sie geschehen. Ein Nachteil ist, dass die Methode nur in System angewendet werden kann, die auch Trigger unterstützen. Dafür ist es möglich alle Änderungen wie Einfügen, Aktualisieren oder Löschen von Daten zu erfassen (ebd.).

Log basiert

Es gibt viele Datenspeicher-Systeme, die Logs über die Aktionen auf den Daten führen. Diese werden zum Beispiel genutzt um eine Wiederherstellung möglich zu machen. Ein CDC-Programm kann diese Logs auslesen und daraus die Änderungsdaten erzeugen. Hierdurch gibt es fast keinen extra Aufwand für das eigentliche System. Aber auch hier gilt, dass diese Methode davon abhängig ist ob ein System Logs erstellt und ob diese durch externe Programme abgerufen werden können (Mekterović 2015).

Zeitstempel basiert

Ein weiterer Ansatz ist die Verwendung von Zeitstempeln mit den Zeitpunkten der Erstellung und letzten Änderung. Diese Zeitstempel müssen jedem Datensatz vorhanden sein. Die Verantwortung dafür kann entweder bei dem Ersteller der Daten liegen oder durch das Speichersystem automatisch hinzugefügt werden. Das CDC-Programm überprüft regelmäßig alle Zeitstempel der Einträge in den Daten. Wenn diese zwischen dem letzten und dem aktuellen Durchlauf liegen wird die Änderung erfasst. Hierbei werden nur kumulierte Änderungen seit dem letzten Durchlauf erfasst. Es ist nicht möglich nachzuvollziehen, welche und wie viele Änderungen in der Zeit gemacht wurden. Außerdem lassen sich auch mit dieser Methode kein Löschungen erfassen (ebd.). Der Aufwand für diese Methode kann relativ hoch werden, da ohne Indices auf den Zeitstempeln immer die gesamten Daten gelesen werden müssen Ram 2000.

Snapshot basiert

Die letzte Methode ist das Vergleichen zweier Momentaufnahmen (Snapshots) eines Datensatzes. Dabei wird bei jedem Durchlauf zuerst ein aktueller Snapshot generiert. Dieser wird danach mit dem des vorherigen Durchlaufs verglichen, um alle Änderungen zu erhalten. Hierfür muss ein

separater Speicherort für die Snapshots festgelegt werde. Wie bei den Zeitstempeln ist es nicht möglich den gesamten Änderungsverlauf zwischen zwei Snapshots nachzuvollziehen. Außerdem müssen für den Vergleich immer alle Daten geladen werden, was zu einem hohen Rechen- und Speicheraufwand führen kann Schmidt u. a. 2015.

Kapitel 2

Anforderungen

Die Ingestion soll Daten aus unterschiedlichsten Quellen aufnehmen können. Für die aufgenommen Daten soll das System bereits Speicher bereitstellen, die standardmäßig verwendet werden können. Um jedoch flexibel zu bleiben muss auch die Option gegeben werden weiteres Speichersystem anzubinden. Es reicht aus, wenn die optionale Versionierung der Daten, durch das Data-Lake-System, nur auf dem internen Speicher gegeben ist. Änderungen am Verhalten der Ingestion in kleinerem Rahmen sollen ohne Anpassungen des Server-Codes möglich sein.

Wie bereits erwähnt, ist der Data-Lake-Prototyp eine monolithische Anwendung. Das bedeutet, dass die gesamte Anwendung als Komplettlösung in einem Programm entwickelt und bereitgestellt wird. Solche Ansätze sind Anfangs leichter umzusetzen, haben aber größere Nachteile in Bereichen wie Fehlertoleranz und Wartbarkeit. Daher soll für die Ingestion-Schnittstelle der Microservice-Ansatz verfolgt werden. Hierbei werden die Funktionalitäten und Aufgaben auf mehrere kleinere Anwendungen aufgeteilt. Das hat den, wie von Gos und Zabierowski (2020) dargestellt, mehrere Vorteile. Die Wartung fällt bei mehreren kleinen Programmen leichter, da sie übersichtlicher und verständlicher sind. Bei Fehlfunktionen einzelner Mircoservices fällt außerdem nicht die komplette Anwendung aus, sondern nur die Funktion, für die der Service zuständig war.

Zuletzt ist es einfacher bestimmte Aspekte der Software zu skalieren und bei Updates bleibt eine höhere Verfügbarkeit, da nur ein kleiner Teil des Systems neu gestartet werden muss.

Aus den oben genannten Zielen lassen sich jetzt genauere Anforderungen entwickeln, die die Ingestion-Schnittstelle erfüllen soll. Dazu werden nachfolgend die einzelnen Ziele in Abschnitte aufgeteilt und die dazugehörigen Anforderungen festgehalten. In der Evaluierung kann dann überprüft werden, ob alle Anforderung durch das Ergebnis erfüllt werden. Die Unterkapitel sind so aufgebaut, dass erst eine genauere Erklärung für das Ziel gegeben wird und dann in einzelnen Paragraphen die Anforderungen nummeriert aufgelistet werden.

2.1 Quellen- und Formatunabhänigkeit

Es soll die Möglichkeit gegeben werden, Daten aus jeder beliebigen Quelle in das System zu integrieren. Dazu muss die Schnittstelle sowohl in der Lage sein direkt Daten entgegen zu nehmen als auch aus anderen Systemen zu extrahieren. Unter System wird hierbei jedoch nicht nur eine Datenbank verstanden, sonder es können unter anderem auch Dateien, APIs oder Datenströme gemeint sein. Ebenso soll es möglich sein, den Speicher im Data-Lake-System für die Daten auszuwählen. Da im Prototyp Apache Spark verwendet wird, ist bereits die Möglichkeit geben verschiedenste Formate zu verarbeiten.

ANF_01 Die Schnittstelle muss in der Lage sein Quelldaten entgegen zu nehmen, die an das Data-Lake-System gesendet werden. Diese müssen so verwaltet werden, dass sie über Apache Spark gelesen werden können.

ANF_02 Da Apache Spark nicht von sich aus in der Lage ist, alle Datenformate zu verstehen, muss es möglich sein die SparkSession mit benö-

tigten Paketen zu erweitern.

ANF_03 Für die Unterstützung verschiedenster Quell- und Zielsysteme verwendet Apache Spark zum Lesen und Speichern von Dateien eine Format-Parameter und Optionen. Diese sollen komplett konfigurierbar sein um alle Systeme verwenden zu können.

ANF_04 Einige Funktionalitäten, wie zum Beispiel das Ausführen einer Reihenfolge von Abfragen an eine Programmierschnittstelle können nicht durch Apache Spark abgedeckt werden. Daher soll es eine Möglichkeit geben der Ingestion-Schnittstelle eigenen Programmcode mit zu geben, der diese Funktionen abdeckt.

2.2 Kontinuierliches Laden

Da Daten sich mit der Zeit ändern, soll die Ingestion-Schnittstelle in der Lage sein, neue Daten aus einer Datenquelle, die bereits aufgenommen wurde, erneut zu laden. Die Implementierung sollen das erneute Anstoßen, eine Zeitsteuerung oder Datenströme zulassen.

ANF_05 Es soll möglich sein, Datenströme in das Data-Lake-System zu integrieren und als Quelle für kontinuierliche Daten zu verwenden.

ANF_06 Um aktuelle Daten aus Datenquellen, die nicht über einen Datenstrom verfügen, zu integrieren, soll es eine zeitgesteuerte wiederholte Ausführung geben.

ANF_07 Es wird ein API-Endpunkt benötigt, über den die Ingestion für eine bestimmte Datenquelle angestoßen werden kann. Dieser soll auch

dazu verwendet werden, externe Systeme, wie eigene CDC-Lösungen anzubinden.

ANF_08 Eine gleichzeitige Ausführung mehrerer Ingestions auf der gleichen Datenquelle könnte leicht zu Konflikten in den Daten führen. Daher soll sichergestellt werden, dass das System genau diesen Fall nicht zulässt.

2.3 Datenversionierung

Durch das kontinuierliche Laden von Daten, entstehen laufend neue Versionen eines Datensatzes einer Datenquelle. Diese Veränderungen der Daten können in vielen Anwendungsfällen bei der Auswertung von Interesse sein. Dabei gibt es zwei verschiedene Wege, diese Daten in das System ein zu pflegen. Einmal die Verwendung von CDC-Lösungen, die bereits in den Daten die Informationen über Änderungen enthalten und zweitens kann einfach der aktuelle Stand eines Datensatzes erneut geladen werden.

ANF_09 Daher soll das System eine Möglichkeit bieten das Einfügen, Aktualisieren oder Löschen von Daten festzuhalten und zur Abfrage zur Verfügung zu stellen. Außerdem sollen damit auch die Daten zu bestimmten Zeitpunkten rekonstruierbar sein.

ANF_10 Um für alle eingehenden Daten die Möglichkeit der Versionierung im System anbieten zu können, muss eine CDC-Implementierung eingebaut werden, die für jede Datenquelle ausgeführt werden kann.

ANF_11 Auch die Verwendung einer eigenen CDC-Lösung für eine Datenquelle soll unterstützt werden. Dazu muss eine Quelle mit Änderungs-

daten für eine bereits aufgenommen Datenquelle erstellt werden können.

2.4 Architektur

Wie bereits erwähnt, soll die Ingestion in einer Mircoservice-Architektur umgesetzt werden. Außerdem wird eine Schnittstelle für die Interaktion mit dem System benötigt. Daraus ergeben sich folgende Anforderungen, an die Architektur.

ANF_12 Die Interaktions-Schnittstelle mit dem System soll eine REST-API sein, die keine Konflikte mit dem aktuellen Prototypen erzeugt.

ANF_13 Die Aufgaben müssen klar getrennt auf die einzelnen Mircoservices aufgeteilt werden. Die Überschneidungen zwischen den Mircoservices sollten so gering wie möglich gehalten werden.

ANF_14 Für die Kommunikation zwischen den Microservices soll eine einheitliche Lösung entwickelt werden. Diese soll es auch ermöglichen neue Mircoservices einfach in die Architektur einzubringen.

Kapitel 3

Entwurf

Nach Zhao, Megdiche und Ravat (2021) ist der Ingestion-Prozess eines Data-Lake-Systems, als erster Schritt im Lebenszyklus der Daten, maßgebend dafür, wie gut die Daten später verwendet und verarbeitet werden können. Hier spielen besonders die Metadaten eine große Rolle. Da dieser Bereich jedoch nicht in den Rahmen dieser Arbeit passt, wird hier nur die Aufnahme und Integration der Daten behandelt. Hier durch fällt bereits ein Minimum an Metadaten an. Genau die Daten, die benötigt werden um eine Ingestion auszuführen und die, die bei der Versionierung erzeugt werden.

Da ein wichtiges Prinzip bei Data Lakes das Speichern der Rohdaten ist, diese aber nicht immer passend für die weiter Verarbeitung sind, kann ein ähnliches System wie von Rooney u. a. (2019) beschireben verwendet werden. Die dortige Aufteilung in die drei Zonen DropZone, LandingZone und IntegrationZone wurde für einen speziellen Fall entwickelt und passt nicht in das generelle Data-Lake-System. Aber die Aufteilung der Zugriffe und Ändeurngen an Daten in Bereiche ist auch hier anwendbar. Der erste Bereich in dem Daten abgelegt werden ist der Ingestion-Bereich. Die Datensätze, die hier gespeichert werden repräsentieren einen bestimmte Datenquelle. Nur die Ingestion-Schnittstelle sollte Daten in diesem Bereich erstellen oder verändern. Ein Metadaten-Management-System kann

diese Daten einlesen um direkt nach der Ingestion weitere Metadaten zu erstellen.

3.1 Architektur

Zu Anfang ist es sinnvoll, die Architektur des Systems festzulegen. Diese bestimmt, welche Komponenten und Microservices entwickelt und verbunden werden müssen. Der erste Schritt dabei ist es, die Aufgaben aufzuteilen, die das System erfüllen soll und dann auf Microservices verteilt werden. Die weitere Entwicklung des Systems stützt sich dann auf die fertige Architektur.

Eine erste und offensichtliche Aufteilung gibt bei den benötigten Arbeitsschritten Laden der Daten, Deltaberechnung und Speichern der Daten. Da diese aber ein zusammenhängender Prozess sind, der in ein Job in Spark ausgeführt wird, sollte hier auch keine Aufteilung geschehen. Besser ist die Betrachtung verschiedener Funktionen. Hier findet man die API, das kontinuierliche Ausführen und die Ingestion mit Spark der Daten als gut trennbare Teile.

Bei der **API** handelt es sich um den Service für die Interaktion mit dem Data-Lake-System. Durch die Anforderungen ist bereits festgelegt, dass dieser ein Web-Server mit einer REST-Schnittstelle ist. Es geht zwar in dieser Arbeit nur um die Ingestion, aber der API-Service sollte Schnittstellen zu allen Funktionen des Data-Lake-Systems enthalten.

Die **Ingestion** ist dafür zuständig, die Datenquellen zu verarbeiten und den kompletten Prozess vom Laden bis zum Speichern der Daten in *Apache Spark* auszuführen. Die Ausführung soll für eine Datenquelle nur einmal gleichzeitig aber parallel für unterschiedliche laufen.

Bei einer zeitgesteuerten oder Datenstrom-Ingestion muss die **kontinuierliche Ausführung** sichergestellt werden. Für alle Datenquellen muss regelmäßig geprüft werden, ob für diese gerade eine Ingestion aus-

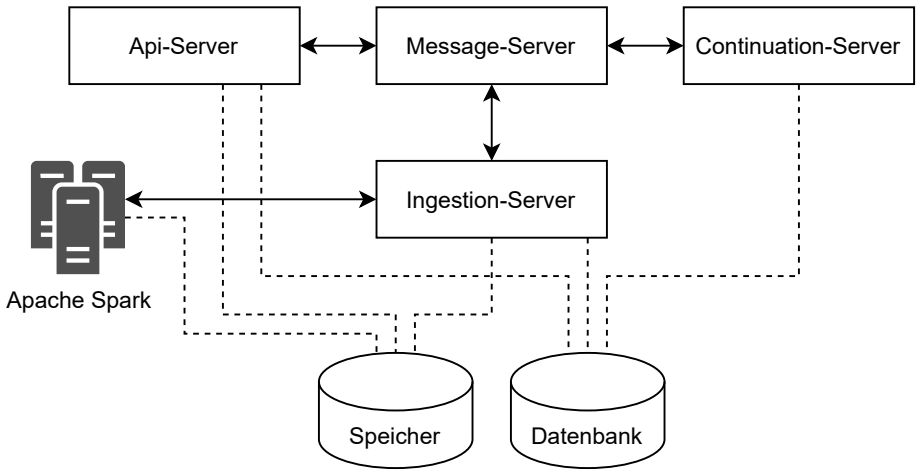


Abbildung 3.1: System Architektur der Ingestion

geführt wird und ausgeführt werden sollten. Falls keine ausgeführt wird aber sollte, wird die Ingestion für diese Datenquelle gestartet.

Neben diesen Mircoservices wird noch ein Nachrichtensystem benötigt. Das Nachrichtensystem stellt die Kommunikation zwischen den Microservices dar. Hier ist es wichtig, dass es einem Sender möglich ist, Nachrichten an einen oder auch an mehrere Empfänger zu senden. So soll sichergestellt werden, dass bestehende Microservices einfach repliziert und neue eingefügt werden können.

Es werden zwei Speicher benötigt. Eine Datenbank, die ausschließlich prozeduale Daten enthält. Das sind alle Daten, die für die Funktion des Systems notwenig sind. Und den Speicher für die Ablage aller aufgenommen Daten.

3.2 Plugins

In 2.1 wird durch ANF_04 gefordert, dass zusätzlicher Code bei der Ingestion ausgeführt werden können soll. Das soll durch Plugins umgesetzt werden. Plugins werden jeder Datenquelle einzeln hinzugefügt und an

verschiedenen, fest definierten Punkten ausgeführt. Da die Plugins eventuell auf Software-Bibliotheken zurückgreifen müssen, die nicht auf dem Data-Lake-System vorhanden sind, kann zusätzlich eine Liste von Abhängigkeiten der Plugins angegeben werden. Das Prinzip der Plugins kann auch über die Ingestion hinaus im System angewendet werden.

Für die Ingestion gibt es zwei Stellen an denen die Möglichkeit für Plugins gegeben sein sollte. Die erste ist wie durch die Anforderung gefordert das Laden von Daten. Genauer bedeutet das, dass das Plugin die Aufgabe übernimmt eine DataFrame zu erstellen, dass später wieder gespeichert wird. Das Plugin erstzt die normale Funktion zum Laden in ein DataFrame.

Als zweite sollte es Plugins geben, mit denen man nach dem Laden das DataFrame manipulieren kann. Das streng gesehen gegen das Prinzip eines Data Lakes, Daten unverändert in ihrem Rohzustand zu speichern. Aber zum Beispiel bei der Anbindung von Kafka Datenströmen, werden die Daten nur als Bytestrom übertragen. Hier sollte es schon möglich sein, diese Bytes wieder in das Format zu bringen, dass sie vor dem Versenden hatten.

3.3 Datenmodelle

3.3.1 DatasourceDefinition

Für das Data-Lake-System muss eine Modellierung entwickelt werden, mit der alle Datenquellen dargestellt werden können. Das entwickelt Modell definiert eine Datenquelle und bekommt den Namne DatasourceDefinition (deutsch: Datenquellendefinition). Da Apache Spark verwendet wird, ist einer der wichtigsten Punkte zur definition einer Datenquelle die Erfassung der Informationen, die in Spark benötigt werden. Dazu gehören:

- extra Abhängigkeiten für Spark (spark.jars.packages)
- das Format und Optionen für die Reader
- und bei benutzerdefinierten Speichern das Format, die Optionen und einen Schreibmodus für die Writer

Neben den Spark spezfischen werden noch folgende weiter Informationen erfasst:

- ein Name für die Datenquelle
- die Id einer anderen Datenquelle, falls die aktuelle eine Updatequelle ist
- Datum der Erstellung und letzten Änderung
- den Namen der Id-Spalte in den Daten (wird später für die Deltaberechnung benötigt)
- den Typ, der zu lesenden Daten
- bei Datei-Ingestions eine Liste der zu laden Dateien
- den Typ, wie die Daten geschrieben werden müssen
- eine Liste mit der Zeitsteuerung für eine kontinuierliche Ingestion
- eine Liste mit Plugindateien
- und eine Liste mit Abhängigkeiten der Plugins

Die Lese-Typen ergeben sich aus der Betrachtung, wie die Daten in das Data-Lake-System gelangen und welche Struktur sie haben. Bei einer Pull-Ingestion ist das System dafür verantwortlich Daten aus einer Quelle zu laden. Dies ist zum Beispiel bei Datenbanken der Fall. Das Gegenteil dazu ist eine Push-Ingestion, bei der die Daten direkt an den Data Lake

gesendet werden. Diese muss jedoch nochmal in zwei unterschiedliche Typen unterteilt werden. Bei einer Stream-Ingestion, also bei Datenströmen, werden kontinuierlich neue Daten an das System gesendet werden. Und bei einer File-Ingestion werden Dateien hochgeladen, die die Daten enthalten, wobei wichtig ist, dass alle Dateien das gleiche Dateiformat haben. Die Dateien können sich noch einmal und Daten- und Quelldatein unterscheiden. Daten-Dateien enthalten unstrukturierte Daten und werden einfach im HDFS mit abgelegt ohne weiter verarbeitet zu werden. Das könnten zum Beispiel Bilder oder Videos sein. Quelldateien enthalten mindestens semistrukturierte Daten und dienen dem Zweck, in ein anderes Speicherformat, wie zum Beispiel Parquet oder eine Delta Tabelle geschrieben zu werden. Es ist nicht möglich Daten direkt an die API zu senden. Alle Push-Ingestions sollen über diese beiden Typen abgebildet werden.

Die Schreib-Typen kommen von den drei Optionen wo Daten abgelegt werden können. Custom bedeutet, dass der in der DatasourceDefinition konfigurierte Speicher verwendet werden soll. Delta ist das Speichern im internen Speicher aber mit einer Versionierung und Default ist das Speichern ohne Versionierung.

Für die Umsetzung einer unkomplizierten Versionierung werden alle veränderlichen Informationen einer DatasourceDefinition in Revisionen gespeichert. Das betrifft alle oben genannten Felder. Die Revisionen einer DatasourceDefinition erhalten eine fortlaufende Nummer. Die DatasourceDefinition selbst hält dann nur noch eine Liste aller Revisionen und die Nummer der aktuellen.

3.3.2 IngestionEvent

Auch für den Zustand einer Ingestion musss ein Modell entwickelt werden. Die Auführungen der Ingestions werden hier IngestionEvent geannant. Dieses enthält wie die Revision eine fortlaufende Nummer, ein Start-

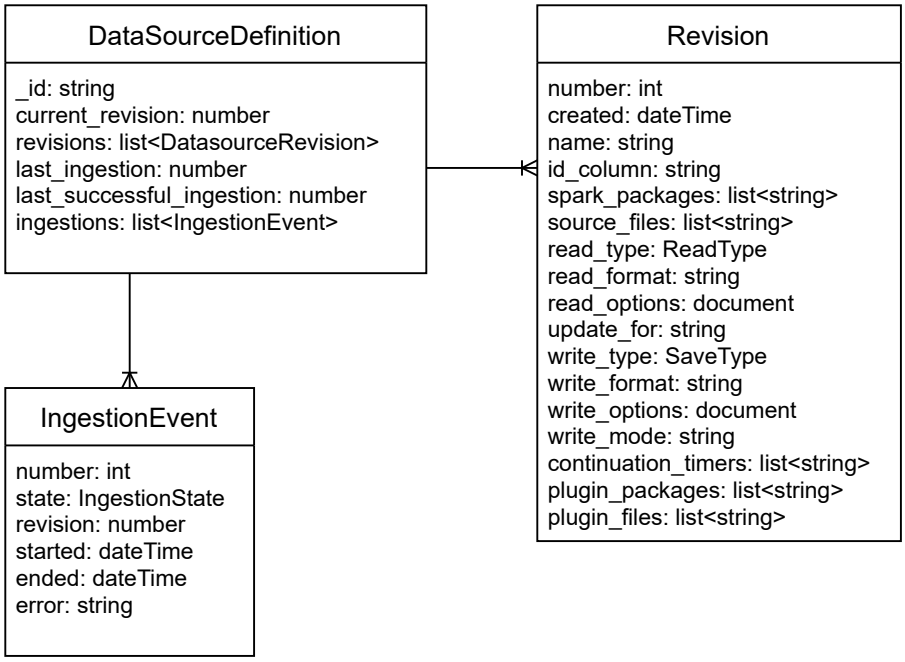


Abbildung 3.2: Übersicht Datenmodell

und Enddatum, den aktuellen Status indem es befindet, die Nummer der Revision mit der es gestartet wurde und eine Fehlernachricht.

Das IngestionEvent kann auch an die DatasourceDefinition mit angehangen werden. Dafür werden Felder hinzugefügt, die alle IngestionEvents, die Nummer des letzten und die Nummer des letzten erfolgreichen IngestionEvents enthalten.

3.4 API-Service

Der API-Service benötigt nicht viel Entwicklung. Es müssen lediglich die Enspunkte definiert werden, die Zugriff auf die verschiedenen Funktionen der Ingestion-Schnittstelle bereitstellen. Dazu gehören die Verwaltung von Datenquellen und das Starten einer Ingestion. Da es sich um eine REST-Schnittstelle handelt, werden die Endpunkte hier durch einen Pfad

und eine HTTP-Methode definiert (Tabelle 3.1). Außerdem kümmert sich der API-Service um die Erstellung von DatasourceDefinitions und deren Revisionen und IngestionEvents. Bei Anfragen zum Starten einer Ingestion versendet der API-Server eine Nachricht mit der Id einer Definition der zu ladenen Datenquelle.

GET /datasources Liefert alle im System gespeicherten Datenquellen
GET /datasources/<id> Liefert die Datenquelle mit der im Pfad übergebenen Id
POST /datasources Bearbeitet die Daten Datenquelle mit der im Pfad übergebenen Id
PUT /datasources/<id> Erstellt eine neue Datenquelle
GET /datasources/<id>/run Startet eine Ingestion der Datenquelle mit der im Pfad übergebenen Id

Tabelle 3.1: Endpunkte des API-Servers

3.5 Continuation-Service

Für die Sicherstellung der korrekten Ausführung kontinuierlicher Ingestions, müssen regelmäßig alle Datenquellen überprüft werden. Dabei gibt es zwei Bedingungen nach denen entscheiden wird, ob eine Ingestion werden muss. Bei Datenströmen gilt allgemein, wenn dieser nicht läuft, dann muss die Ingestion automatisch neu gestartet werden. Eine

Ausnahme dabei ist, wenn die Verarbeitung durch den Benutzer explizit beendet wurde.

Der zweite Fall ist die Zeitsteuerung. Für eine zeitgesteuerte kontinuierliche Ingestion werden einer Datenquelle ein oder mehrere Timer hinzugefügt. Der Continuation-Service prüft, ob der Timer zu diesem Zeitpunkt zutrifft oder nicht. Wenn das der Fall ist und bisher keine Ingestion auf der Datenquelle läuft, dann wird eine neue gestartet.

In Unix-Systemen gibt es bereits eine Lösung für die Notation solcher Timer. Dort gibt es die sogenannten Cron-Jobs, mit deren Hilfe Aufgaben automatisch und regelmäßig ausgeführt werden können. Dabei wird der Zeitpunkt der Ausführung über fünf Felder festgelegt. Diese geben die Minute, die Stunde, den Tag des Monats, den Monat und den Tag der Woche als Zahlen an. Als Erweiterung kann man „*“ als Platzhalter für alle möglichen Werte verwenden, man kann mehrere Werte als mit Kommata getrennt angeben oder mit „/x“ eine Liste in Schritten der Größe x erzeugen (*crontab(5) — Linux manual page* 2012).

Diese Notation soll auch für die Zeitsteuerung der Ingestions genutzt werden. Als Referenz wird dabei die koordinierte Weltzeit (UTC) genommen, damit die Ausführung unabhängig vom Standort einheitlich bleibt. Wenn eine Datenquelle mehrere Timer hat, reicht es aus, dass einer von diesen zutrifft.

3.6 Ingestion-Service

Der Ingestion-Service hat die Aufgabe eine Ingestion für eine Datenquelle durchzuführen. Dazu gehören das Laden der Daten in ein DataFrame, die Deltaberechnung und das Speichern. Das meiste davon wird jedoch nicht von dem Service selbst, sondern auf dem Spark Cluster gemacht. Der Service führt die Vorbereitung und das Deplyoment des Jobs auf dem Cluster aus.

Der Ingestion-Service wartet auf die Nachricht zur Ausführung einer Ingestion, mit der Id der DatasourceDefinition. Als erstes wird dann geprüft, ob bereits eine Ingestion der Datenquelle aktiv ist. Falls das nicht der Fall ist, wird ein neuer Prozess gestartet, indem die Ingestion ausgeführt wird. Auf diese Art können mehrer Ingestion von verschiedenen Quellen parallel bearbeitet werden.

Der Ablauf einer einzelnen Ingestion kann unabhängig vom Lese- und Schreib-Typ in einem allgemeinen Ablauf abgebildet werden. Als erstes wird die Ingestion vorbereitet. Hier werden die Plugins installiert und eine SparkSession erstellt. Im nächsten Schritt werden die Daten aus der Quelle geladen. Wenn es sich dabei um Ändeurngsdaten aus einer Updatequelle handelt, können diese direkt in den entsprechenden Zieldatensatz eingepflegt werden. Ist das nicht der Fall, foglt eine Entscheidung, ob Änderungsdaten berechnet werden müssen. Es gelten die folgenden zwei Regeln:

- der Speicher-Typ ist Delta
- es ist nicht die erste Ingestion dieser Datenquelle

Wurden Änderungsdaten berechnet, werden diese eingepflegt und ansonsten einfach gespeichert. Handelt es sich nicht um einen benutzerdefinierten Speicher, werden die alten Daten mit den neuen überschrieben.

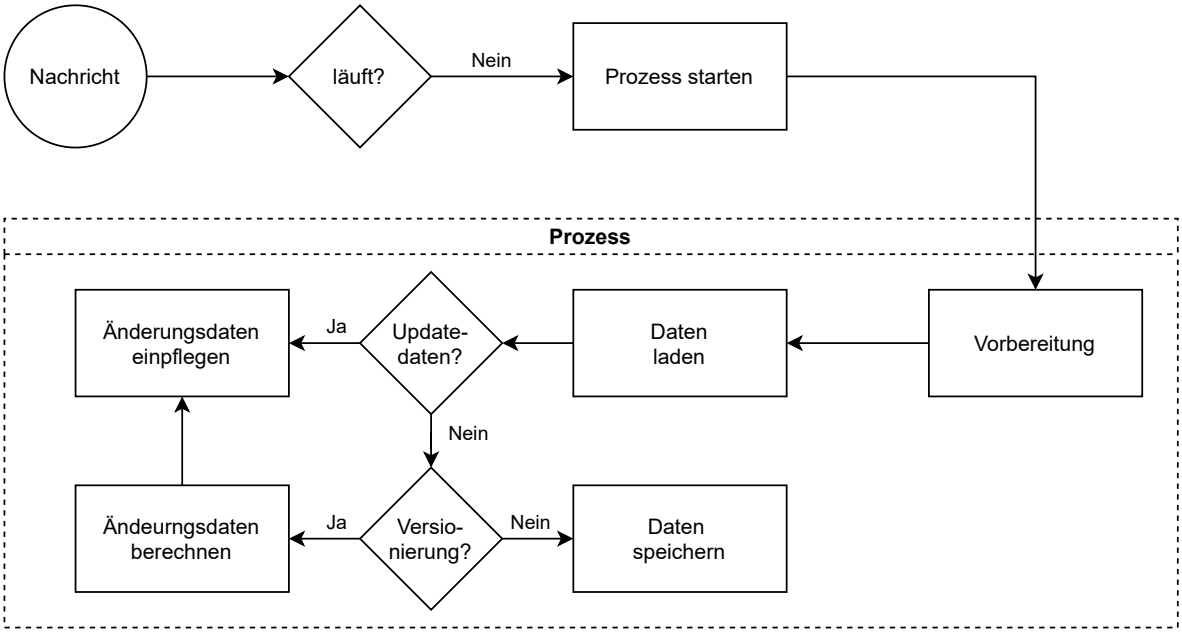


Abbildung 3.3: Ablauf einer Ingestion

Kapitel 4

Implementierung

4.1 Programmiersprache

Durch die Verwendung von Apache Spark ist die Auswahl der Programmiersprachen auf Java, Python und Scala eingegrenzt. Da der Prototyp in Python geschrieben wurde und dieser zum Api-Service erweitert werden soll, ist Python hierfür die Wahl. Auch für die Implementierung des Ingestion-Service bietet Python einige Vorteile.

Bei der Verwendung von Java und Scala wird ein fertig kompiliert mit dem Befehl „spark-submit“ zur Aufügrung an Spark gesendet. Das bedeutet, dass der Code dieser Jobs bereits feststehen muss. Neben dieser Option gibt es in Python auch die Möglichkeit, dass sich der Interpreter zur Laufzeit um die korrekte Ausführung der Jobs kümmert. So ist es möglich ebenfalls für jede Anfrage speziell konifugrierte Jobs zu erstellen, die nicht vorher schon kompiliert sein müssen. Das senkt die Komplexität bei der Entwicklung der Ingestion (Apache Spark 2021b).

Auch für die Plugins hat Python einen Vorteil. Man kann dynamisch Programmcode aus Dateien laden und inspizieren. So können für jede Ausführung die Plugins einer Datenquelle frisch geladen werden. Es muss nur dafür gesorgt werden, dass auch die Abhängigkeiten alle erfüllt sind.

Um die Entwicklung einheitlich zu halten, wird auch der Continuation-Service in Python implementiert.

4.2 Nachrichtensystem

Für die Übermittlung von Nachrichten zwischen verschiedenen Anwendungen gibt es sogenannte Message-Broker. Diese koordinieren als Mittelsmann die Verteilung der Nachrichten an verschiedene Empfänger. Das hat den Vorteil, dass der Sender unabhängig von den Empfängern wird und die Kommunikation asynchron statt finden kann (Apukhtin, Shirokopetleva und Skovorodnikova 2019).

Es gibt mittlerweile einige Projekte, diese Aufgabe auf verschiedene Arten lösen. Hier wird dafür Apache Kafka verwendet, welches im Big Data Bereich weit verbreitet ist um Datenströme zu verarbeiten. Daher macht es Sinn, dieses in das Data-Lake-System zu integrieren und darin bereit zu stellen. Um das System dabei nicht unnötig komplex und zu groß werden zu lassen, wird daher auf einen anderen Message-Broker verzichtet.

Da Kafka ein Event-Streaming-System ist, wird ab hier nicht mehr vom Austausch von Nachtrichten, sondern von Events gesprochen. Für diese Events müssen Topics zur Einordnung fetsgelegt werden. Die Schlüssel sollten dabei so gewählt werden, dass Kafka auch für andere Datenströme verwendet werden kann, ohne, dass zu Konflikten kommt. Daher werden die Topics der internen Kommunikation des Data-Lake-Systems immer mit „dls__“ als Prefix benannt werden. Danach folgt der Bereich, den das Event betrifft, hier zum Beispiel „ingestion“. An diesen Namen kann dann noch weiter Unterscheidung angehängt werden. Für das Ausführen einer Ingestion wäre damit die Topic „dls__ingestion__run“.

Wenn mehrere Consumer in einer Gruppe für eine Topic sind, werden Events nicht an alle sondern immer nur an einen aus der Gruppe gesen-

det. Dieser Mechanismus kann für den Lastausgleich an bestimmten Stellen verwendet werden. Für die Ingestion kann so der Ingestion-Services einfach repliziert werden.

4.3 Datenbank und Datenmodell

Da das Datenmodell einer Datenquelle eine verschachtelte Struktur hat, bietet sich hier als einfachste Lösung die Verwendung einer dokumentorientierten NoSQl-Datenbank an. Das hat den Vorteil, dass diese Listen direkt in den Objekten der Datenquellen abgelegt werden können. In relationalen Datenbanken, die Tabellen verwenden, müsste man für jedes Modell eine eigene Tabelle erstellen und die Verknüpfungen über über JOIN-Operationen auflösen. Bei jeder Abfrage einer Datenquelle werden die verknüpften Einträge der Ingestion-Events oder Revisionen gebraucht, was somit zu einem größeren Aufwand führt. Außerdem gibt es viele Anfragen auf die Datenquellen, da diese nicht zwischen den Mircoservices ausgetauscht werden und so nicht im Speicher vom Service verwaltet werden können. Daher ist es effizienter die relevanten Daten direkt mit einer Abfrage laden zu können. Hier kommt MongoDB[1] als Datenbank zum Einsatz. MongoDB kann frei verwendet werden und bei bei größeren Datenmengen verteilt eingesetzt werden.

4.4 Interner Datenspeicher

Die Entwicklung einer eigenen Lösung zum Speichern von Daten mit Versionierung passt nicht in den zeitlichen Rahmen dieser Arbeit. Daher wird der Delta Lake verwendet, da dieser alle benötigten Funktionenfür die Versionierung bietet und eine voll unterstützte Schnittstelle zu Spark hat. Da das System auf eigenen Server laufen soll, wird als Speicher für den

[1] https://www.mongodb.com/

Delta Lake ein HDFS Cluster verwendet. Auch die Quelldateien der Ingestion und die Plugins können im HDFS abgelegt werden und sind damit für alle Microservices abrufbar. Der Zugriff zum Speichern der Daten geschieht über den Delta Lake beziehungsweise Spark und für die Quelldateien und Plugins über die WebHDFS REST-Schnittstelle.

Neben den versionierten müssen auch Daten mit und ohne Struktur gespeichert werden. Auch hierfür wird das HDFS verwendet. Strukturierte und semistrukturierte Daten können im Parquet Format abgelegt werden. Das ist das gleiche Format, das auch der Delta Lake verwendet. Werden so wie sie hochgeladen wurden im HDFS abgelegt.

Die Ordner werden folgendermaßen organisiert. Für alle Daten wird im Root-Verzeichniss des HDFS ein Ordner "datalakeängelegt. Dieser enthält die Unterordner für die Hochgeladenen Quell-Dateien „sources“, die Plugins „plugins“ und die geladenen Daten „data“. In den Ordnern „sources“ und „plugins’"werden dann für jede Datenquelle Ordner mit deren Id angelegt, in denen die hochgeladenen Dateien abgelegt werden. Für die geladenen Daten wird nocheinmal und die Ordner „structured“ für Daten mit Struktur, "´unstructured’für unstrukturierte Daten und „delta’für die Delta Tabellen angelegt. Diese unterteilen sich dann ebenfalls wieder in Ordner für jede Datenquelle mit der Id als Name.

4.5 Api-Service

Daten werden an den API-Service im Body einer Anfrage im form-data Format übergeben. So kann man neben den Informationen zu der Datenquelle in der gleich Anfrage auch weitere Dateien mit hochladen. Dabei werden alle Daten als Schlüssel-Wert-Paar übertragen. Für die DatasourceDefinition muss der Schlüssel „datasource-definition“ verwendet werden. Die Schlüssel der hochgeladenen Dateien sind frei wählbar. Die Eingabe der DatasourceDefinition wird im JSON Format übertragen. Abbil-

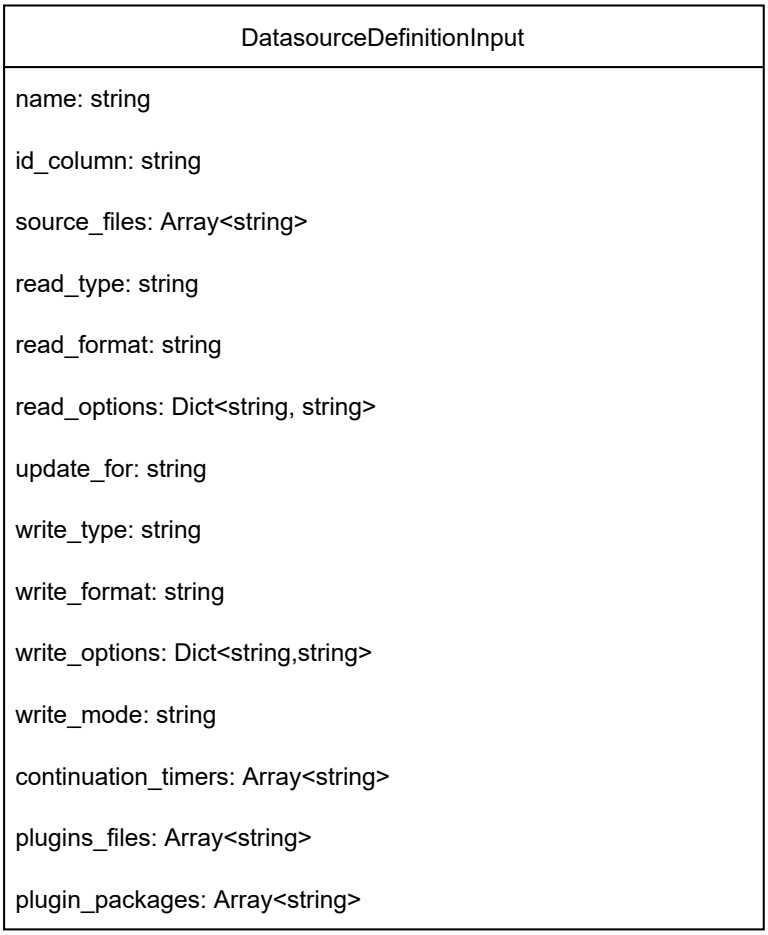


Abbildung 4.1: Felder Datenquellen-Eingabe

dung 4.1 zeigt das dazugehörige Model.

4.5.1 Hochladen von Dateien

Für jede Datenquelle können Quell- oder Plugindateien hochgeladen werden. Ein Eintrag in der Liste der Dateien kann entweder ein Dateiname oder der Schlüssel einer Datei sein, die in der Anfrage mitgesendet wird. Bei der verarbeitung der Eingabe prüft der Api-Service für jeden Eintrag der Listen, ob eine Datei mit diesem Schlüssel hochgeladen wurde. Ist

das der Fall, wird die entsprechende Datei in das HDFS hochgeladen und der Name der Datei der neuen Revision angehangen. Wenn keine Datei in der Anfrage gefunden wurde, wird geschaut ob im HDFS eine Datei mit dem Namen existiert. Wird eine Datei gefunden, wird der Name an die neue Revision angehangen. Sonst wird dieser Eintrag ignoriert.

4.6 Continuation-Service

Der Continuation-Service ist ein einzelner Prozess. In einer Schleife werden erst alle Datenquellen überprüft, die eine Zeitsteuerung enthalten. Hier wird eine Ingestion gestartet, wenn der Status des letzten IngestionEvents „STOPPED“ oder „FINISHED“ ist. Im zweiten Schritt werden alle Datenquellen mit Datenströmen geprüft. Datenströme werden nur bei einem Status von „FINSIHED“ automtaisch gestartet, da „STOPPED“ bedeutet, dass die Ausführung beabsichtigt beendet wurde. Dabei wird die Schleife maximal einmal jede Minute durchlaufen, da über die Zeitsteuerung keine schneller Ausführung möglich ist. In Algorithmus 1 ist die Logik eines Durchlaufes zu sehen.

4.7 Ingestion-Service

Wie in Abschnitt 3.6auf Seite 29 beschireben, wird für ein Event ein Prozess gestartet. Der Name der Topic des Events zum Starten einer Inegstion ist „dls__ingestion__run“. In Abschnitt 4.7.3auf Seite 43 wird der detailitere Ablauf mit den verschiedenen Wegen für Read- und SaveTypes beschrieben. Vorher werden die benötigt Grundlagen zur Ausführung erleutert.

Algorithmus 1: Continuation loop

```
loopStart ← now
definitions ← query DatasourceDefinitions with continuation
 timers
forall definition in definitions do
    timers ← timers of source
    if state is STOPPED or FINISHED then
        forall timer in timers do
            if timer is now then
                send run event
                end check for this DatasourceDefinition
            end
        end
    end
end
definitions ← query DatasourceDefinitions with type STREAM
 forall definition in definitions do
    if state is FINISHED then
        send run event
    end
end
looEnd ← now
loopDuration ← loopEnd - loopStart
if loopDuration < 60 sec then
    sleep for 60 sec - loopDuration
end
```

4.7.1 Berechnen und Speichern der Änderungsdaten

Um Änderungen in eine Delta Tabelle zu übernehemn, müssen die Änderungsdaten in einem Format sein, dass Aufschluss darüber gibt, welche Daten hinzugefügt, welche geändert und welche gelöscht worden sind. Das hier verwendete Format muss genau dem gleichen Schema entsprechen, wie die original Daten. Zusätzlich soll eine Spalte oder ein Feld auf der obersten Eben mit dem Namen „cd_delted“ vorhanden sein. Es enhält einen Boolean-Wert, der sagt, ob der Eintrag aus den Daten glöscht wurde oder nicht. Der Datensatz mit den Änderungsdaten darf außerdem nur geänderte Daten enthalten. Aus diesem Format können, wie später gezeigt, die drei Operationen abgeleitet werden.

Um nicht auf ein externes Change-Data-Capture-System angewiesen zu sein, gibt es eine interne Lösung, die auf alle (semi-)strukturierten Daten angewendet werden kann. Der Algorithmus 2 zeigt, wie aus zwei DataFrames die Änderungsdaten erzeugt werden. Als Eingabe werden ein linkes DataFrame, mit dem aktuellen internen Stand, eine rechtes DataFrame, mit den neuen Daten und der Name der Id-Spalte benötigt. Das Ergebnis ist ein DataFrame mit allen aktualisierten Datensätzen. Es hat das gleiche Schema wie die Ursprungsdaten, aber mit der zusätzlichen Spalte, die Auskunft darüber gibt, ob ein Datensatz gelöscht wurde.

Das Einpflegen der Änderungen wird über die Delta Lake API gelöst. Dazu werden die Änderungsdaten mit den aktuellen Daten über die Id-Spalte zusammengeführt. Dabei können verschieden Fälle definiert werden. Wenn die Ids einer Zeile gleich sind und die Spalte „cd_deleted“ $true$ enthält, wird die Zeile aus den Daten gelöscht, ansonsten wird der Datensatz aktualisiert. Wenn die Ids nicht übereinstimmen und die Spalte „cd_deleted’"$false$ ist, dir der Datensatz als neue Zeile eingefügt.

Algorithmus 2: Deltaberechnung

Data: leftDataFrame, rightDataFrame, idColumn
Result: changeDataFrame
$leftDataFrame \leftarrow$ add column with row hash ;
$leftDataFrame \leftarrow$ prefix column names with „left_“ ;
$rightDataFrame \leftarrow$ add column with row hash ;
$rightDataFrame \leftarrow$ prefix columnnames with „right_“ ;
$changeDataFrame \leftarrow$ full join over $idColumn$ of $leftDataFrame$ and $rightDataFrame$;
$changeDataFrame \leftarrow$ remove all rows where $left\_hash$ equals $right\_hash$;
$changeDataFrame \leftarrow$ remove hash columns
$changeDataFrame \leftarrow$ add row $cd\_deleted$;
if $right\_idColumn$ *is* $null$ **then**
  $cd\_deleted = true$
else
  $cd\_deleted = false$
end
$changeDataFrame \leftarrow$ merge left and right $idColumn$ into one $idColumn$
if $left\_idColumn$ *is not* $null$ **then**
  $idColumn = left\_idColumn$
else
  $idColun = right\_idColumn$
end
$changeDataFrame \leftarrow$ merge remaining columns with left and right value by always taking the right and removing prefix

4.7.2 Plugins verwalten

Für jede Ingestion wird auf dem Speichersystem des Mircoservices ein temporärer Ordner angelegt, in den die Plugins und deren Abhänigkeiten installiert werden. Nach der Installation wird noch eine Datei angelegt, die für alle Packete die Versionen enthält. Das dient dazu, bei einer erneuten Ausfürhung auf dem Service nicht alle Packete neu installieren zu müssen, sondern nur die mit geänderten Version. Das beschleunigt die Ausführung der Ingestion. Zum Schluss wird der Ordner, in den die Packete installiert wurden an den Python-Pfad mit angehangen. Damit wird dieser während der Ausführung manipuliert und die Packete sind verfügbar. Da jede Ingestion in einem eigenen Prozess beinflussen die Änderungen an dem Python-Pfad den Ingestion-Service oder andere Prozesse nicht.

Im zweiten Schritt wird jede Python-Datei im Pluginordner als Modul geladen. Die im Modul verfügbaren Methoden werden dann überprüft, ob sie auf eine Definition der möglichen Plugins passen. Dazu wird der Algorithmus 3auf der nächsten Seite verwendet. Deisem werden das geladene Modul, ein Name der Methode, ein optionaler Rückgabetyp der Methode und eine Liste von Parameter, bei denen Name und Typ definiert ist. Zur Überprüfung können alle Methoden in dem Modul auf ihren Namen geprüft werden. Wenn eine Methode gefunden wurde, wird eine Signatur erzeugt und mit der übergebenen Definition verglichen. Das Ergebnis sagt dann, ob diese Methode ein Plugin ist oder nicht.

Jedes geladene Modul wird auf Load- oder AfterLoad-Methoden geprüft. Eine gefundene Load-Methode überschreibt immer die letzte gefunden, da bei einer Ausführung auch das Ergebnis dieser Methoden überschireben werden würde. Die AfterLoad-Methoden dagegen werden in einer Liste gespeichert.

Algorithmus 3: Pluginmethode überprüfen

```
Data: plugin, name, return_type, parameters
Result: matches
if plugin has NOT method with name name then
    return false
end
signature ← signature of mehtod name
if return_type is given AND signature NOT returns type of return_type
 then
    return false
end
forall param in parameters do
    if signature has NOT a parameter named param.name then
        return false
    end
    if signature type of parameter param.name is NOT parameter.type
     then
        return false
    end
end
return true
```

4.7.3 Ausführung der Ingestion

Die Ausführung startet mit der Initialisierung, die die für die Ingestion benötigte Daten lädt. Das ist zum Beispiel die DatasourceDefinition zu der Id aus den Events. Danach wird entschieden ob eine Ingestion über Spark notwendig ist. Hier spielt der Lese-Typ eine große Rolle. Eine Ingestion von Daten-Dateien benötigt keine Spark, es werden einfach direkt die Dateien in das entsprechende Zielverzeichniss kopiert. Für alle anderen Typen wird im nächsten Schritt die Ingestion vorbereitet. Es werden die Plugins aus dem HDFS geladen und eine SparkSession erstellt. Das Laden der Daten in ein DataFrame geschieht anshcließend entweder über ein Plugin in oder das standard Vorgehen. Falls auch After-Load-Plugins vorhanden sind werden diese ausgeführt. Für die geladenen Daten wird dann entschieden, ob es sich um Änderungsdaten handelt oder welche berechnet werden müssen. Je nach Schreib-Typ werden dann die Daten beziehungsweise Ädneurngsdaten gespeichert. Für Datenströme wird am Ende noch ein Hintergrund Task gestartet, der auf Kafka Events zum stoppen dieser Ingestion wartet. Hierfür wird die Topic „dls__ingestion__stop_ingestion“ mit der Id als Wert verwendet. Wenn die Ingestion beendet wurde, wird zum Schluss die SparkSession gestoppt.

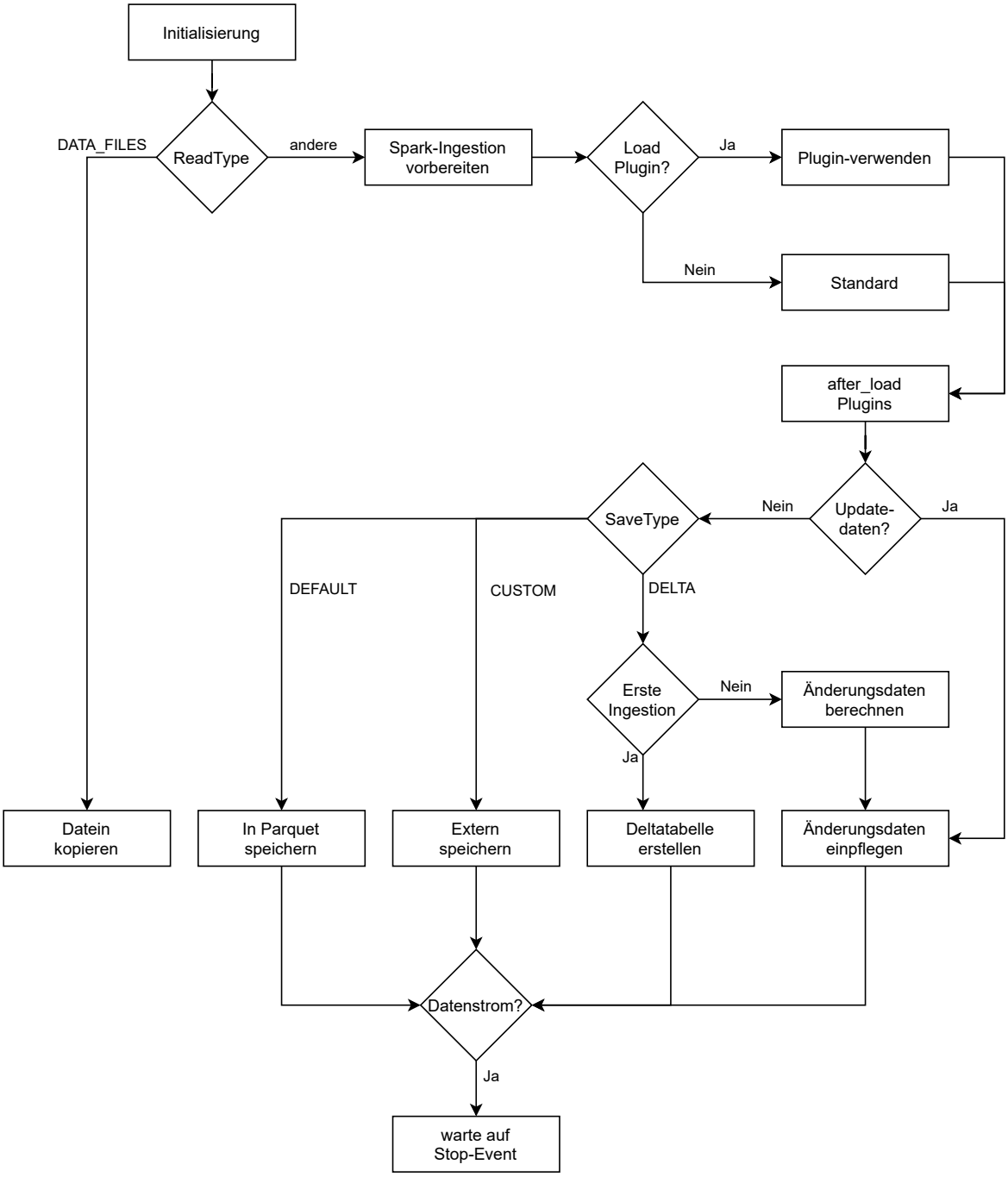


Abbildung 4.2: Ablauf einer Ingestion

Kapitel 5

Evaluierung

Kapitel 6

Ausblick

Literatur

Apache Kafka (2021). *Kafka Documentaion*. URL: https://kafka.apache.org/documentation/ (besucht am 28. 09. 2021).

Apache Parquet (2021). *Apache Parquet*. URL: https://parquet.apache.org/ (besucht am 14. 11. 2021).

Apache Spark (2021a). *Spark Overview*. URL: https://spark.apache.org/docs/latest (besucht am 04. 09. 2021).

— (2021b). *Spark Quick Start*. URL: https://spark.apache.org/docs/3.2.0/quick-start.html (besucht am 27. 10. 2021).

Apukhtin, Vladyslav, Mariya Shirokopetleva und Victoria Skovorodnikova (2019). „The Relevance of Using Message Brokers in Robust Enterprise Applications“. In: *2019 IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S T)*, S. 305–309.

Armbrust, Michael u. a. (2020). „Delta lake: high-performance ACID table storage over cloud object stores“. In: *Proceedings of the VLDB Endowment* 13.12, S. 3411–3424.

Borthakur, Dhruba (2010). „HDFS architecture“. In: *Document on Hadoop Wiki. URL http://hadoop. apache. org/common/docs/r0* 20.

crontab(5) — Linux manual page (Nov. 2012).

Fang, Huang (2015). „Managing data lakes in big data era: What’s a data lake and why has it became popular in data management ecosystem“. In: *2015 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER)*, S. 820–824.

Gos, Konrad und Wojciech Zabierowski (2020). „The comparison of microservice and monolithic architecture“. In: *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*. IEEE, S. 150–153.

Jarke, Matthias und Christoph Quix (2017). „On warehouses, lakes, and spaces: the changing role of conceptual modeling for data integration“. In: *Conceptual Modeling Perspectives*. Springer, S. 231–245.

Martin, Alexander, Marcel Thiel und Robin Kuller (2020/21). *Development of a Data Lake System*.

Mekterović Igor und Brkić, Ljiljana (2015). „Delta view generation for incremental loading of large dimensions in a data warehouse“. In: S. 1417–1422.

Ram P. und Do, L. (2000). „Extracting delta for incremental data warehouse maintenance“. In: S. 220–229.

Rooney, Sean u. a. (2019). „Experiences with Managing Data Ingestion into a Corporate Datalake“. In: *2019 IEEE 5th International Conference on Collaboration and Internet Computing (CIC)*, S. 101–109. DOI: 10.1109/CIC48465.2019.00021.

Schmidt, Felipe Mathias u. a. (2015). „Change data capture in NoSQL databases: A functional and performance comparison“. In: S. 562–567.

SEAGATE TECHNOLOGY (2020). *Rethink Data Report 2020*. URL: https://www.seagate.com/files/www-content/our-story/rethink-data/files/rethink-data-report-2020-de-de.pdf.

Singh, Ajit und Sultan Ahmad (2019). „Architecture of data lake“. In: *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* 5.2.

ZAHARIA, MATEI u. a. (2016). „Apache Spark: A Unified Engine for Big Data Processing.“ In: *Communications of the ACM* 59.11, S. 56–65. ISSN: 00010782. URL: https://printkr.hs-niederrhein.de:2589/login.aspx?direct=true&db=buh&AN=119379442&site=ehost-live.

Zhao, Yan, Imen Megdiche und Franck Ravat (2021). „Data Lake Ingestion Management“. In: *CoRR*.