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: 16. 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 Nachrichten 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 Seite 13 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 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 Seite 14 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 andere 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

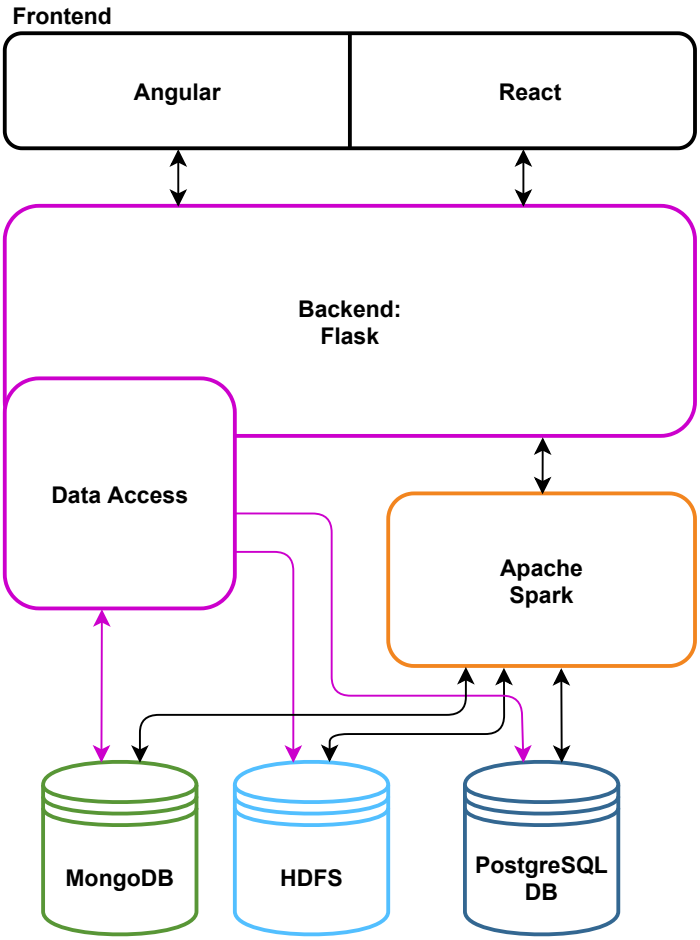
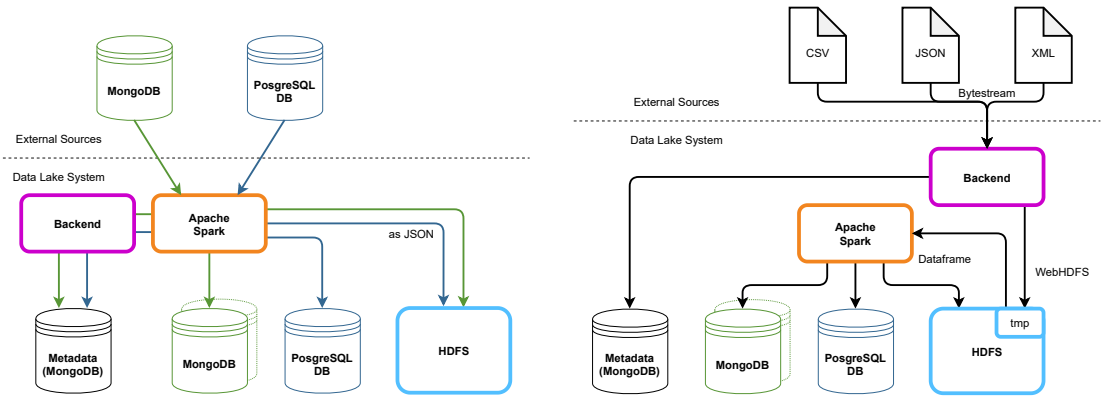


Abbildung 1.1: Architektur des Prototypen, Quelle: *Development of a Data Lake System*, S. 2



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

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

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 (Ram 2000).

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 (Mekterović 2015). 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

Entwicklung

In diesem Kapitel werden die einzelnen Komponenten, die für eine komplette Ingestion notwendig sind entwickelt. Nach **DL-Ing-Mgmt** 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. Um diese Aufgabe so gut wie möglich zu erfüllen, wurden im vorherigen Kapitel bereits die Anforderung festgelegt. Daraus ergeben sich die drei Inhaltlichen Bereiche, nach denen die weiter Entwicklung aufgeteilt ist.

Der erste ist die Ingestion. Diese beschreibt das erfassen von Informationen über eine Datenquelle und das Laden der Daten aus der Datenquelle. Dabei werden die Daten noch nicht intern im Data Lake wieder abgelegt sondern existieren nur flüchtig im Arbeitsspeicher.

Die Deltaerkennung ist dafür zuständig die Unterschiede zwischen den geladenen Daten im Arbeitsspeicher und den Bestandsdaten aus dem internen Data Lake Speicher zu finden. Aus diesen Unterschieden sollen Änderungsdaten erstellt werden, die dann bei der weiteren Verarbeitung verwendet werden können.

Der dritte Bereich ist das Speichern von Daten. Als Zielspeicher gibt es hier einmal ein vordefiniertes internes Speichersystem, es ist aber auch auch möglich ein benutzerdefiniertes zu verwenden. Der Unterschied da-

bei ist, dass im internen Speicher die Datenversionierung durch das Data-Lake-System unterstützt wird, während dies bei benutzerdefinierten nicht der Fall ist.

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. Diese Aufgaben werden dann auf Microservices verteilt. Die weitere Entwicklung des Systems stützt sich dann auf die fertige Architektur.

3.1.1 Aufgabenverteilung

Durch die Verwendung von *Apache Spark* ist es nicht sinnvoll, dass Laden der Daten, die Deltaerkennung und das Speichern zu trennen. In *Apache Spark* werden alle drei Arbeitsschritte auf einem DataFrame ausgeführt. Das heißt, dass dieses zwischen den Microservices ausgetauscht werden müsste, was zu einem erheblichen Aufwand führen würde. Aus diesem Grund kann die logische Aufteilung in Ingestion, Deltaerkennung und Speicherung nicht auch als Aufgabenverteilung verwendet werden.

Besser trennbare Aufgaben findet man bei der Betrachtung der technischen Seite. Hierbei gibt es die API, die Ingestion in *Apache Spark* und das kontinuierliche Ausführen.

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

3.1.2 Komponenten und Microservices

Die drei oben genannten Aufgaben können jeweils einem Microservice zugeordnet werden. In Abbildung 3.1auf der nächsten Seite ist eine dazu passende Architektur zu sehen. Der API-Service übernimmt die REST-Schnittstelle, der Ingestion-Service kümmert sich um die Ausführung der Ingestion und der Continuation-Service stellt die kontinuierliche Ausführung sicher.

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.

Zum Ablegen von internen Informationen wird eine Datenbank benötigt. Alle Microservices haben zugriff auf diese Datenbank und können Daten in ihr bearbeiten. Es handelt sich dabei aber nicht um den internen Speicher für geladene Daten des Data-Lake-Systems sondern nur um Daten, die für den Betrieb des Systems benötigt werden. Darunter fallen zum Beispiel Authentifizierungsdaten oder Verbindungsinformationen von Datenquellen.

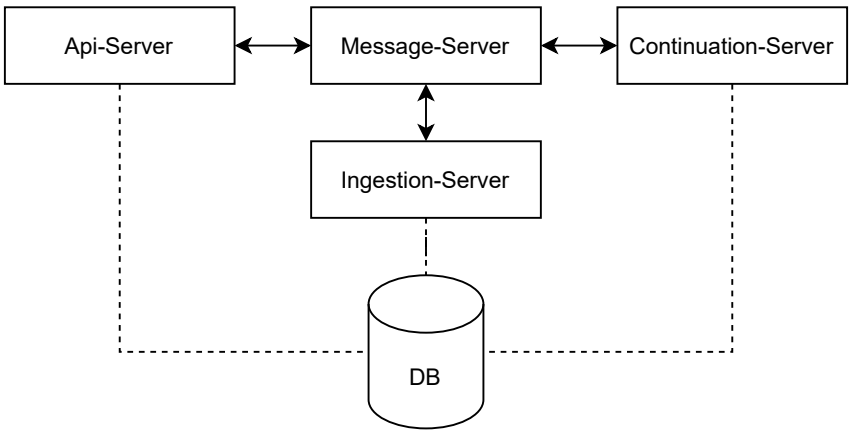


Abbildung 3.1: Architektur der Ingestion Komponenten

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. Die Plugins werden jeder Datenquelle einzeln hinzugefügt und werden an verschiedenen, fest definierten Punkten der Ingestion 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.

Bei der Ingestion gibt es zwei Stellen, an denen das Einbringen eines Plugins sinnvoll sein kann. Die erste ist zum Laden der Daten als Load-Plugin. Hier wird das standardmäßige Vorgehen der Ingestion mit dem des Plugins ersetzt. Dadurch wird es möglich anders als nur über *Apache Spark* Daten zu laden. Ein Beispiel dafür ist die Verwendung einer REST-API als Datenquelle. Das Plugin kann erst über mehrere Abfragen der REST-API den Datensatz abholen und diesen dann erst in eine DataFrame umwandeln. Ein Load-Plugin muss immer eine DataFrame zurück geben, mit dem danach in der Ingestion weiter verfahren werden kann. Damit ein DataFrame erstellt werden kann, muss dem Plugin außerdem

die entsprechende SparkSession mitgegeben werden.

Das After-Load-Plugin setzt im Gegensatz direkt nach dem Laden der Daten an. Diesem Plugin wird das geladene DataFrame übergeben und es muss auch wieder ein DataFrame zurück geben. Es kann genutzt werden um vor dem Speichern der Daten kleinere Anpassungen am Datensatz zu machen.

3.3 Datenquellen

Ein weiterer Kernpunkt der Ingestion-Schnittstelle ist die Modellierung der Datenquellen. Zur Entwicklung eines Datenmodells für die Datenquellen, wird als erstes betrachtet, welche verschiedenen Typen von Datenquellen möglich sind. Danach wird unter Zuhilfenahme des Ergebnisses ein Modell entwickelt. Das fertige Datenmodell enthält dann neben den für den Betrieb benötigten Informationen auch Daten über den Verlauf der Ingestion einer Datenquelle. Um den Verlauf von Änderungen an einer Datenquelle möglich zu machen werden alle Überarbeitungen in Revisionen festgehalten.

3.3.1 Datenquellen-Typen

Da das System alle möglichen Datenquellen unterstützen soll, werden hier mögliche Typen dargestellt, mit denen sich alle Datenquellen abdecken lassen. Durch die Verwendung von *Apache Spark* spielt bei der Unterscheidung der Daten quellen die Struktur keine Rolle. Es können sowohl strukturierte als auch semi- oder unstrukturierte Daten verarbeitet werden.

Die Typen der Datenquellen ergeben sich aus der Betrachtung, wie die Daten in das Data-Lake-System gelangen. Bei dem Pull-Prinzip ist das System dafür verantwortlich Daten aus einer Quelle zu laden. Das Gegenteil

dazu ist dass Pull-Prinzip. Bei diesem werden Daten direkt an das System gesendet. Das Push-Prinzip lässt sich dabei noch weiter aufteilen. Es gibt als erstes Datenströme, bei denen kontinuierlich laufend neue Daten an das System gesendet werden. Als zweites gibt es Dateien, die als Datenquelle dienen. Es wird bewusst auf die Möglichkeit verzichtet ganze Datensätze direkt an das Data-Lake-System zu senden, da es einfacher ist, eine große Menge von Daten auf ein oder mehrere Dateien zu verteilen und diese an System zu übergeben.

Daraus ergeben sich verschiedene Typen nach denen die Datenquellen aufgeteilt werden können. Diese sind Pull, Datei und Datenstrom. Bei der Verarbeitung können eine Pull- und eine Datei-Ingestion gleich behandelt werden. Für beide besteht die Möglichkeit die Ingestion nur einmalig auszuführen oder als kontinuierliche Ingestion über manuelle oder zeitgesteuerte Auslösung. Datenströme hingegen sind immer kontinuierlich.

3.3.2 Datenquellen-Modell

Das Modell einer Datenquelle besteht aus drei Teilen. Eine Liste von Ingestion-Events, die Informationen über jede ausgeführte Ingestion enthalten. Die Revisionen, die alle Informationen zur Datenquelle beinhalten, die durch den Benutzer geändert werden können. In einer Datenquelle werden diese beiden Listen zusammen mit unveränderlichen Informationen wie einer Id aggregiert.

Im folgenden werden die Felder der Modelle beschrieben. Falls eine Information direkt aus *Apache Spark* abgeleitet ist, wird auch Angegeben an welcher Stelle diese angewendet wird.

Revision

Nummer Eine aufsteigende Nummer, die die Revision identifiziert. Sie startet bei 0 und jede Nummer ist für eine Datenquelle einzigartig. Gleich-

zeitig spiegelt die Nummer auch den zeitlichen Verlauf wieder.

Erstellungsdatum Das Datum an dem die Revision erstellt wurde. Da eine Revision nicht geändert werden kann, ist das Erstellungsdatum einer Revision auch das Datum der Änderung der dazugehörigen Datenquelle.

Name Ein Name, der der Datenquelle im Datalake gegeben wird. Dieser dient nur dazu die Datenquelle als Benutzer besser identifizieren zu können.

Id Spalte Der Name der Spalte oder des Feldes, das einen Datensatz eindeutig identifiziert. Die Id Spalte wird für die Zuordnung der Datensätze bei der Deltaerkennung benötigt.

Spark Abhängigkeiten Eine Liste von Abhängigkeiten, die der Spark-Session bei der Ingestion mitgegeben wird. Diese Liste entspricht der Option „spark.jars.packages“ die bei der Erstellung einer Spark Session gesetzt wird.

Quelldateien Bei der Ingestion können die Daten über das Push-Prinzip in Form von Dateien an den Server gesendet werden. Die Liste der Quelldateien enthält die Pfade zu diesen Dateien.

Typ beim Lesen Der Typ der zu lesenden Daten, wie in 3.3.1 beschrieben.

Format beim Lesen Das Format in dem die Daten gelesen werden sollen. Der Wert aus diesem Feld wird bei der Ingestion über die `format` Methode eines Readers gesetzt.

Optionen beim Lesen Eine Liste von Schlüssel-Wert-Paaren, die als Optionen des Readers in *Apache Spark* gesetzt werden. Über diese wird die Verbindung zur Datenquelle definiert.

Aktualisierungsquelle Es kann für einen Datenquelle festgelegt werden, dass diese Aktualisierungen für eine andere enthält. Dann wird sie Aktualisierungsquelle genannt. Das ermöglicht die Nutzung von eigenen Change Data Capture Lösungen. Dabei muss die Datenquelle aber dem einem festgelegten Anderäungsdatenformat entsprechen, das später genauer erläutert wird. Das Feld selbst enthält die Id der Ziel-Datenquelle für die Aktualisierung.

Typ beim Schreiben Der Typ für das Schreiben der Daten legt fest, ob intern mit Versionierung oder in einen freien Speicher geschrieben werden soll.

Format und Optionen beim Schreiben Das Format und die Optionen beim Schreiben funktionieren analog zu denen beim Lesen. Sie werden jedoch nicht beim Reader gesetzt sondern beim Writer. Gerade bei Datenströmen muss darauf geachtet werden, dass nicht in jedem Format ein Datenstrom geschrieben werden kann.

Schreibmodus Der Modus in dem die Daten geschrieben werden. Die Auswahlmöglichkeiten werden auch durch *Apache Spark* festgelegt.

Zeitsteuerung Eine Liste die festlegt, zu welchen Zeitpunkte eine Ingestion der Datenquelle ausgeführt werden soll.

Plugin Abhängigkeiten Eine Liste von Abhängigkeiten, die von den Plugins der Datenquelle benötigt werden.

Plugins Die Speicherorte der Plugins.

Ingestion-Event

Nummer Die Nummer um ein Ingestion-Event zu identifizieren. Sie funktioniert genau wie die Nummer der Revision.

Status Der aktuelle Status in dem sich das Event befindet. Er gibt Auskunft, ob die Ingestion gestartet wurde, gerade läuft oder beendet wurde.

Revisionsnummer Die Nummer der Revision, mit der die Ingestion gestartet wurde. Mit Hilfe der Revisionsnummer ist es leichter Fehler in der Ingestion zu finden und zu beheben.

Start- und Endzeit Die Zeitpunkte des Starts und Endes eines Ingestion-Durchlaufs. Wenn die Ingestion noch in der Ausführung ist, ist keine Endzeit gesetzt.

Fehler Die Fehlerausgabe, wenn die Ingestion nicht erfolgreich ausgeführt werden konnte.

Datenquelle

Id Eine Identifikationsnummer, die für jede Datenquelle eindeutig ist.

Aktuelle Revision Die Nummer der aktuellen Revision. Neue Ingestions werden mit den Informationen aus dieser Revision ausgeführt.

Alle Revisionen Eine Liste aller Revisionen.

Letzte Ingestion Die Nummer des zuletzt gestarteten Ingestion-Durchlaufs.

Letzte erfolgreiche Ingestion Die Nummer des letzten erfolgreich abgeschlossenen Ingestion-Durchlauf.

Alle Ingestion-Events Eine Liste aller Ingestion Events.

3.4 API-Service

Für den API-Service müssen Endpunkte definiert werden. Diese Endpunkte bilden die verschiedenen Funktionen ab, die auf der Ingestion-Schnittstelle ausgeführt werden können. Dazu gehören die Verwaltung von Datenquellen und das Starten einer Ingestion. Da es bereits festgelegt wurde, dass es sich um eine REST-Schnittstelle handelt, werden Endpunkte durch einen Pfad und eine HTTP-Methode definiert. Nachfolgend werden alle Endpunkte aufgelistet.

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

Außerdem kümmert sich der API-Service um die Erstellung von Datenquellen, deren Revisionen und Ingestion-Events. Bei Anfragen zum Starten einer Ingestion versendet der API-Server eine Nachricht, mit der Id der Datenquelle.

3.5 Continuation-Service

Für die Sicherstellung der korrekten Ausführung von kontinuierlichen Ingestions, müssen regelmäßig alle Datenquellen überprüft werden. Dabei gibt es zwei Fälle, die entscheiden ob eine neue Ingestion für eine Datenquelle gestartet 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 wie in der Einleitung des Kapitels bereits genannt, das Laden der Daten, die Deltaerkennung und das Speichern. Damit ein Ingestion-Service mehrere Ingestions zu verschiedenen Datenquellen gleichzeitig durchführen kann, wartet dieser auf Benachrichtigungen der anderen Services, dass eine Ingestion gestartet werden soll. In dieser Nachricht wird auch die Id der Datenquelle übertragen. Dann lädt der Ingestion-Service die Datenquelle aus der internen Datenbank und prüft ob eine Ingestion für diese bereits läuft. Wenn das nicht der Fall ist, wird die Ingestion in einem neuen Prozess gestartet.

3.6.1 Ablauf

Der Ablauf einer einzelnen Ingestion kann über alle Typen gleich etwas abstrahiert gezeigt werden. Im ersten Schritt müssen die Plugins für die Ingestion geladen werden. Dabei ist es wichtig auch eventuelle Abhängigkeiten von Bibliotheken aufzulösen.

Danach können die Daten entweder über das Plugin oder normal in ein Dataframe geladen werden. Ingestions von Dateien oder nach dem Pull-Prinzip können gleich abgehandelt werden. Nur die Pfade zu den Dateien müssen durch den Service ergänzt werden. Eine Ingestion eines Datenstroms hat eine eigene Behandlung.

Im Anschluss wir geprüft, ob eine Deltaerkennung für die Daten durchgeführt wird. Dazu müssen drei Bedienungen erfüllt sein. Es darf sich nicht um eine Datenquelle handeln, für die ein benutzerdefinierter Speicher festgelegt wurde. Für diese kann keine Versionierung unterstütze werden. Außerdem ist es für Aktualisierungsquellen nicht nötig, eine Deltaerkennung durchzuführen, da diese schon im Änderungsdatenforamt sein sollen. Für den letzten Punkt muss es sich um die wiederholte Inge-

stion einer Datenquelle handeln, da bei der ersten keine Bestandsdaten für eine Deltaerkennung existieren.

Zum Schluss wird auch beim Speichern der Daten unterschieden. Bei benutzerdefinierten Speichern können die Daten wie konfiguriert ohne weiteren Aufwand gespeichert werden. Das gilt für sowohl Datenströme als auch andere Ingestions. Änderungsdaten müssen in eine existierenden Speicher integriert werden. Der letzte Fall bedeutet, dass Daten das erste mal im internen Speichersystem gespeichert werden und auch hier in einer neuen Datensatz abgelegt werden können.

Kapitel 4

Umsetzung

Für eine Umsetzung der entwickelten Architektur ist zuerst die Frage der Techniken zu klären. Dazu zählen die verwendete Programmiersprache, Frameworks und fertige Software.

4.1 Programmiersprache

Da bereits im Data-Lake-Prototyp die Programmiersprache Python verwendet wird, ist dies die erste Möglichkeit. Für die Verwendung mit *Apache Spark* stehen außerdem noch zwei weitere Sprachen, Java und Scala, zur Verfügung. Python bietet jedoch im Vergleich hat dabei Python einen entscheiden Vorteil, der zur Wahl von Python führt. Bei der Verwendung von Java und Scala müssen über ein „spark-submit“ fertig kompilierte Jobs an das Cluster gesendet werden. Das bedeutet, dass der Code dieser Jobs bereits feststehen muss.

Diese Option besteht auch bei Python. Zusätzlich gibt es jedoch auch die Möglichkeit, dass der Interpreter zur Laufzeit sich um die korrekte Ausführung der Jobs kümmert. So ist es möglich ebenfalls zur Laufzeit für jede Anfrage speziell definierte Jobs zu erstellen, die nicht vorher schon kompiliert sein müssen. Das spart sowohl Komplexität bei der Entwick-

lung als auch Zeit bei der Ausführung, da es nicht nötig ist für jede Anfrage einen Job neu zu kompilieren (Apache Spark 2021b).

Außerdem hat Python die Möglichkeit ebenfalls zu Laufzeit Programmcode aus einer Datei zu importieren. Dieser Mechanismus soll verwendet werden um die Funktion von Plugins zu ermöglichen.

Aus den oben genannten Gründen ist Python die Programmiersprache, die bei der Ingestion zum Einsatz kommen soll. Zur Integration der Ingestion-Schnittstelle in den alten Data-Lake-Prototyp wird dessen Server als Grundlage für den Api-Service genommen und ist somit auch in Python geschrieben. Bei dem Continuation-Service gibt es keine großen Vor- oder Nachteile zwischen den Programmiersprachen. Hier fällt die Wahl auf Python auf Grund der Einheitlichkeit.

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.

Durch die Verwendung von *Kafka* als Message-Broker müssen die Topics der Nachrichten festgelegt werden. Hier kann es zu Konflikten so-

wohl mit internen Nachrichten als auch bei anderen Datenströmen kommen, wenn die Namen falsch gewählt werden. Daher sollten Topics, die zur internen Kommunikation des Data-Lake-Systems gehören immer mit „dls__“ als Prefix benannt werden. Danach folgt der Bereich in dem die Nachricht verwendet wird, hier zum Beispiel „ingestion“. An diesen Namen kann dann noch weiter Unterscheidung angehängt werden. Im Fall der Nachricht für das Ausführen einer Ingestion wäre dann die Topic „dls__ingestion__run“.

Wenn mehrere Consumer in einer Gruppe für eine Topic sind, werden Nachrichten nicht an alle sonder immer nur an einen aus der Gruppe gesendet. 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 Listen von anderen Datenmodell enthält, 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 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 aber auch die verknüpften Einträge der Ingestion-Events oder Revisionen gebraucht werden. 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. Das Speichern der Felder für die Optionen beim Lesen und Schreiben wird auch dadurch erleichter, dass es kein Schema gibt. Man kann so die Optionen einfach in einem

Dokument in der Revision speichern.

Hier kommt *MongoDB*[1] als Datenbank zum Einsatz. *MongoDB* kann frei verwendet werden und bei bei größeren Datenmengen verteilt eingesetzt werden. Abbildung 4.1auf der nächsten Seite zeigt einen Überblick, wie das entwickelte Datenmodell aus 3.3.2 gespeichert wird.

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 Funktionen bietet und eine volle Schnittstelle zu Spark hat. Da das System auf eigenen Server laufen soll, wird als Speicher für den Delta Lake ein HDFS Cluster verwendet. Außerdem können auch die Quelldateien der Ingestion und die Plugins im HDFS abgelegt werden und so zentral durch alle Microservices erreicht werden. Der Zugriff zum Speichern der Daten geschieht über den Delta Lake beziehungsweise Spark und für die Quelldateien und Plugins über die WebHDFS REST-Schnittstelle.

Wenn eine Datenquelle entweder mit Versionierung gespeichert wird oder Quelldateien oder Pluigns enthält, wird für diese ein Ornder im HDFS angelegt. Diese enthält dann einen Unterordner für die Delta Tabelle und für jede Revision. In den Revisionsordnern werden dann zugehörige Quelldatein und Plugins abgelegt.

4.5 Api-Service

Der Api-Service basiert auf dem Server aus dem Masterprojekt, ein mit Flask[2] in Python geschriebener Web-Server. Um eine saubere Trennung

[1] https://www.mongodb.com/
[2] https://flask.palletsprojects.com/

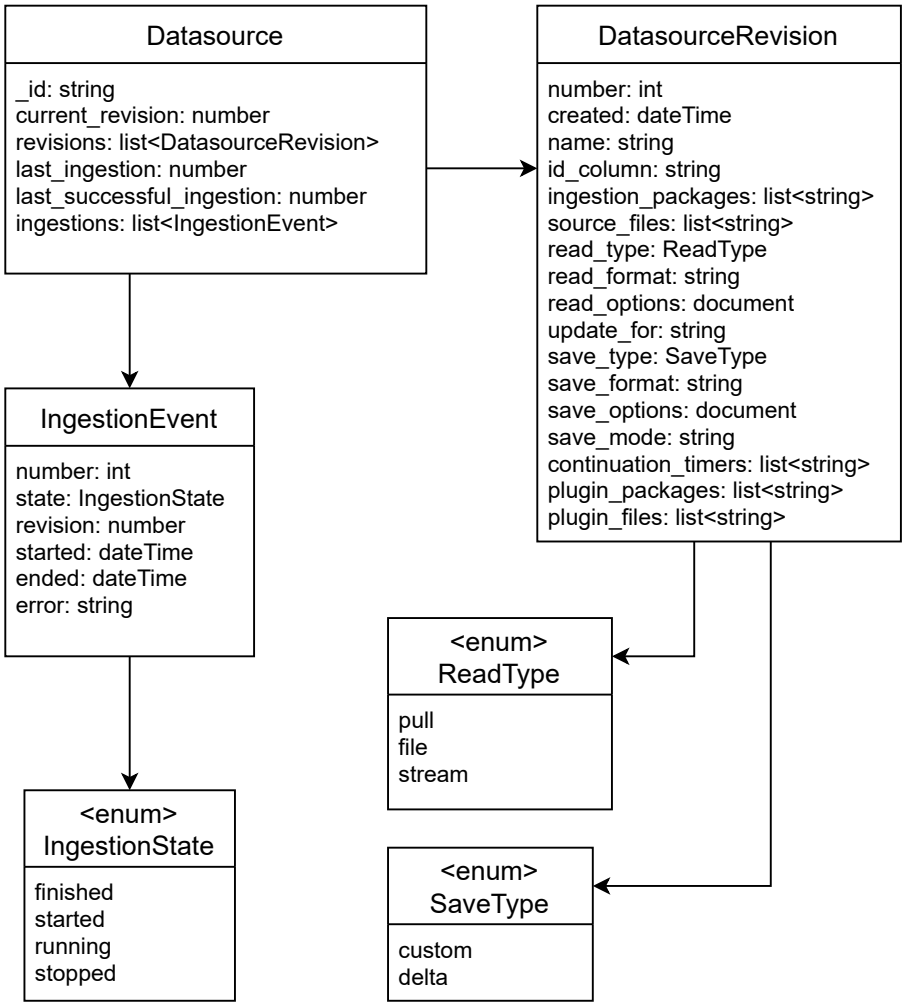


Abbildung 4.1: Übersicht Datenmodell

zwischen den existierenden und den neuen REST Endpunkten zu erreichen, wird die alte API unter den Pfad „/api/v1/“ verschoben und die neue unter „/api/v2/“ erstellt. Kommunikation mit dem Data Lake System findet nur statt, wenn eine Anfrage zur Ausführung einer Ingestion bearbeitet wird. Hier wird ein Event mit dem Schlüssel „dls__ingestion__run“ erzeugt.

4.6 Continuation-Service

Der Continuation-Service ist ein einzelner Prozess. In einer Schleife werden erst alle Datenquellen überprüft, die eine Zeitsteuerung enthalten. Im zweiten Schritt werden alle Datenquellen mit Datenströmen geprüft. 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.

Algorithmus 1: Continuation loop

```
loopStart ← now  checkSources ← query datasource with timers
 forall sources in checkSources do
 |  timers ← timers of source  forall timer in timers do
 |  |  if timer is now then
 |  |  |  if source is not running then
 |  |  |  |  send run event
 |  |  |  end
 |  |  end
 |  end
end
looEnd ← now  loopDuration ← loopEnd - loopStart  if
 loopDuration < 60 sec then
 |  sleep for 60 sec - loopDuration
end
```

4.7 Ingestion-Service

Der Ingestion-Service wartet auf den Empfang von Nachrichten mit dem Schlüssel „dls_ingestion_run“. Diese Nachrichten sollten immer die Id einer Datenquelle enthalten, die geladen werden soll. Bevor das passiert wird überprüft, ob bereits eine Ingestion für diese Datenquelle läuft. Die Ingestion selber läuft in vier Phasen ab.

Zuerst wird die Ingestion mit Spark vorbereitet. Für jede Datenquelle wird ein temporärer Ordner angelegt. In diesen Ordner werden die Plugin-Dateien aus dem HDFS heruntergeladen und über das Packetmanagement von Python, pip, die Abhängigkeiten für die Plugins installiert. Aus den Informationen in der Datenquelle wird eine SparkSession erstellt.

In der zweiten Phase werden die Daten aus einer Datenquelle in eine DataFrame geladen. Durch die Verwendung von Spark’s Structured Streaming können alle Typen von Datenquellen ähnlich verarbeitet werden. Das Sturctured Streaming ist ein Bibliothek, die auf Spark SQL aufbaut und es ermöglicht, Daten mit Hilder der DataFrame API zu verarbeiten. Für die Datenquelle wird mit den gesetzten Optionen ein DataStream- oder DataFrame-Reader erzeugt, mit dem die Daten geladen werden. Bei der File-Ingestion werden zusätzlich die Pfande zu den Quelldateien im HDFS gesetzt.

Für die geladenen Daten muss dann entschieden werden, ob eine Deltaerkennung und somit eine Umwandlung in Änderungsdaten nötig ist. Folgende Regeln müssen zutreffen, damit eine Deltaerkennung ausgeführt wird. Es handelt sich nicht um einen Datenstrom. Die Ingestion ist nicht die erste dieser Datenquelle oder hat einen benutzerdefinierten Speicherort. Die Datenquelle ist keine Updatequelle. In Algorithmus 2 ist zu sehen, 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.

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

Die letzte Phase ist das Speichern. Für Datenströme wird entweder der benutzerdefinierte Speicherort gewählt, oder die Daten werden über den „append“-Modus in eine Delta Tabelle geschrieben. Bei File- und Pull-Ingestions gibt es ebenfalls zuerst die Unterscheidung ob ein benutzer-

definierter Speicher verwendet wird oder nicht. Wenn die Daten intern gespeichert werden sollen gelten nachfolgende Regeln. Eine neue Delta Tabelle wird erstellt, wenn es sich um die erste Ingestion einer Datenquelle handelt, die keine Updatequelle ist. Sonst werden die Änderungsdaten in die Delta Tabelle der Zieldatenquelle eingepflegt. Für Updatequellen ist die Zieldatenquelle über die Konfiguration festgelegt und für eine normalen Datenquelle ist sie es selbst. 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.

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.

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.