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: 31. Dezember 2021

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

Eidesstattliche Erklärung

Name: Alexander Martin
Matrikelnr.: 1018332
Titel: Entwurf und Implementierung einer generischen Ingestion-Schnittstelle mit Versionierung für Data-Lake-Systeme

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

Heutzutage spielen Daten eine immer wichtiger Rolle. Durch den vermehrten Einsatz von IoT-Geräten und moderne Cloud-Speicher-Lösungen, wächst die Zahl an anfallenden Daten vielen Firmen und Forschungseinrichtungen stetig. Mit steigenden Datenmengen und der diversen Strukturen der Daten ist deren Verwaltung ein komplexes Thema geworden. Diese Arbeit befasst sich mit der technischen Herausforderung Daten aus verschiedensten Quellen zu Verwalten. Als Lösung hierfür wurden Data-Lake-Systeme vorgeschlagen. Im Kontext des HIT-Institut der Hochschule Niederrhein, wurde ein Prototyp für einen Data Lake entwickelt. In dieser Arbeit wird eine Schnittstelle entwickelt, über die Benutzer Daten aus unterschiedlichen Quellen in dieses System laden können. Dabei werden Metadaten über die Datenquellen gesammelt. Mit der Schnittstelle ist es auch möglich, die geladenen Daten zu versionieren, um weiter Verarbeitungen effizienter zu machen.

Abstract

In todays world data play an important role. The amount of data in comapnies and research facilities is growing due to the increasing use of IoT-devices and modern Cloud-Storage-Solutions. With the bigger amount of data and their varying structures the data management got more complex. This thesis takes on the technical challenge of managing data from different sources. Data lakes are a proposed solution for this problem. A prototype for a data lake was developed at the HIT-institute at the Hochschule Niederrhein. This thesis devolopes an interface for users to ingest data from diffenrent sources to the system. The interface collects metadata about the data sources. It is also capable of saving data with versioning to make further processing more efficient.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Heutzutage spielen Daten eine immer wichtigere Rolle. In vielen Firmen und Forschungseinrichtungen wächst die Zahl an unterschiedlichen Daten stetig. Im *Rethink Data Report 2020* **rethink_data_2020** wurde eine Studie durchgeführt, die eine Steigerung von 42% der Menge an anfallenden Daten pro Jahr prognostiziert. Diese Daten sind zum Beispiel auf den vermehrten Einsatz von IoT-Geräten oder ausführlicher werdende Analysen zurückzuführen. Auch, dass es durch moderne Cloud-Speicher-Lösungen einfacher geworden ist große Datenmengen zu speichern, begünstigt diese Entwicklung. Daten sind eine wertvolle Ressource und müssen entsprechend gut verwaltet werden. Durch die unterschiedlichen Formate und Strukturen, zum Beispiel Datenbanktabellen, JSON-Dateien oder Bilder ist die Verwaltung ein komplexes Thema. Die Bewältigung der organisatorischen Aufgaben fällt in den Bereich Data Governance. Diese Arbeit beschäftigt sich mit den technischen Herausforderungen Daten aus einer Vielzahl von Datenquellen zu verwalten.

In vielen Unternehmen besteht das Problem, dass Daten in sogenannten Datensilos gelagert werden. Das bedeutet, dass für verschiedene Anwendungen oder Formate isolierte Speichersysteme verwendet werden. Dabei entsteht das Problem, dass der Überblick, welche Daten es in der gesamten Systemlandschaft gibt, verloren geht. Auch Daten für Analysen untereinander zu verknüpfen wird mit zunehmender Datenmenge und Variation schwieriger.

Ein klassischer Ansatz, der die Analysen vereinfachen soll, ist das Data Warehouse. Ein Data Warehouse besteht aus verschiedenen Datamarts, in denen Da-

ten für bestimmte Analysen abgelegt werden. Daten werden aus den Quellsystemen geladen, mit verschiedenen Transformationen auf ein für den Datamart globales Schema gebracht und dann abgespeichert (**dw**). Dieses Vorgehen wird auch ETL (Extract Transform Load) genannt, da die Daten erst aus der Quelle extrahiert, dann transformiert und gespeichert werden und erst nach diesen Schritten für eine Analyse geladen werden können. Bei der Transformation der Daten kann es oft passieren, dass ein Teil ihres Informationsgehalts verloren geht, da nicht alle Felder in den Datamart übernommen werden. Außerdem wird eine Änderung an einem Schema teurer, je mehr Daten bereits integriert wurden. Als Lösung für diese Probleme wurden Data Lakes vorgeschlagen (**dixon2010pentaho**; **datalake_03**).

1.1 Data-Lake-Systeme

Ein Data Lake ist ein System, das die Erfassung, Verfeinerung, Archivierung und Erkundung von Daten vereinfacht und verbessert (**datalake_01**). Es sollen große Datenmengen möglichst kostensparend gespeichert und verschiedenste Formate verarbeiten werden können (**datalake_02**). Diese Systeme verfolgen dabei den ELT (Extract Load Transform) Ansatz. Das Kernprinzip ist die Speicherung der Daten in ihrem Rohformat. Die Transformation findet erst statt, nachdem die Daten für weitere Verarbeitungen geladen wurden. Dadurch fällt der Aufwand für eine Transformation vor dem Speichern, wie bei einem Data Warehouse, weg und Daten sind schneller für weitere Verarbeitungen bereit. Außerdem gehen dabei keine Informationen mehr verloren.

Wie in **??** gezeigt wird, basiert ein Data-Lake-System auf vier verschiedenen Ebenen. Diese sind die Interaktions-, Transformations-, Speicher- und Ingestion-Ebene. Über die Interaktions-Ebene kann auf die Daten im Data Lake zugegriffen und Metadaten verwaltet werden. Die Transformations-Ebene bereitet die Daten auf, nachdem diese aus der Speicher-Ebene geladen wurden. Als letztes gibt es die Ingestion-Ebene. Diese ist für die Integration von Daten in den Data Lake verantwortlich. Gleichzeitig werden hier auch Metadaten aus den Daten extrahiert. Die Umsetzung eines solchen Data Lakes kann je nach Voraussetzungen des Einsatzbereichs variieren.

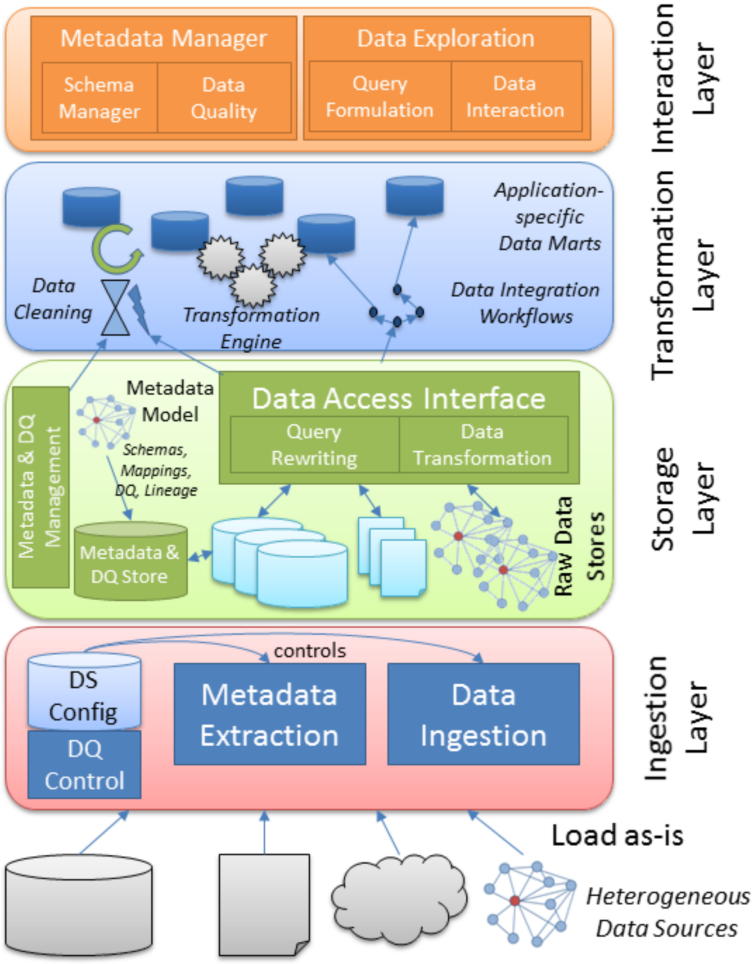


Abbildung 1.1: Architektur eines Data Lakes (**datalake_03**)

1.2 Motivation

In einem Data Lake spielen die Metadaten eine zentrale Rolle. Die Qualität der Metadaten bestimmt, wie gut Daten im Data Lake gefunden und in einen Zusammenhang gebracht werden können. In den Metadaten können dabei sowohl Informationen über die Herkunft von Daten als auch über deren Inhalt oder Qualität stehen.

Für das HIT-Institut der Hochschule Niederrhein soll ein Data Lake entwickelt werden, der für die Verwaltung der Daten eingesetzt wird. In diesem System sollen verschiedene Quellen zusammengebracht werden. Dazu gehören Datenbank-Systeme, Dateien und Daten aus speziellen Software-Systemen, die über eine REST-API erreichbar sind. Hier sollen die Daten jedoch nicht nur einmalig in das System geladen, sondern kontinuierlich aktualisiert werden. Der einfachste Ansatz ständig den vollständigen Datensatz hinzuzufügen ist jedoch zu aufwändig und speicherintensiv. Daher muss das Data-Lake-System eine Versionierung der Daten unterstützen und Änderungen in den Quellen erkennen können, so dass nur diese gespeichert werden. Dies würde auch weiteren Prozessen, wie Transformation, Integration oder Analyse der Daten beschleunigen. Diese müssten nur noch die Änderungen und nicht mehr den kompletten Datensatz verarbeiten. Es gibt aktuell noch kein fertiges System, dass all diesen Ansprüchen genügt.

1.2.1 Zielsetzung der Arbeit

In einem Masterprojekt wurde bereits ein Prototyp für ein generelles Data-Lake-System entwickelt (**prototyp**). Es wurde so aufgebaut, dass der einfache Einsatz in verschiedenen Anwendungsbereichen möglich ist. Es beinhaltet Komponenten für die Umsetzung der in **??** und durch **datalake_03** beschriebenen Funktionen und die dafür notwendigen Komponenten.

Mit dem Prototypen als Grundlage wird in dieser Arbeit eine Schnittstelle für die Ingestion entwickelt. Dabei müssen drei allgemeine Bedingungen erfüllt werden:

- die Schnittstelle soll unabhängig von und mit allen Datenquellen verwendet werden können,

- eine Aktualisierung der Daten soll kontinuierlich und automatisch möglich sein und
- die Daten sollen mit einer Versionierung gespeichert werden können.

Genauer betrachtet ergeben sich daraus folgende Aufgaben und Fragestellungen:

- Entwicklung einer Ingestion, die mit geringem spezifischen Aufwand mit allen Datenquellen kompatibel ist
  - Wie sieht eine Ingestion aus?
  - Wie werden Datenströme geladen?
  - Wie werden Daten aus einer API geladen?
  - Lässt sich daraus eine allgemeine Art der Definition ableiten, die von der Ingestion-Schnittstelle verwendet werden kann?
- Die Erkennung und Speicherung von Änderungen zwischen dem aktuellen Stand im System und dem Stand der Datenquelle
  - Wie erkennt man die Änderungen zwischen den Daten?
  - Wann soll diese Erkennung gemacht werden?
  - Lässt sich die Erkennung für alle Datenquellen gleich gestalten?
  - Lässt sich die Erkennung erweiterbar gestalten, um komplexere Situationen abzubilden?
  - Wie wird die Deltaerkennung in den Ingestion-Ablauf integriert?
- Speichern und Verwalten der Versionierung von Daten
  - Wie speichert man die Daten mit Versionierung?
  - Wie können Abfragen über die Datenversionen gestellt werden?
- Evaluierung des Systems
  - Anwendung des Systems mit Daten, die verschiedene Datenquellen abbilden

1.2.2 Aufbau

Die Arbeit gliedert sich in sieben Kapitel. Im nächsten Kapitel werden wichtige Grundlagen für die Arbeit vermittelt und verwandte Arbeiten erläutert. Der weitere Aufbau orientiert sich an der Vorgehensweise der Software-Entwicklung. Zuerst werden in Kapitel 3 die Anforderungen ermittelt. Dazu werden die Ziele für die Ingestion-Schnittstelle definiert. Aus diesen Zielen können dann genaue Anforderungen abgeleitet werden. Danach wird ein Entwurf für das System erstellt. Der erste Schritt ist das Design einer Architektur. Danach werden alle für die Umsetzung dieser Architektur benötigten Komponenten herausgearbeitet. Kapitel 4 befasst sich mit der Umsetzung der Architektur und Komponenten. Hier werden verwendete Techniken erläutert und begründet. Außerdem werden wichtige Implementierungsdetails erläutert. Auf genaue Beschreibungen der Programmierung wird verzichtet, da diese nicht bestimmend für das System sind. In Kapitel 5 wird das System evaluiert. Die Evaluierung teilt sich in funktionale Tests und in Benchmarks auf. Die Arbeit schließt mit einer Zusammenfassung über der wichtigsten Ergebnisse und einem Ausblick auf weitere Arbeiten.

Kapitel 2

Grundlagen und Verwandte Arbeiten

Dieses Kapitel befasst sich mit den technischen Grundlagen dieser Arbeit und verwandten Arbeiten. Unter den technischen Grundlagen werden zunächst Rahmenwerke erklärt, die im Laufe dieser Arbeit eingesetzt werden. Danach folgt eine genauere Erklärung von Data Lakes. Zum Schluss wird auf das Change-Data-Capture (deutsch: Änderungserfassung) eingegangen. Das ist nicht direkt Teil der Entwicklung in dieser Arbeit, spielt aber im Anwendungsbereich eine große Rolle.

2.1 Technische Grundlagen

2.1.1 Hadoop

Hadoop[1] ist ein Projekt von Apache, für die verteilte Datenverarbeitung auf einem Cluster. Die Verarbeitung basiert auf dem Map-Reduce (**mapred**). Das ist ein Programmier-Modell, bei dem Daten in Form von Schlüssel-Wert-Paaren verarbeitet werden. Grob beschrieben, besteht es aus zwei Funktionen. Die erste ist die Map-Funktion, bei der aus Daten eine Zwischensammlung von Schlüssel-Wert-Paaren erzeugt wird. Danach werden diese nach den Schlüsseln sortiert

[1] https://hadoop.apache.org/

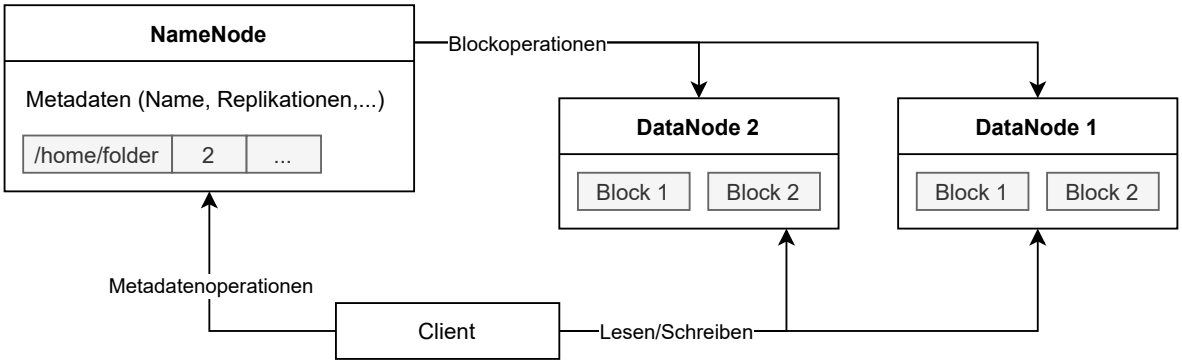


Abbildung 2.1: HDFS-Architektur[2]

und die Reduce-Funktion, zum Zusammenfassen der Paare ausgeführt. Ein Hadoop-Cluster enthält zwei wichtige Komponenten. YARN (**yarn**) wird für das Ressourcen-Management benutzt. Die zweite Komponente, das HDFS, ist ein verteiltes und fehlertolerantes Dateisystem, welches ursprünglich für Hadoop entwickelt wurde. Für dieses Projekt ist nur das HDFS relevant und wird deswegen näher erläutert.

Das HDFS wurde entwickelt, um auf Hardware mit geringen Kosten zu laufen und große Datenmengen zu verarbeiten. 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 aufgebaut. Dateien und Ordner können im Namensraum hierarchisch organisiert werden. Es unterstützt jedoch keine Zugriffsberechtigung 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 Blöcke, bis auf den letzten, 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 Namensraums und die Verteilung der einzelnen Blöcke einer Datei. Er reguliert dazu noch den Zugriff durch Clients und führt Operationen

[2]Nach https://hadoop.apache.org/docs/r1.2.1/images/hdfsarchitecture.gif, Zugriff: 29.12.2021

auf dem Dateisystem, wie das Öffnen, Schließen oder Umbenennen von Ordnern und Dateien aus. Die DataNodes speichern die einzelnen Blöcke der Dateien. Auf Anweisung des NameNode werden Blöcke erstellt, gelöscht oder repliziert. Außerdem bearbeiten sie Anfragen zum Lesen und Schreiben von Dateien (**hdfs**).

Mit Apache Parquet steht ein Format zur Verfügung, mit dem Daten im HDFS effizient gespeichert werden können. Parquet ist ein spalten-orientiertes Speicherformat, das auch die Kompression und Kodierung der Daten unterstützt. Über einen Algorithmus zur Zerlegung von Verschachtelung ist auch das Speichern von semistrukturierten Daten möglich (**parquet**).

2.1.2 Apache Spark

Apache Spark ist eine Datenverabeitungs-Engine für die effiziente, verteilte Verarbeitung von Big Data. Das Ziel bei der Entwicklung von Apache Spark war es, ein einheitliches Rahmenwerk für Big-Data-Prozesse zu schaffen. Das Problem war, dass die bis dahin verbreitetste Lösung Hadoop keine einheitliche Abfragesprache und kein einheitliches Datenmodell hat. Durch Spark ist das Arbeiten mit zum Beispiel SQL, Datenströmen, maschinellem Lernen oder Graph-Daten möglich. Es gibt viele Bibliotheken für die Verwendung verschiedener Datenquellen in Spark. Durch ihre Optimierung erreichen diese eine ähnliche Performance wie manuell dafür implementierte Big-Data-Prozesse. 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: Resiliente Verteilte Datensätze). RDDs sind fehlertolerante Sammlungen von Objekten, die auf die Worker-Instanzen verteilt und parallel bearbeitet werden können. Diese werden flüchtig im Hauptspeicher gehalten, können aber für spätere schnellere Zugriffe zwischengespeichert (persistiert) werden. Die Erstellung und Bearbeitung von RDDs geschieht über sogenannte Transformationen. Die Transformationen werden in einem Herkunftsgraphen gespeichert, wodurch eine Wiederherstellung bei Fehlern an jedem Punkt möglich ist.

Für die Verarbeitung von strukturierten oder semistrukturierten Daten gibt es zusätzlich die eigene Abfragesprache SparkSQL, die sich stark an SQL ori-

entiert. Es gibt Bibliotheken für die Sprachen Scala, Java, Python und R. Auf den RDDs gibt es noch eine weitere Abstraktionsebene, die DataFrames. 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 (**spark**).

Die Interaktion mit einem Spark-Cluster kann über eine interaktive Shell oder einen, in einer der unterstützen Sprachen programmierten, Job 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 mit zusätzlichen Bibliotheken im SparkContext, kann die Unterstützung weiterer Formate hinzugefügt werden. Von den verwendeten Bibliotheken und dem Format 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 einem Dateispeicherort (**spark-website**).

2.1.3 Apache Kafka

Apache Kafka ist ein verteiltes Event-Streaming-System. Die Vermittlung der auftretenden Events läuft in Echtzeit ab. Kafka basiert auf dem nach dem Publish-Subscribe-Modell. Events können von Produzenten veröffentlicht werden und Konsumenten können auf diese Events abonnieren. Durch die Verteilung in einem Cluster kann Kafka den Ausfall einzelner Server ausgleichen. Zusätzlich können Ströme von Events für einen beliebigen Zeitraum gespeichert werden.

Ein Kafka-Cluster (**??**) besteht aus mehreren Brokern, die von Apache Zookeeper verwaltet werden. Die Broker sind für die Verteilung und Speicherung von Events zuständig. Mit den Brokern sind die Client-Anwendungen als Produzenten oder Konsumenten verbunden. Produzenten senden Events an das Cluster und Konsumenten empfangen Events. Ein Event repräsentiert den Fakt, dass etwas „passiert“ ist und besteht aus einem Schlüssel, einem Wert, einem Zeitstempel und optionalen Metadaten. Dabei werden die Werte nicht interpretiert, sondern einfach als Byte-Block versendet und können so eine beliebige Struktur haben.

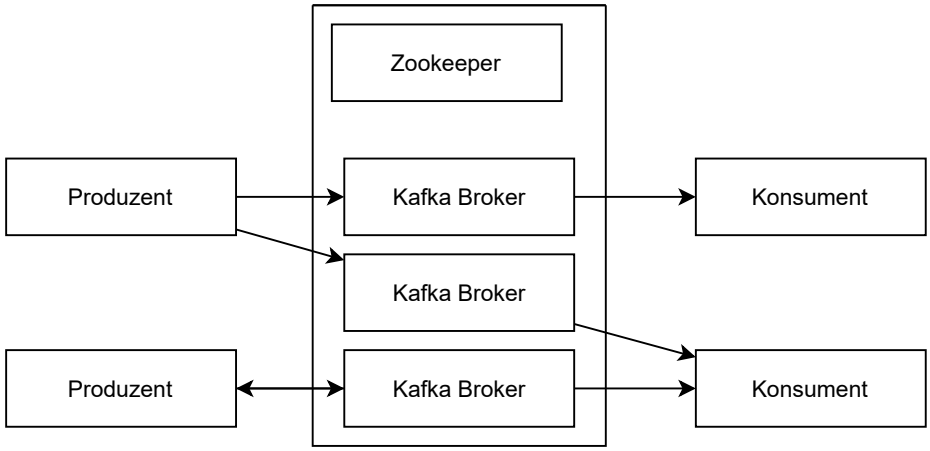


Abbildung 2.2: Kafka-Cluster

Events werden in sogenannte Topics unterteilt. Events in einem Topic können mehrfach gelesen werden und werden nicht nach dem Konsumieren gelöscht. Es kann aber für jedes Topic eine Dauer festgelegt werden, nach der die Events verworfen werden. Um ein Topic fehlertolerant zu machen, kann dieses repliziert werden.

Topics werden in Partitionen über verschiedene Broker aufgeteilt, so dass das ganze System gut skalierbar wird. Ein Produzent kann auch Events auf mehreren Brokern gleichzeitig veröffentlichen. Wenn ein Event in einem Topic veröffentlicht wird, wird dieses an eine der Partitionen angehängt. Events, die den gleichen Schlüssel haben, werden immer der gleichen Partition zugeordnet. Dabei bleibt die Reihenfolge der Events innerhalb einer Partition garantiert erhalten. (**kafka-docs**).

Konsumenten können auch in Gruppen zusammengefasst werden (**??**). Innerhalb einer Gruppe werden Events eines Topic innerhalb der Mitglieder, die dieses Topic abonniert haben, aufgeteilt. Diese Funktion kann zum Beispiel für den Lastausgleich verwendet werden.

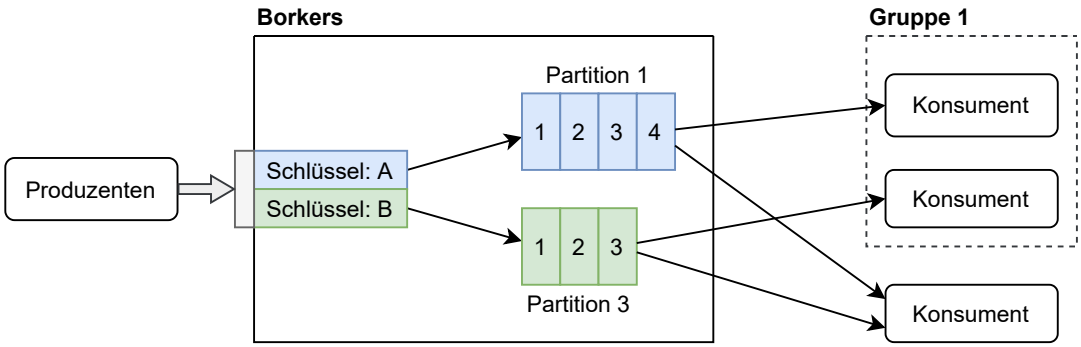


Abbildung 2.3: Kafka-Gruppen

2.2 Data Lake

In der Einleitung (**??**) wurde der Data Lake als Lösung für die Probleme im Big-Data-Bereich bereits kurz beschrieben. In diesem Abschnitt wird noch einmal genauer auf Data-Lake-Systeme, deren Definition, Architektur und existierende Systeme eingegangen.

2.2.1 Definition eines Data Lake

Neben der Definition, die in der Einleitung gegeben wird, wurde von **sawadogo2021data** eine detailliertere Definition für Data Lakes aufgestellt. Hiernach sind Data-Lake-Systeme ein skalierbarer Speicher für Daten jeden Typs. Die Daten werden im Rohformat gespeichert und hauptsächlich durch Datenspezialisten, wie Statistiker oder Analysten, für die Extraktion von Wissen verwendet. Ein Data Lake hat dabei die folgenden Eigenschaften:

1. ein Metadaten-Katalog, um die Datenqualität sicher zu stellen,
2. Regeln und Werkzeuge für die Data Governance,
3. Zugänglichkeit zu den Daten für verschiedene Arten von Benutzern,
4. die Integration von Daten jeden Typs,
5. sowohl eine logische als auch eine physische Gliederung und
6. die Skalierbarkeit von Speicher und Verarbeitung.

2.2.2 Architekturen für Data Lakes

Von **inmon2016data** wird das System in sogenannten Ponds (Teiche) strukturiert. Jeder Pond ist mit einem spezialisierten Speichersystem verknüpft und beinhaltet Daten eines bestimmten Typs. Einige Ponds führen zudem weitere Verarbeitungen der Daten, wie Aufbereitung oder Analysen aus. Inmon hat eine Architektur aus fünf Ponds aufgestellt (**??**).

1. Daten werden im Raw Data Pond im Rohformat gespeichert und fließen von dort aus in andere Ponds. Dieser dient also als Eintrittspunkt in das System für neue Daten. Nach dem Verlassen des Pond werden die Daten aus diesem gelöscht.

2. Der Analog Data Pond enthält analoge Daten, meist von IoT-Geräten. Diese werden hier auf ein aussagekräftiges und verwaltbares Volumen reduziert und umstrukturiert.

3. In den Application Data Pond kommen Daten, die von Software-Anwendungen erzeugt wurden. Diese sind häufig strukturierte Daten aus relationalen Datenbank-Systemen. Sie werden für Analysen integriert und aufbereitet.

4. Der Textual Data Pond enthält unstrukturierte Daten und Prozesse, die deren Analyse erleichtern.

5. Im Archival Data Pond werden alle Daten gespeichert, die nicht mehr aktiv verwendet, aber eventuell in der Zukunft nochmal gebraucht werden.

Ein anderer Ansatz ist die Unterteilung des Data Lake in Zonen (**??**). Hier werden die Daten nach ihrem Verfeinerungsgrad in einer entsprechenden Zone abgelegt. Dabei durchlaufen sie die einzelnen Zonen hintereinander. Die Anzahl der Zonen und deren Verfeinerungsgrad ist dabei je nach Anwendung unterschiedlich (**dl-zones**).

Eine speziellere Architektur ist die Lambda-Architektur, die für die verteilte Verarbeitung von Echtzeit- und Batch-Daten verwendet wird. Eine Lambda-Architektur besteht aus drei Ebenen (Layers) (**lambda-arch**):

1. Die Batch Layer hat zwei Aufgaben. Die erste Aufgabe ist das verteilte Speichern von wachsenden Daten. Dafür kann zum Beispiel das HDFS verwen-

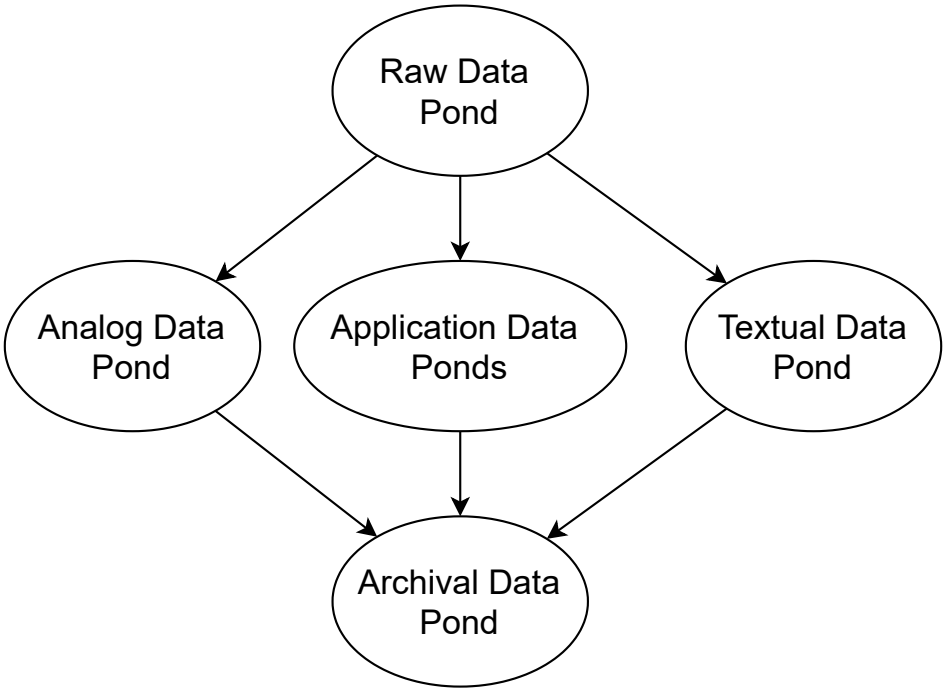


Abbildung 2.4: Ponds-Architektur eines Data Lake nach Inmon

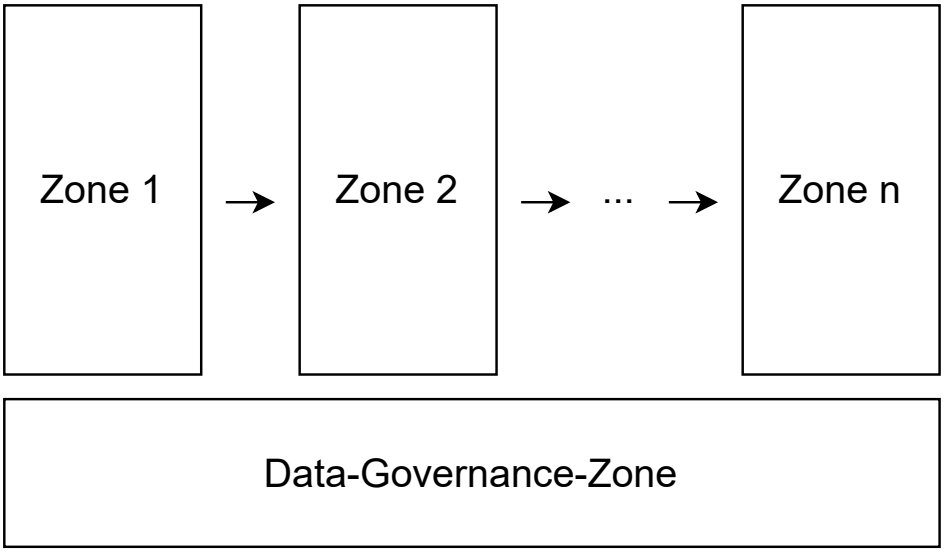


Abbildung 2.5: Prinzip der Zonen-Architektur

det werden. Als zweite Aufgabe werden Batch Views für die verteilten Daten vorberechnet, um Anfragen schneller beantworten zu können.

2. In der Speed Layer werden inkrementell Echtzeit-Views auf Daten verwaltet. Dadurch wird die Lücke gefüllt, die bei den Views in der Batch-Ebene entstehen können. Die Speed Layer enthält immer nur aktuelle Daten. Ältere Daten werden durch die Batch Layer aufgenommen.

3. Die Serving Layer enthält Indices über alle Batch Views um Anfragen mit geringer Latenz bearbeiten zu können. Sie ist dafür verantwortlich die Views aus der Batch und der Speed Layer zusammenzuführen um Echtzeitergebnisse über alle Daten bereit zu stellen.

Nach **sawadogo2021data** können die Architekturen von Data-Lake-Systemen auch anders unterteilt werden. Bei datenorientierten Architekturen wird der Data Lake in verschiedene Datenbereiche unterteilt. Die funktionsorientierten Architekturen dagegen teilen das Data-Lake-System nach den Funktionen auf, die in ähnlichen Bereichen zusammengefasst werden. Ein Beispiel ist die Architektur aus der Einleitung von **datalake_03**. In hybriden Architekturen können auch beide Ansätze kombiniert werden.

2.2.3 Existierende Data-Lake-Systeme und Rahmenwerke

Es gibt bereits verschiedene Rahmenwerke oder Systeme für die Umsetzung eines Data Lake. Nachfolgend wird eine Auswahl daraus vorgestellt.

CoreDB CoreDB ist ein Service, der es erlaubt über eine einzige REST-API Daten und Metadaten in einem Data Lake zu organisieren, zu indizieren und abzufragen. Es können sowohl relationale als NoSQL-Datenbanksysteme mit CoreDB verwendet werden. Für die Suche in den Daten wird elastic[3] verwendet. Das Design von CoreDB unterstützt sowohl Sicherheit und Zugriffskontrolle als auch Verfolgung und Herkunft um überwachende Metadaten sammeln zu können (**coredb**).

[3] https://www.elastic.co/

Azure Data Lake In dem Cloud-Angebot von Microsoft gibt es den Azure Data Lake[4]. Hier werden viele Funktionen, die für den Aufbau eines Data Lake notwendig sind als Cloud-Lösung bereitgestellt. Dazu gehören unter anderem Hadoop, Apache Spark und ein Speichersystem zum Speichern aller Daten. Außerdem gibt es weitere Dienste zur Analyse oder Integration der Daten.

Kylo Kylo[5] ist ein Projekt für eine Data-Lake-Management-Plattform. In dieser Plattform ist eine Ingestion-Komponente enthalten, die die Bereinigung und Validierung von Daten unterstützt. Außerdem gibt es Funktionen für die Aufbereitung und Erkundung von Daten oder zur Systemüberwachung. Zusätzlich ist Apache Nifi[6] zur Erstellung von Verarbeitungs-Pipelines integriert. Die Entwicklung an Kylo wird seit über einem Jahr nicht mehr fortgeführt.

Hudi Apache Hudi[7] ist eine Plattform, um selbst-verwaltete Data Lakes mit einer Optimierung für Datenstromverarbeitung aufzubauen. Zu den Features von Hudi gehört zum Beispiel die Indizierung von Änderung und das Zurückgehen in den Daten zu einem bestimmten Zeitpunkt. Hudi unterstützt sowohl inkrementelle Abfragen als auch Batch-Verarbeitung von Daten.

Bei diesen System fehlen entweder eine ausführliche Metadatenpflege oder eine Daten Versionierung, sie bilden nur einen bestimmten Teil eines Data Lake oder sind für spezielle Anwendungsfälle. Daher wurde bisher kein geeignetes Data-Lake-System für die Anwendung am HIT gefunden

2.2.4 Delta Lake

Als eine Lösung für die Versionierung von Daten gibt es den Delta Lake. Delta Lake ist eine extra Speicherebene, die auf dem HDFS oder einem Objektspeicher in der Cloud, wie Amazons S3, angewendet werden kann. Das Ziel ist es, diesen Speichern ACID-Transaktionen, schnelles Arbeiten mit Metadaten der Tabelle

[4] https://azure.microsoft.com/de-de/solutions/data-lake/
[5] https://kylo.io/
[6] https://nifi.apache.org/
[7] https://hudi.apache.org/

und eine Versionierung der Daten hinzuzufügen. Daten werden in sogenannten Delta-Tabellen mit Metadaten und Logs gespeichert.

Eine Delta-Tabelle wird zunächst durch ein Verzeichnis im Dateisystem dargestellt. Die tatsächlichen Daten werden in diesem Verzeichnis als Parquet-Dateien abgelegt. Dabei können die Daten auch noch in Unterverzeichnisse aufgeteilt werden, zum Beispiel für jedes Datum ein Verzeichnis. Neben den Datenverzeichnissen gibt es in jeder Delta-Tabelle einen Ordner für die Logs in Form von JSON Dateien mit aufsteigender Nummerierung. Metadaten werden sowohl innerhalb der Parquet- als auch in der Log-Dateien gespeichert.

Im Delta Lake wird ein Protokoll für den Zugriff verwendet, dass es mehreren Clients ermöglicht gleichzeitig lesen zu können, aber immer nur einem das Schreiben erlaubt. Dabei werden beim Schreiben immer erst neue Datensätze, die zur Tabelle hinzugefügt werden sollen, in das Verzeichnis der Delta-Tabelle geschrieben. Danach wird eine neue Log-Datei erstellt.

Beim Lesen werden die Log-Dateien als Grundlage verwendet um, daraus zusammen mit den gespeicherten Daten den Zustand der Tabelle an einem bestimmten Zeitpunkt zu erzeugen. Standardmäßig wird beim Lesen immer die aktuellste Version verwendet, man kann aber 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 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 eigenen Server für die Pflege der Delta-Tabellen benötigt. Diese Funktionen werden von den Clients übernommen. Der Delta Lake unterstützt sowohl die Batch-Verarbeitung von Daten als auch Datenströme und bietet volle Integration in Spark (**deltalake**).

2.2.5 Existierender Data-Lake-Prototyp

In einem Masterprojekt an der Hochschule Niederrhein (**prototyp**) wurde ein Prototyp für ein Data-Lake-System entwickelt. In **??** ist ein Überblick über dessen Architektur zu sehen. Es handelt sich hierbei um eine Client-Server-Anwendung.

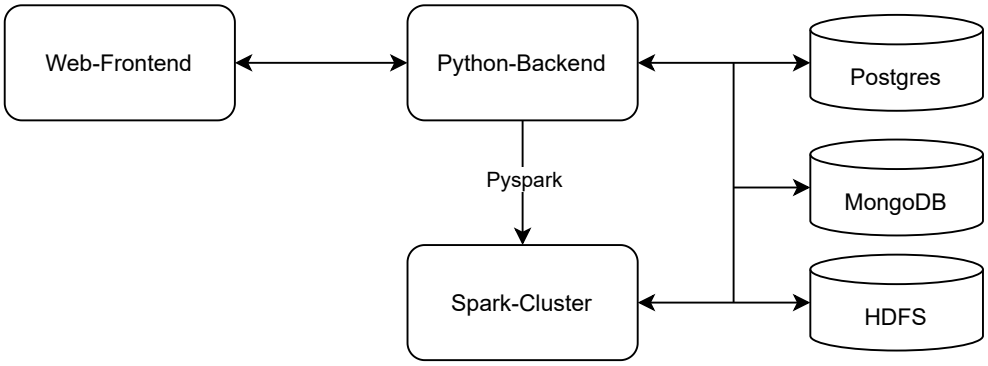
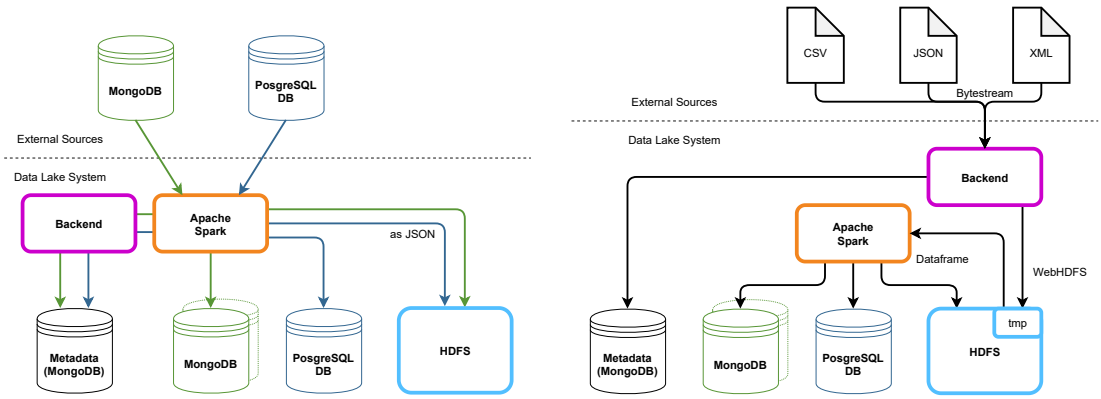


Abbildung 2.6: Architektur des Prototypen

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 Spark-Cluster gelöst. Zum Speichern der Daten stehen drei verschiedene System zu Verfügung. Es kommen eine PostgreSQL Datenbank für strukturierte, eine MongoDB für semistrukturierte und ein HDFS für unstrukturierte Daten zum Einsatz.

Die Verarbeitung der Ingestion ist im Prototyp abhängig von der Datenquelle und dem ausgewählten Zielspeicher. In **??** 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 Verarbeitungen der Quellen zu den Speichern fest im Code des Servers einprogrammiert. So ist es nicht möglich während der Laufzeit neue Datenquellen zu integrieren.

Im Vorlauf dieser Arbeit wurde ein Refactoring des Prototypen durchgeführt. Dabei wurde festgestellt, dass die Erweiterung des Data-Lake-Systems um die Kompatibilität mit weiteren Datenquellen ein aufwändiger Prozess ist. Auch durch die gewählten Speichersysteme für die geladenen Daten erschweren die Integration einer Lösung für die Versionierung der Daten. Daher wurde beschlossen, dass eine dedizierte Ingestion-Schnittstelle für das System entwickelt werden soll.



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

Abbildung 2.7: Ingestion-Verarbeitung des Prototypen, **prototyp**

2.3 Change Data Capture

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 weiterzugeben, damit folgende Verarbeitungsschritte nur die Änderungen berücksichtigen und nicht auf den gesamten Datenbestand zurückgreifen müssen. Dafür gibt es verschiedene Ansätze, die auf Datenbank-Triggern (**boeing**), Log-Einträgen (**delta-view_gen**), Zeitstempeln (**delta-view_gen**; **boeing**) oder Snapshots (**cdc_in_nosql**) basieren.

2.3.1 Änderungserfassung Datenbank-Triggern

Datenbank-Trigger sind Funktionen, 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 Systemen angewendet werden kann, die auch Trigger unterstützen. Dafür ist es möglich alle Änderungen wie Einfügen, Aktualisieren oder Löschen von Daten zu erfassen (**boeing**).

2.3.2 Log-basierte Änderungserfassung

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 zusätzlichen Aufwand für das eigentliche System. Aber auch hier gilt, dass diese Methode davon abhängig ist, ob ein System Logs erstellt und diese durch externe Programme abgerufen werden können (**delta-view_gen**).

2.3.3 Zeitstempel-basierte Änderungserfassung

Ein weiterer Ansatz ist die Verwendung von Zeitstempeln mit den Zeitpunkten der Erstellung und letzten Änderung. Diese Zeitstempel müssen in 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 (**delta-view_gen**). Der Aufwand für diese Methode kann relativ hoch werden, da ohne Indices auf den Zeitstempeln immer die gesamten Daten gelesen werden müssen (**boeing**).

2.3.4 Snapshot-basiert Änderungserfassung

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 werden. 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 (**cdc_in_nosql**).

Kapitel 3

Anforderungen

Basierend auf der Zielsetzung (**??**) können die Anforderungen erarbeitet werden, nach denen die Ingestion-Schnittstelle entwickelt werden soll. Dafür werden nachfolgend die Ziele in einzelne Abschnitte aufgeteilt. In diesen Abschnitten werden die Ziele nochmal genauer erläutert und die einzelnen Anforderungen herausgearbeitet. Auf die Erfüllung der Ziele und Anforderungen wird dann nochmal in der Evaluierung und Zusammenfassung eingegangen.

3.1 Quellen- und Formatunabhängigkeit

Am HIT werden verschiedenste Daten verarbeitet. Viele kommen aus Datenbanksystemen oder Dateien. Es gibt aber auch spezielle Systeme, die ihre Daten nur über eine REST-API zur Verfügung stellen. Alle diese Daten sollen mit der Ingestion-Schnittstelle geladen und gespeichert werden können. Dazu muss diese sowohl Daten entgegennehmen als auch aus Systemen extrahieren können und im Data-Lake-System muss ein Speicher integriert sein, der Daten aus allen Strukturen und Formaten speichern kann. Um eine Flexibilität auch beim Einsatz des Systems in einer anderen Umgebung zu gewährleisten, soll nicht nur der interne, sondern auch externe Systeme zum Speichern der Daten verwendet werden können.

ANF_01 Die Schnittstelle muss in der Lage sein Quelldaten entgegenzunehmen, 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 benötigten Paketen zu erweitern.

ANF_03 Für die Unterstützung verschiedenster Quell- und Zielspeicher verwendet Apache Spark zum Lesen und Speichern einen Format-Parameter und variable Optionen. Diese sollen komplett konfigurierbar sein, um alle Systeme verwenden zu können.

ANF_04 Einige Quellen, zum Beispiel eine REST-API, können nicht direkt über Spark in ein DataFrame gelesen werden. Daher muss eine Möglichkeit gegeben werden, die Ingestion zur Laufzeit um eigenen Programmcode zu erweitern, der ein DataFrame manuell aus einer Reihe von Abfragen aufbaut.

3.2 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 soll 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 wiederholte Ausführung mit einer Zeitsteuerung 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 diesen Fall nicht zulässt.

Wie bereits erwähnt, ist der Prototyp des Data-Lake-Systems eine monolithische Anwendung. Das bedeutet, dass die gesamte Anwendung als Komplettlösung in einem Programm entwickelt und bereitgestellt worden ist. 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, wie von **microservices** 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.

3.3 Datenversionierung

Die Änderungen, die an Daten gemacht werden, sind wichtige Informationen, um weitere Verarbeitungen oder Analysen zu optimieren. Gerade durch die kontinuierliche Ingestion von neuen Daten am HIT entsteht eine Vielzahl solcher Änderungen. Dabei gibt es zwei verschiedene Arten, wie Änderungen in das Data-Lake-System eingespielt werden können. Die erste Art ist die Ingestion von Daten aus einer externen Change-Data-Capture-Lösung (**??**) als Änderungen an einer Datenquelle, die bereits ins System geladen wurde. Bei der zweiten Art wird eine Datenquelle, die bereits geladen wurde, ein weiteres Mal geladen. Hierbei ist es notwendig, das die Ingestion-Schnittstelle die Änderungen zwischen den beiden Ständen erkennen kann. Die Änderungen an den Daten im System sollen über eine Versionierung nachvollziehbar gespeichert werden.

ANF_09 Das System soll eine Möglichkeit bieten, Änderungen an Daten als Versionen zu speichern 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 Änderungsdaten für eine bereits aufgenommene Datenquelle erstellt werden können.

3.4 Datenversionierung

Metadaten spielen eine wichtige Rolle bei der Qualität der geladenen Daten. Die Ingestion-Ebene kann schon beim Laden der Daten Metadaten erfassen. Für diese wird ein Metadatenmodell benötigt, dass alle Metadaten abbildet, die bei der Ingestion erfasst werden können.

ANF_12 Die Ingestion soll beim Laden der Daten Metadaten erfassen.

ANF_13 Das Metadatenmodell soll alle Informationen enthalten, die bei der Ingestion gesammelt werden können.

3.5 Architektur

Die Architektur des Data-Lake-Systems soll in Mircoservices aufgebaut werden. Außerdem wird eine Schnittstelle für die Interaktion mit dem System benötigt. Daraus ergeben sich drei Anforderungen, an die Architektur.

ANF_14 Die Interaktionsschnittstelle mit dem System soll eine REST-API sein, die in das aktuelle Prototyp-System integriert werden soll.

ANF_15 Durch eine klare Trennung der Aufgaben für die Microservices sollen deren Überschneidungen und Abhängigkeiten so gering wie möglich gehalten werden.

ANF_16 Für die Kommunikation zwischen den Microservices soll eine einheitliche Lösung verwendet werden. Diese soll es auch ermöglichen neue Mircoservices einfach in die Architektur einzubringen.

Kapitel 4

Entwurf

Der Ingestion-Prozess ist als erster Schritt im Lebenszyklus der Daten maßgebend für deren Qualität und Aussagekraft bei der späteren Verarbeitung und Analyse (**ingestion_01**). Daher muss schon bei dem Entwurf nicht nur auf die Anforderungen Rücksicht genommen werden, sondern auch auf das fertige Data-Lake-System.

4.1 Architektur

In dieser Arbeit kann kein komplettes Data-Lake-System entworfen werden. Es muss aber eine Entscheidung getroffen werden, nach welcher Architektur das System aufgebaut werden soll. Damit zuverlässig die Änderungen in einer Datenquelle erkannt werden können, dürfen die Daten, die von der Ingestion geschrieben wurden, nicht direkt verändert werden. Hierfür wären sowohl die Zonen- als auch die Ponds-Architektur geeignet. Beide können auch als eine datenorientierte Architektur bezeichnet werden. Für die Implementierung des Data Lake soll eine Microservice-Architektur zum Einsatz kommen. Daraus folgt, dass hier ebenfalls eine Aufteilung nach Funktionen der Komponenten notwendig ist. Daher reicht eine pure datenorientierte Architektur des Data Lake nicht aus. Am besten eignet sich in diesem Fall die von **sawadogo2021data** beschriebene hybride Architektur.

4.1.1 Zonen-Architektur

Von **ingestion_02** wurde eine Architektur basierend auf drei Zonen entwickelt.

1. In der Drop Zone werden alle Daten ohne weitere Verarbeitung gespeichert. Nur die Datenproduzenten haben Zugriff auf diese Zone und können Daten schreiben.

2. Die Landing Zone ist ein unveränderlicher Speicher, der Daten-Assets enthält, die jeweils zu einer Daten-Sammlung gehören. Projekte können mit den Assets aus der Landing Zone interagieren. Benutzer müssen lesenden Zugriff auf bestimmte Assets über einen Governance-Prozess beantragen.

3. Projekte können in der Integration-Zone angereicherte Daten speichern. Diese Anreicherung kann zum Beispiel aus einer Aufarbeitung oder Umstrukturierung bestehen, die speziell für ein Projekt benötigt wird. Es ist außerdem möglich Daten aus der Drop Zone wieder in die Landing Zone zu laden.

Zu diesem Zeitpunkt kann noch keine Aussage darüber getroffen werden, ob diese Architektur für die komplette Umsetzung des Data-Lakes geeignet ist. Der Ansatz der Drop Zone jedoch, in der nur Datenproduzenten schreiben können, erfüllt genau die Bedingung, dass die Daten der Ingestion nicht durch andere verändert werden dürfen. Es ist hier wichtig, dass die Daten mit denen neue Daten verglichen werden, seit der letzten Ingestion nicht verändert wurden, um genaue Aussagen über die geschehenen Änderungen in der Datenquelle treffen zu können. Daher wird zumindest eine Zonen-Architektur mit der Drop-Zone für den Entwurf übernommen. Die Datenproduzenten, in diesem Szenario, sind alle Microservices, die für die Speicherung von Daten zuständig sind. Die Aufteilung in Funktionen kann ebenfalls noch nicht abgeschlossen werden, da in dieser Arbeit die Ingestion-Schnittstelle nur als die erste Funktion entwickelt wird. Weitere Funktionen werden erst im Verlauf der Entwicklung hinzugefügt. Dabei muss aber immer mit überlegt werden, ob neben den Funktionen auch weitere Zonen in der Data-Lake-Architektur hinzugefügt werden.

4.1.2 Microservice-Architektur

Neben der Architektur des Data Lake muss auch eine Architektur für die Microservices entworfen werden. Diese legt fest, welche Komponenten benötigt werden, welche Aufgaben sie bearbeiten und wie sie miteinander interagieren. Der erste Schritt dabei ist es, trennbare Aufgaben zu identifizieren und aufzuteilen. Der Aufbau der Microservice-Architektur kann entweder funktions- oder prozessorientiert angegangen werden.

Bei einem funktionsorientierten Aufbau wird das System in einzelne Funktionen unterteilt. Teile dieser Unterteilung können dann einzelnen Microservices zugewiesen werden. Für die Ingestion-Schnittstelle sind das die Funktionen zum Laden der Daten, für die Deltaberechnung und zur Speicherung der Daten. Da diese aber ein zusammenhängender Arbeitsablauf sind, der in einem Spark-Job ausgeführt werden kann, sollten diese auch nicht auf verschiedene Microservices aufgeteilt werden.

Hier ist der prozessorientierte Ansatz besser geeignet. Dabei werden die technischen Abläufe betrachtet, um eine Unterteilung abzuleiten. Bei der Ingestion-Schnittstelle lassen sich ergeben sich die API, das kontinuierliche Ausführen und die Ausführung der Ingestion mit Spark. In **??** ist die Architektur für die Ingestion-Schnittstelle dargestellt.

Bei dem **API-Service** handelt es sich um den Service für die Interaktion mit dem Data-Lake-System. Durch **??** ergibt sich, dass dieser ein Web-Server mit einer REST-API 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.

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

Bei einer zeitgesteuerten oder Datenstrom-Ingestion muss die kontinuierliche Ausführung sichergestellt werden. Das wird durch den **Continuation-Service** übernommen. Für alle Datenquellen muss regelmäßig geprüft werden, ob für diese gerade eine Ingestion ausgeführt wird und ausgeführt werden sollten. Falls keine Ingestion ausgeführt wird, aber ausgeführt werden sollte, wird die Inge-

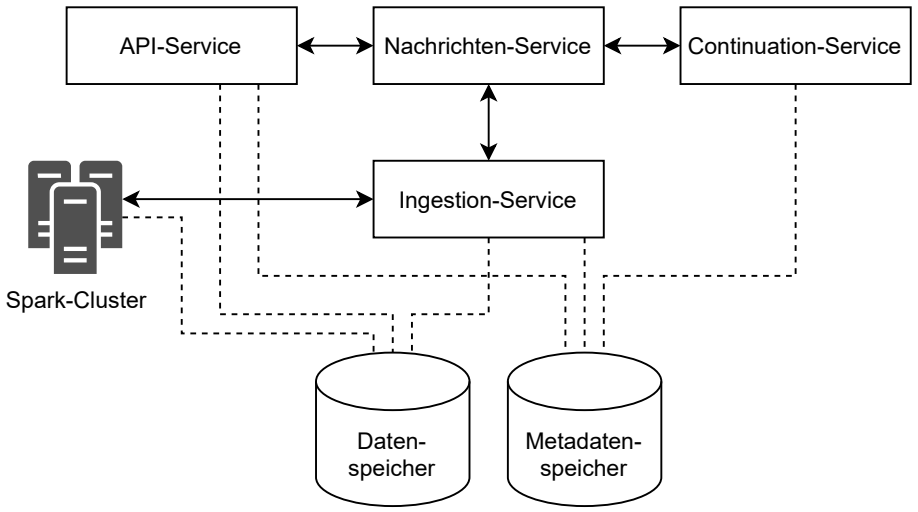


Abbildung 4.1: Microservice-Architektur der Ingestion

stion 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. Für das Speichern von Daten und Metadaten wird jeweils ein Speichersystem benötigt.

4.2 Plugins

In **??** wird gefordert, dass zusätzlicher Code bei der Ingestion ausgeführt werden können soll. Das kann durch Plugins umgesetzt werden. Plugins können einer Datenquelle hinzugefügt und an verschiedenen, fest definierten Punkten ausgeführt werden. 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 beim Ausbau 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 Möglichkeit ist, wie durch die Anforderung gefordert, das Laden von Daten. Genauer bedeutet das, dass das Plugin die Aufgabe übernimmt das DataFrame zu erstellen, welches später wieder gespeichert wird. Das Plugin ersetzt die normale Funktion zum Laden in ein DataFrame. Dies wird zum Beispiel bei der Ingestion von Daten aus einer REST-API benötigt. Hier muss ein DataFrame manuell aus Daten erzeugt werden, die erst über einen REST-Client abgefragt werden.

Als zweite Möglichkeit sollte es Plugins geben, mit denen man nach dem Laden den DataFrame manipulieren kann. Streng genommen widerspricht diese Möglichkeit dem Data-Lake-Prinzip, Daten unverändert in ihrem Rohzustand zu speichern. Allerdings ist zum Beispiel bei der Anbindung von Kafka-Datenströmen eine solche Nachbearbeitung sinnvoll, da die Daten nur als Byte-Block übertragen werden. Für solche Fälle sollte es möglich sein, die unstrukturierten Byte-Daten in ein strukturiertes Format zu überführen.

4.3 Metadatenmodell

Wie bereits erwähnt muss ein Modell erstellt werden, dass die Metadaten abbildet, die bei einer Ingestion erfasst werden (**??**). Zu diesen Metadaten gehören alle Informationen über die Herkunft der Daten, also die Datenquelle und das Speicherziel. Diese werden zum Großteil in für Spark erforderlichen Parametern widergespiegelt. Ein weiterer Teil der Metadaten sind alle Informationen über die Ausführung einer Ingestion.

4.3.1 DatasourceDefinition

Das Metadatenmodell beschreibt eine Datenquelle und hat als zentrales Element das Konzept DatasourceDefinition. Ein Teil des Modells besteht aus Feldern, die für Spark erforderlichen Informationen enthalten. Dazu gehören:

- zusätzliche Abhängigkeiten für Spark (spark.jars.packages),
- das Format und Optionen für die Reader und

- bei benutzerdefinierten Speichersystemen das Format, die Optionen und einen Schreibmodus für die Writer.

Neben den Spark spezifischen werden noch folgende weitere Informationen erfasst:

- ein Name für die Datenquelle,
- die Id einer anderen Datenquelle, falls die aktuelle eine Update-Quelle ist,
- das 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,
- eine Liste der zu laden Dateien, wenn es sich um eine Datei-Ingestion handelt,
- 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 möglichen 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. Und bei einer File-Ingestion werden Dateien hochgeladen, die die Daten enthalten, wobei wichtig ist, dass alle Dateien das gleiche Dateiformat und je nach Typ eine ähnliche oder gleiche Struktur haben. Die Dateien können sich in Daten- und Quelldateien unterscheiden. Datendateien 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 möglichen Schreib-Typen werden aus den Speicherzielen abgeleitet. 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 der DatasourceDefinition werden alle veränderlichen Informationen 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.

4.3.2 IngestionEvent

Das Modell für den Ablauf einer Ingestion ist das IngestionEvent. Dieses enthält, wie die Revision, eine fortlaufende Nummer, ein Start- und Enddatum, den aktuellen Status indem es sich befindet, die Nummer der Revision mit der es gestartet wurde und eine Fehlernachricht, falls ein Fehler aufgetreten ist.

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

4.4 API-Service

Für den API-Service müssen die Endpunkte definiert werden, die den Zugriff auf die verschiedenen Funktionen der Ingestion-Schnittstelle geben. Dazu gehören die Verwaltung von Datenquellen und das Starten einer Ingestion. Da es sich um eine REST-Schnittstelle handelt, können die Endpunkte durch einen Pfad und eine HTTP-Methode definiert werden (**??**). Außerdem kümmert sich der API-Service um die Erstellung von DatasourceDefinitions und deren Revisionen und

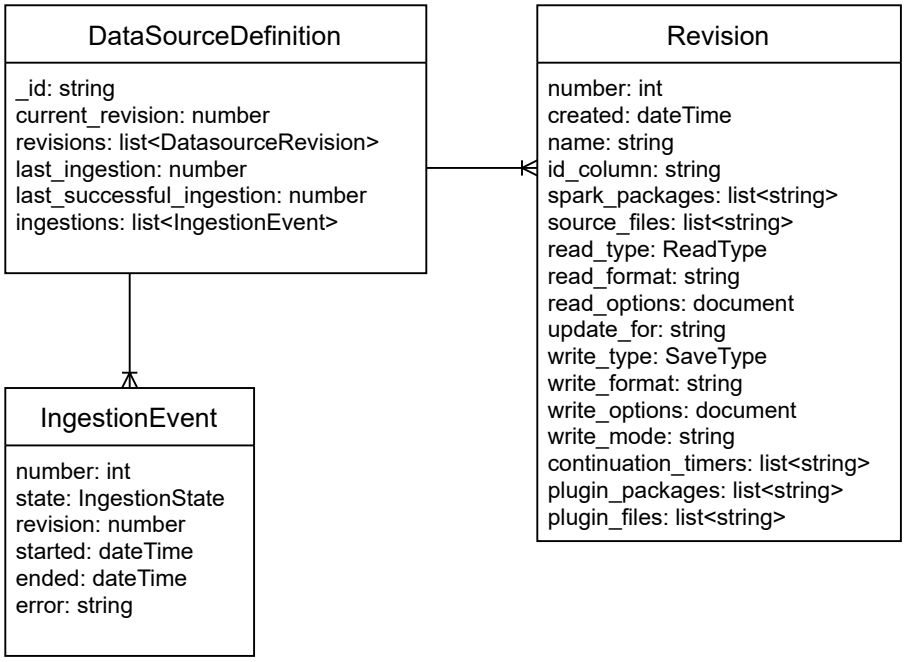


Abbildung 4.2: Übersicht Metadatenmodell

IngestionEvents. Hierbei ist nicht nur die Erstellung der Revisionen wichtig, sondern auch die Handhabung der Quelldateien. Diese müssen so abgelegt und in der Revision verwaltet werden, dass Spark später bei der Ingestion darauf zugreifen kann, aber bereits hochgeladenen Dateien nicht gelöscht werden. Bei Anfragen zum Starten einer Ingestion versendet der API-Server eine Nachricht mit der Id einer Definition der Datenquelle, die geladen werden soll.

4.5 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 ausgeführt werden muss. Bei Datenströmen gilt allgemein, wenn dieser nicht läuft, dann muss die Ingestion automatisch neu gestartet werden. Die einzige Ausnahme ist, wenn die Ingestion durch den Benutzer explizit beendet wurde.

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 4.1: Endpunkte des API-Servers

Der zweite Fall ist die Zeitsteuerung. Für eine zeitgesteuerte kontinuierliche Ingestion werden einer Datenquelle ein oder mehrere Timer hinzugefügt. Der Continuation-Service kontrolliert, ob der Timer zum Zeitpunkt, an dem die Datenquelle überprüft wird, zutrifft oder nicht. Wenn das der Fall ist und bisher keine Ingestion auf der Datenquelle läuft, dann wird eine neue Ingestion 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 und mehrere Werte als Liste mit Kommata getrennt angeben oder mit „/x“ eine Liste in Schritten der Größe x erzeugen.

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.

4.6 Ingestion-Service

Der Ingestion-Service hat die Aufgabe den Spark-Job für die Ausführung einer Ingestion zu erstellen, zu starten, zu überwachen und den Status des IngestionEvents anzupassen. Dazu gehört das Festlegen des Ablaufs zum Laden der Daten in ein DataFrame, zur Deltaberechnung und zum Speichern. Ein zweiter wichtiger Teil ist die Integration der Plugins in den Ingestion-Prozess. Ebenfalls koordiniert der Ingestion-Service die parallele Ausführung von Ingestions.

Der Ingestion-Service wartet auf die Nachricht zur Ausführung einer Ingestion, mit der Id der DatasourceDefinition. Als erstes wird 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 wird die Parallelität ermöglicht.

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

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

Wurden Änderungsdaten berechnet, werden diese eingepflegt. Wurden keine berechnet, werden die Daten einfach gespeichert. Ist das Speicherziel dabei ein Delta-Tabelle wird diese angelegt und ist es eine Parquet-Datei, dann werden die aktuellen Daten überschrieben. Handelt es sich nicht um einen benutzerdefinierten Speicher, werden die alten Daten mit den neuen überschrieben.

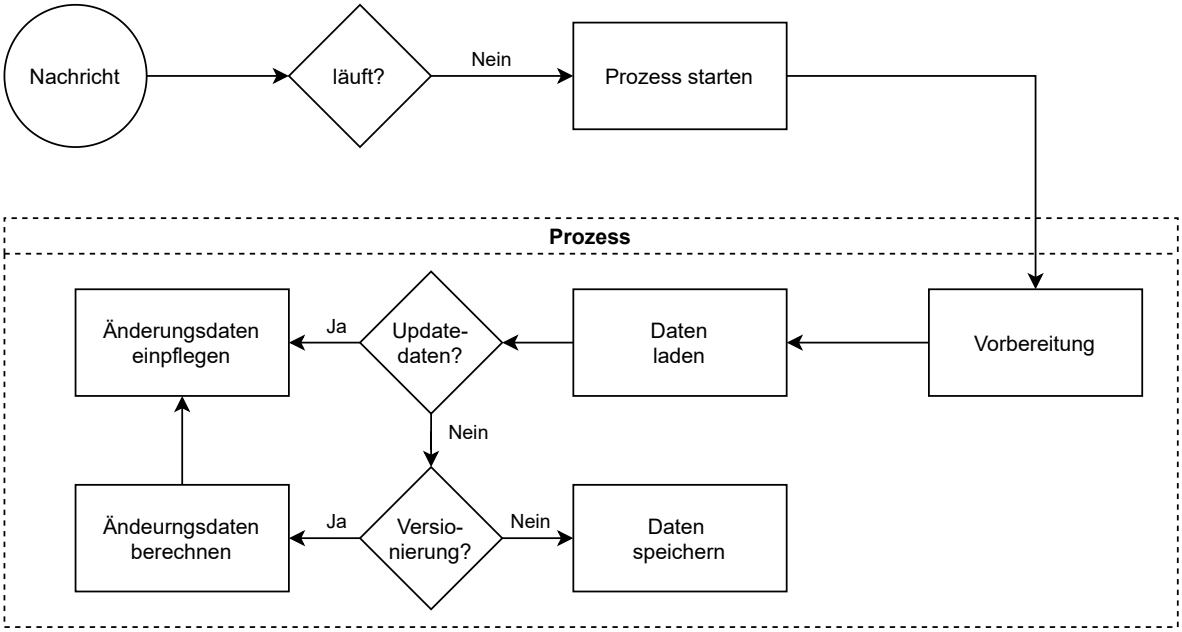


Abbildung 4.3: Ablauf einer Ingestion

Kapitel 5

Implementierung

In diesem Kapitel wird die Implementierung der Ingestion-Schnittstelle besprochen. In **??** ist die Systemarchitektur zu sehen. Diese zeigt alle Komponenten der Schnittstelle und wie diese interagieren. In den folgenden Abschnitten wird deren Funktionsweise erklärt. Dabei wird darauf eingegangen, welche Techniken angewendet wurden und warum diese gewählt wurden. Bei den implementierten Mircoservices wird auch auf entwickelte Algorithmen und Vorgehensweisen eingegangen, mit den bestimmte Aufgaben gelöst werden. Die Beschreibungen gehen dabei nicht zu sehr ins Detail. Hier soll eher vermittelt werden, wie ein Problem gelöst wurde, unabhängig von gewählten Rahmenwerken oder Bibliotheken, solange diese nicht notwendig für die Lösung des Problems sind. Für den API-, Ingestion- und Continuation-Service werden Konfigurationsdateien im YAML-Format[1] verwendet. Über diese werden Verbindungsinformationen und service-übergreifende Parameter konfiguriert. Am Ende wird auch auf die Bereitstellung eingegangen. Das heißt wie das fertige System gestartet und verwendet werden kann.

5.1 Programmiersprache

Da Apache Spark Implementierungen für Java, Python und Scala bereitstellt, sind das auch die drei Sprachen, auf die die Auswahl begrenzt ist. Der Prototyp ist in

[1] https://yaml.org/

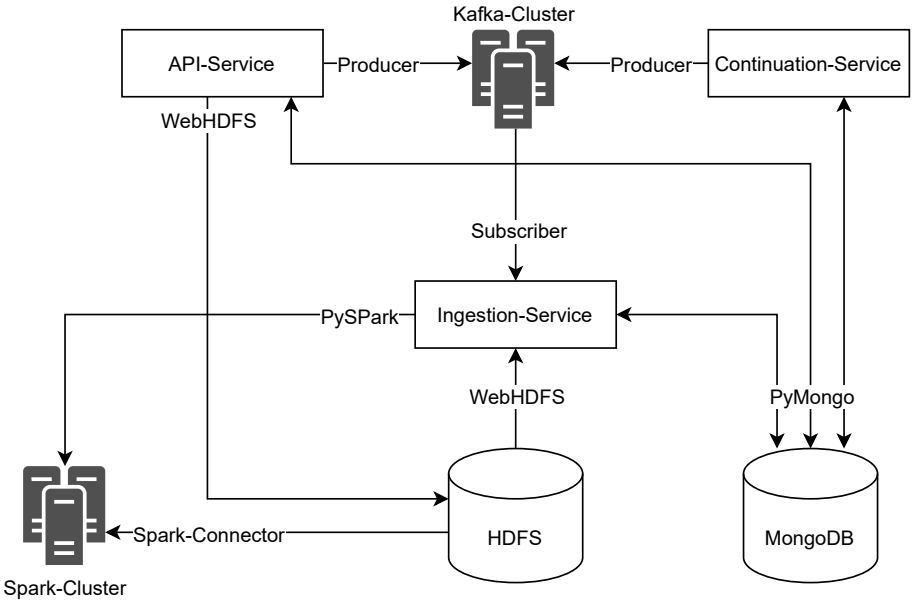


Abbildung 5.1: Implementierte Systemarchitektur

Python geschrieben und da dieser zum API-Service erweitert werden soll, wird Python auch weiter für die Implementierung des API-Service verwendet. Auch für die Implementierung des Ingestion-Service bietet Python einige Vorteile.

Bei der Verwendung von Java und Scala wird ein fertig kompilierter Job über den Befehl „spark-submit“ zur Ausführung an Spark gesendet. Der Spark-Job muss also vor dem Submit bereits kompiliert worden sein. Daraus folgt, dass es zwei Möglichkeiten gibt, wie der Ingestion-Service damit umgeht. Zuerst wäre es möglich, dass alle Jobs, die möglicherweise benötigt werden, dem Ingestion-Service bereits kompiliert vorliegen. Die zweite Möglichkeit wäre, dass die Jobs dynamisch angepasst und immer vor der Ausführung durch den Ingestion-Service kompiliert werden. Beides würde einen erheblichen Aufwand bei der Entwicklung bedeuten, da entweder ein sehr generischer Job erstellt werden müsste, der alle Ingestion-Fälle abdecken kann, oder es müsste Logik eingebaut werden, die die Quell-Dateien des Jobs bearbeitet und diesen dann kompiliert.

In Python gibt es zusätzlich dazu auch die Möglichkeit, dem Interpreter die Übersetzung zur Laufzeit zu überlassen. So ist es möglich für jede Anfrage speziell konfigurierte Jobs zu erstellen, die nicht vorher schon kompiliert werden

und somit feststehen müssen. Das senkt die Komplexität bei der Entwicklung der Ingestion (**pyspark-int**).

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 alle Abhängigkeiten erfüllt, also die notwendigen Bibliotheken auf dem Ingestion-Service installiert sind.

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

5.2 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 das Senden und Empfangen asynchron und damit unabhängig voneinander stattfindet (**message-broker**).

Es gibt mittlerweile einige Projekte, die diese Aufgabe auf verschiedene Art lösen. Hier wird 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 bereitzustellen. Um das System dabei nicht unnötig komplex und zu groß werden zu lassen, wird daher auf einen anderen Message-Broker verzichtet. Außerdem ist die Verteilung der Nachrichten durch Kafka sehr flexibel gestaltbar.

Da Kafka ein Event-Streaming-System ist, wird ab hier nicht mehr von dem Austausch von Nachrichten, sondern von Events gesprochen. Für diese Events müssen Topics zur Einordnung festgelegt werden. Die Namen sollten dabei so gewählt werden, dass Kafka auch für andere Datenströme verwendet werden kann, ohne, dass es zu Konflikten kommt. Daher werden die Topics der internen Kommunikation des Data-Lake-Systems immer mit „dls__“ als Präfix benannt werden. Danach folgt der Bereich, den das Event betrifft, hier zum Beispiel „ingestion“. Zu diesem Präfix können noch weitere Informationen hinzugefügt werden. Für das Ausführen einer Ingestion ist nach diesem Aufbau das Topic

„dls__ingestion__run“.

Im Ingestion-Service können sehr gut die Gruppen bei Kafka-Konsumenten angewendet werden. Dazu werden die Konsumenten jedes Ingestion-Service der selben Gruppe hinzugefügt. So bekommt immer nur ein Ingestion-Service das Event zum Starten einer Ingestion und es muss keine weitere Logik implementiert werden, die sicherstellt, dass nicht mehrere Ingestion-Services eine Ingestion zur gleichen Quelle starten. Das macht die Skalierung und Verteilung der Ingestion-Services sehr einfach.

5.3 Metadaten-Management-System

Da das Metadatenmodell einer Datenquelle eine verschachtelte Struktur hat, bietet sich hier als einfachste Lösung die Verwendung einer dokumenten-orientierten NoSQL-Datenbank an. Das hat den Vorteil, dass die Sammlungen von Revisionen und IngestionEvents direkt in den Objekten der DatasourceDefinition abgelegt werden können. In relationalen Datenbank-Systemen müsste man für jedes Modell eine eigene Tabelle erstellen und die Verknüpfungen über mehrere Abfragen zusammenführen. Bei jeder Abfrage einer Datenquelle werden die verknüpften Einträge der IngestionEvents 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 kann auch wieder eine Komponente aus dem Prototyp übernommen werden. Deswegen kommt MongoDB[2] als Datenbank-System zum Einsatz.

5.4 Datenspeicher

Für das Speichern von Daten mit einer Versionierung ist Delta Lake eine gut gepflegte und in Spark integrierte Lösung. Da im Delta Lake Datensätze immer versioniert werden, muss auch ein Speicher für unversionierte Daten bereitgestellt werden. Für strukturierte und semistrukturierte Daten kommt das Parquet-

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

Format zum Einsatz. Unstrukturierte Daten werden im ursprünglichen Format im HDFS abgelegt.

Als Speicher wird das HDFS verwendet. In diesem können sowohl die Delta Tabellen als auch andere Parquet-, Quell- und Plugin-Dateien abgelegt werden. Alle Microservices haben über die WebHDFS-Schnittstelle Zugriff auf die Daten. Im HDFS wird eine Verzeichnisstruktur (**??**) für alle Daten des Data Lake angelegt. Diese geht von dem Ordner „datalake“ aus, der im Root-Verzeichnis des HDFS angelegt wird. In diesem Ordner werden die Unterordner

- „sources“ für Quell-Dateien,
- „plugins“ für Plugin-Dateien und
- „data“ für die Ablage von geladenen Daten erstellt.

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 werden die Ordner

- „structured“ für semi-/strukturierte Daten ohne Versionierung,
- „unstructured“ für unstrukturierte Daten und
- „delta“ für semi-/strukturierte Daten mit Versionierung

innerhalb des Ordners „data“ angelegt. Diese unterteilen sich dann ebenfalls wieder in Ordner für jede Datenquelle mit der Id als Name.

5.5 API-Service

Bei HTTP-Anfragen können Inhalte in der Anfrage als verschiedene Typen übergeben werden. Hier wird der Typ `mutltipart/forma-data` verwendet. Bei diesem Typ können sowohl Text als auch Dateien übertragen werden. Jeder Text oder jede Datei werden dabei als Wert betrachtet und müssen einen Schlüssel vergeben bekommen. Das heißt, dass alle Informationen als Schlüssel-Wert-Paar

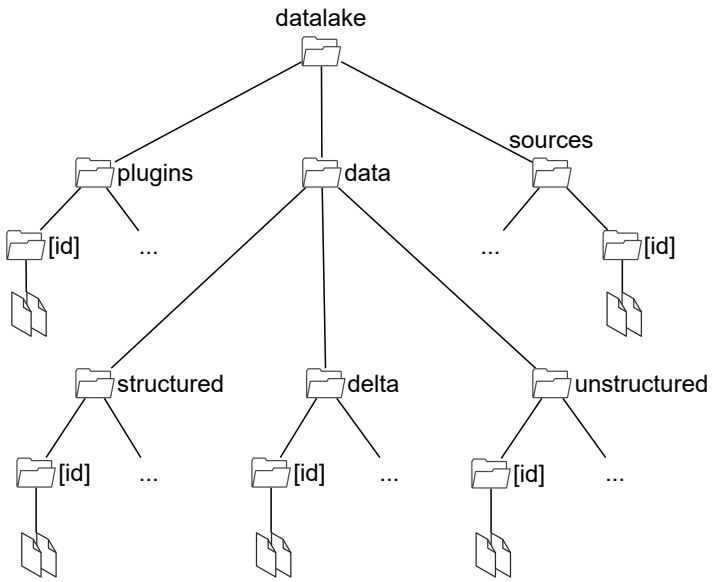


Abbildung 5.2: HDFS Verzeichnisstruktur

an den API-Service gesendet werden. Die Schlüssel, die verwendet werden sollen, werden wie folgt vergeben. Die DatasourceDefinition bekommt den Schlüssel „datasource-definition“. Der Wert dahinter kann entweder eine JSON-Datei oder ein JSON-Text sein. Die Schlüssel der hochgeladenen Dateien sind frei wählbar.

Die DatasourceDefinition besteht zu Teilen aus Feldern, die automatisch durch die Services gefüllt werden. Daher wird ein weiteres Datenmodell für die Eingabe von Informationen bei der REST-API benötigt. Dafür wird die DatasourceDefinitionInput (**??**) verwendet. In diesem Modell befinden sich alle Felder, die durch den Benutzer befüllt werden können. Aus den Daten dieses Modells erstellt der API-Service dann die Revisionen für die DatasourceDefinition.

5.5.1 Hochladen von Dateien

Für jede Datenquelle können Quell- oder Plugin-Dateien hochgeladen werden. Um eine hochgeladene Datei in der Datenquelle auch zu verwenden, muss der Schlüssel, unter dem die Datei hochgeladen wird, in der entsprechenden Liste entweder unter „source_files“ oder unter „plugin_files“ hinzugefügt werden. Al-

DatasourceDefinitionInput
name: string
id_column: string
source_files: Array<string>
read_type: string
read_format: string
read_options: Dict<string, string>
update_for: string
write_type: string
write_format: string
write_options: Dict<string,string>
write_mode: string
continuation_timers: Array<string>
plugins_files: Array<string>
plugin_packages: Array<string>

Abbildung 5.3: Felder Datenquellen-Eingabe

ternativ können auch die Namen unter denen bereits Dateien für die Datenquelle gespeichert wurden in die Listen eingefügt werden.

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. Der Name, unter dem die Datei gespeichert wird, setzt sich aus der Nummer der Revision, dem vergebenen Schlüssel und der Endung der Datei zusammen. Lädt man also eine bei der ersten Erstellung einer DatasourceDefinition eine Datei „data.json“ mit den Schlüssel „file“ hoch, wird diese als „r000_file.json“ im „sources“-Ordner der DarasourceDefinition gespeichert. Analog werden Plugin-Dateien genauso behandelt.

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. Es ist wichtig zu beachten, dass alle Dateien der alten Revision, die nicht explizit in einer der Listen angegeben werden, nicht in der neuen Revision verwendet werden. Sie bleiben jedoch gespeichert und können später wieder hinzugefügt werden. Da die Dateien nach den Datenquellen aufgeteilt sind, ist es aktuell nicht möglich Plugins oder Quell-Dateien in anderen Datenquellen wieder zu verwenden. Hier müsste man die Datei für jede Datenquelle hochladen.

5.6 Continuation-Service

Der Continuation-Service führt durchgehend eine Schleife zum Überprüfen der Datenquellen aus. Bei der Prüfung wird immer der Status des aktuellsten IngestionEvent betrachtet. Je nach Datenquelle wird dann entschieden, ob eine Ingestion gestartet werden soll. Die Logik der Schleife besteht aus drei Teilen, wovon zwei die Datenquellen prüfen und einer die Zeit steuert, nachdem die Schleife erneut ausgeführt wird (**??**). Das soll nur maximal jede Minute geschehen, da auch keine schnellere Ausführung über die Timer definiert werden kann.

Im ersten Teil werden alle Datenquellen einer Datenstrom-Ingestion kontrolliert. Für diese wird eine neue Ingestion gestartet, wenn der Status „FINISHED“ ist. Der Status „STOPPED“ bedeutet, dass die Ausführung mit Absicht beendet

wurde und manuell gestartet werden muss.

Der zweite Teil überprüft alle Datenquellen, bei denen ein oder mehrere Timer gesetzt wurden. Wenn der Status des letzten IngestionEvents nicht „STOPPED“ oder „FINISHED“ ist, wird die Überprüfung dieser Quelle abgebrochen. Ansonsten wird jeder Timer mit dem aktuellen Zeitpunkt verglichen. Hier wird die Methode is_now verwendet. Sie wird durch eine Python-Bibliothek bereitgestellt und prüft, ob ein Cron-Timer, die auch für die kontinuierliche Ingestion eingesetzt werden, zum aktuellen Zeitpunkt zutrifft. Trifft der Timer zu, wird eine Ingestion für die Datenquelle gestartet und sofort die nächste Datenquelle überprüft.

Der dritte Teil kontrolliert die Ausführungshäufigkeit. Dazu wird am Anfang der Schleife der Startzeitpunkt gespeichert. Nachdem alle Überprüfungen beendet wurden, wird auch der Endzeitpunkt gespeichert. Wenn die Differenz der beiden geringer ist als eine Minute, wird die Schleife für die Dauer der Differenz angehalten. Damit ist sichergestellt, dass sie nicht häufiger als einmal pro Minute ausgeführt wird.

5.7 Ingestion-Service

Wie in **??** beschrieben, wird für eine Ingestion ein Prozess gestartet. Zum Starten einer Ingestion wird das Topic „dls__ingestion__run“ verwendet. In **??** wird der detailliere Ablauf mit den verschiedenen Wegen für Read- und SaveTypes beschrieben. Vorher werden die benötigten Grundlagen zur Ausführung erläutert.

5.7.1 Berechnen und Speichern der Änderungsdaten

Um Änderungen in eine Delta-Tabelle zu übernehmen, 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 Schema der Originaldaten entsprechen. Zusätzlich soll eine Spalte oder ein Feld auf der obersten Ebene mit dem Namen „cd_deleted“ vorhanden sein. Darin ist ein Boolean-Wert enthalten, der aussagt, ob der Eintrag aus den Daten gelöscht wurde oder nicht. Der Datensatz mit den Änderungsdaten darf außerdem nur geänderte Daten enthalten. Aus diesem Format können,

Algorithmus 1: Continuation loop

```
Data: -
Result: -
loopStart ← current_time() ;
streams ← all_datasource_definition_of_type_stream() ;
forall definition in streams do
    event ← definition.last_ingestion ;
    if event.state is FINISHED then
        publish_run_event_for(definition.id) ;
    end
end
timed ← all_datasource_definitions_with_timers() ;
forall definition in timed do
    event ← definition.last_ingestion ;
    timers ← definition.revision.continuation_timers ;
    if event.state is STOPPED or FINISHED then
        forall timer in timers do
            if is_now(timer) then
                publish_run_event_for(definition.id) ;
                break
            end
        end
    end
end
loopEnd ← current_time() ;
loopDuration ← loopEnd − loopStart ;
if loopDuration < 60 then
    sleep(60 − loopDuration) ;
end
```

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. Dazu eignet sich der in **??** genannte snapshot-basierte Ansatz. Dieser ist zwar langsamer als alle anderen, aber auch als einziger unabhängig von der Datenquellen und innerhalb des Data-Lake-Systems anwendbar.

snapshot_algos haben einen Algorithmus vorgestellt, der diese Aufgabe mit der Hilfe von Join-Operationen löst. Für den Vergleich werden für jeden Eintrag eindeutige Schlüssel benötigt. Das Feld „id_column“ in der DatasourceDefintion enthält den Namen der Spalte oder des Feldes, das als Schlüssel fungieren soll. Es können auch verschachtelte Felder verwendet werden. Der Schlüssel kann aber nicht aus mehreren Feldern zusammengesetzt werden. Die Namen der Felder in den einzelnen Ebenen müssen dafür mit einem „.“ getrennt werden. Beispielsweise „user.id“ für:

```
{
    user: {
        id: x
    }
    ...
}
```

Join-Algorithmen sind in der Informatik bereits vielfach besprochen worden. Wenn man die Einträge beider Snapshots über ihren Schlüssel verknüpft, kann man so durch einen Vergleich der Ergebnisse alle geänderten Einträge finden. In einem Full-Outer-Join sind zusätzlich alle Einträge vorhanden, die nur in einem der beiden Einträge vorhanden sind. Die restlich Felder bekommen einen Null-Wert. Je nachdem auf welcher Seite die Null-Werte stehen, handelt es sich um eine Einfügung oder Löschung. Da durch SparkSQL eine gute Unterstützung für Join-Operationen gegeben ist, ist dieser Ansatz auch gut für den Einsatz im Data Lake geeignet. Dieses Vorgehen funktioniert sowohl für relationale als auch semistrukturierte Daten. Bei dem Vergleich von semistrukturierten Daten wird jedoch nur die oberste Ebene als Spalte betrachtet. Alle darunter verschachtelten Objekte werden als Werte dieser Spalten betrachtet.

Der **??** zum Vergleich von DataFrames orientiert sich an diesem Vorgehen.

Als Eingabe werden ein linkes DataFrame, mit dem aktuellen internen Stand, eine rechtes DataFrame, mit den neuen Daten und der Name des Schlüssels benötigt. Um den Vergleich der Zeilen später einfacher zu machen, wird zuerst für jede Zeile der zu vergleichenden Datensätze ein Hashwert über alle Spalten berechnet. Außerdem wird eine weitere Spalte mit dem Namen „~id“ hinzugefügt, die den Wert des Schlüssels zugewiesen bekommt. Da wie bereits erwähnt bei semistrukturierten Daten nur die oberste Ebene als Spalten betrachtet wird, ist es so erst möglich Einträge über verschachtelte Schlüssel zu verknüpfen. Danach wird ein Full-Outer-Join über den Daten ausgeführt. Im nächsten Schritt werden alle Zeilen, in denen die Hashwerte gleich sind aus dem Datensatz entfernt, da diese keine Relevanz für die Änderungsdaten haben. Zu den gefilterten Daten wird nun die Spalte „cd_deleted“ hinzugefügt. Ist in den alten Daten eine Zeile vorhanden und in den neuen nicht mehr, so ist der Hashwert auf der rechten Seite $Null$. In diesem Fall bekommt die Spalte „cd_deleted“ den Wert $true$ für alle anderen ist der Wert $false$. Da das Ergebnis der Join-Operation alle Spalten doppelt enthält, einmal von links und einmal von rechts, müssen diese noch zusammengeführt werden. Dazu können immer, außer bei dem Schlüssel, die Werte der rechten Seite genommen werden, da diese die neuen Werte enthält. Der Wert für den Schlüssel wird auch von der rechten Seite übernommen, es sei denn dieser ist $Null$ (bei gelöschten Zeilen), dann muss der linke Wert verwendet werden, da der Schlüssel nicht $Null$ sein darf. Zum Schluss müssen die zusätzlich für den Vergleich hinzugefügten Spalten wieder entfernt werden. Das Ergebnis ist ein DataFrame mit allen Änderungen. Es hat das gleiche Schema wie die Ursprungsdaten, aber mit der zusätzlichen Spalte „cd_deleted“, die Auskunft darüber gibt, ob ein Datensatz gelöscht wurde.

Das Einpflegen der Änderungen unterscheidet sich nach dem gewählten Speicherziel. Für Delta-Tabellen wird die Delta Lake API verwendet. Dazu werden die Änderungsdaten mit den aktuellen Daten über den Schlüssel zusammengeführt. Dabei können verschiedene 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, wird der Datensatz als neue Zeile eingefügt.

Für Parquet-Dateien werden die Änderungsdaten über Spark mit den Be-

Algorithmus 2: Deltaberechnung

```
Input: left: DataFrame, right: DataFrame, id_column: String
Output: change_data: DataFrame
forall df in [left, right] do
    df.add_column(name: „hash“, value: df.hash_over_all_rows());
    df.add_column(name: „~id“, value: df.get_value(id_column));
end
change_data ← left.join(right, type: full-outer, column: „~id“);
change_data.remove_all_row(condition: left.hash equals right.hash);
change_data.add_column(name: „cd_deleted“, value: false);
forall row in change_data.rows do
    if row.right.hash is Null then
        row.cd_deleted ← true;
    end
end
forall column in left.columns do
    if column is id_column then
        forall row in change_data.rows do
            if row.right.hash is Null then
                row.id_column ← row.left.id_column;
            else
                row.id_column ← row.right.id_column;
            end
        end
    else
        forall row in change_data.rows do
            row.column ← row.right.column;
        end
    end
    change_data.remove_column(right.column);
    change_data.remove_column(left.column);
end
change_data.remove_column(„~id“);
change_data.remove_column(„hash“);
```

standsdaten zusammengeführt. Beide Datensätze werden über einen Full-Outer-Join verknüpft. Danach werden alle Zeilen, in denen das Feld „cd_deleted“ den Wert $true$ enthält, gelöscht. Ähnlich zu der Deltaberechnung werden die übrigen Daten wieder auf das Originalschema gebracht. Dabei werden für alle Zeilen, bei denen entweder kein Eintrag aus den Bestandsdaten existiert oder die Änderungsdaten sich von ihnen unterscheiden, die Werte der Änderungsdaten übernommen. In den anderen Zeilen werden die Bestandsdaten beibehalten.

5.7.2 Plugin-Management

Da Python verwendet wird, kann nicht, wie zum Beispiel in Java, ein Interface definiert werden, dass ein Plugin implementieren muss. Es ist aber möglich die Namen, Parameter und Rückgabetyp einer Methode zu überprüfen. Mit diesem Ansatz können Methoden aus den hochgeladen Plugin-Dateien validiert werden. Für die Ingestion werden diese zwei Methoden definiert:

1. Load-Methode:

```
Name: "load"
Rückgabetyp: DataFrame
Parameter:
    - Name: "spark"
      Typ: SparkSession
```

2. After-Load-Methode:

```
Name: "after_load"
Rückgabetyp: DataFrame
Parameter:
    - Name: "dataframe"
      Typ: DataFrame
```

Für jede Ingestion wird auf dem Speichersystem des Mircoservices ein temporärer Ordner angelegt, in den die Plugin-Dateien abgelegt und deren Abhän-

gigkeiten installiert werden. Nach der Installation wird noch eine Datei angelegt, die für alle Pakete die Versionsnummern enthält. Das dient dazu, bei einer erneuten Ausführung auf dem Service nicht alle Pakete neu installieren zu müssen, sondern nur die mit einer geänderten Versionsnummer. Das beschleunigt die Ausführung der Ingestion. Zum Schluss wird der Ordner, mit den installierten Paketen, zum Python-Pfad hinzugefügt. Damit wird dieser während der Ausführung manipuliert und die Pakete sind verfügbar. Da jede Ingestion in einem eigenen Prozess ausgeführt wird, beeinflussen die Änderungen an dem Python-Pfad den Ingestion-Service oder andere Prozesse nicht.

Im zweiten Schritt wird jede Plugin-Datei als Python-Modul geladen. Jedes Modul enthält dann die in der Plugin-Datei definierten Methoden. Diese Methoden können mit den Definitionen verglichen werden. Dazu wird der **??** verwendet. Diesem werden das geladene Modul, ein Name der Methode, ein optionaler Rückgabetyp der Methode und eine Liste von Parametern, bei denen Name und Typ definiert sind. 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 die Load- oder AfterLoad-Methoden geprüft. Eine gefundene Load-Methode überschreibt immer die vorher gefundene, da jede Ingestion nur einen Weg zum Laden der Daten haben darf. Die After-Load-Methoden dagegen werden in einer Liste gespeichert. Es kann jedoch immer nur eine After-Load-Methode pro Plugin-Datei geben. Diese können dann auch hintereinander ausgeführt werden, um auf einem DataFrame mehrere Modifikationen auszuführen. Alle gefunden Methoden werden in einem Plugin-Manager gespeichert, um während der Ingestion aufgerufen werden zu können. Die Reihenfolge der Ausführung entspricht der Reihenfolge, wie die Plugin-Dateien bei der Erstellung in der Liste angegeben wurden.

5.7.3 Ausführung der Ingestion

Die Ausführung startet mit der Initialisierung, welche 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 notwen-

Algorithmus 3: Pluginmethode überprüfen

Input: *plugin*: Python-Modul, *name*: String, *return_type*: Typ, *parameters*: Liste
Output: *matches*: Boolean
if *plugin* ***has no*** *method name* **then**
  return *false*
end
signature ← signature of mehtod *name*
if *signature.return_type* ***is not*** *return_type* **then**
  return *false*
end
forall *param* ***in*** *parameters* **do**
  if *signature* ***has no*** *parameter param.name* **then**
    return *false*
  end
  if *signature.parameter.type* ***is not*** *parameter.type* **then**
    return *false*
  end
end
return *true*

dig ist. Hier spielt der Lesetyp eine große Rolle. Bei der Ingestion von unstrukturierten Daten wird Spark nicht benötigt. Hier können die Dateien, über die WebHDFS-Schnittstelle direkt im HDFS verschoben werden. 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 anschließend entweder über ein Plugin in oder das Standardvorgehen. 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 Änderungsdaten gespeichert. Für Datenströme wird am Ende noch ein Hintergrund Task gestartet, der auf Kafka Events zum Stoppen dieser Ingestion wartet. Hierfür wird das Topic „dls__ingestion__stop_ingestion“ mit der Id als Wert verwendet. Wenn die Ingestion beendet wurde, wird zum Schluss die SparkSession gestoppt.

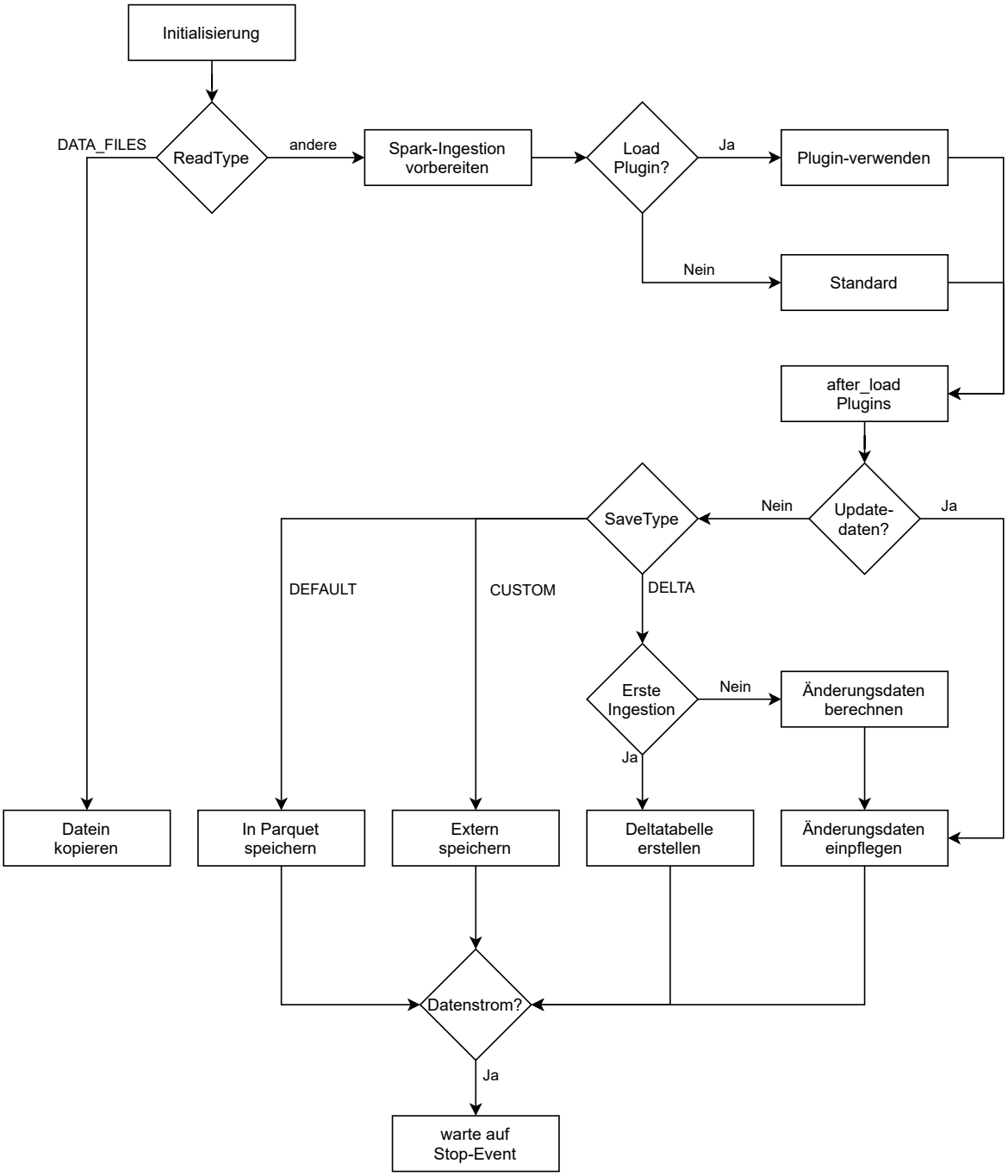


Abbildung 5.4: Ablauf einer Ingestion

5.8 Bereitstellung

Bei der Bereitstellung des gesamten Systems gibt es einen wichtigen Punkt, der beachtet werden muss. Viele der eingesetzten Services funktionieren als Cluster und benötigen mehrere Server, um ausgeführt zu werden. Nicht jede Umgebung kann diese Anzahl an Servern bereitstellen. Daher müssen diese virtualisiert werden. Dazu wird hier Docker[3] verwendet. Docker ist eine Laufzeit für Container. Diese sind ähnlich zu klassischen virtuellen Maschinen, verbrauchen aber weniger Ressourcen. Ein weiterer Vorteil ist, dass Container für bestimmte Anwendungen gebaut werden. Ist ein Container einmal gebaut, kann er einfach verteilt und gestartet werden und ist direkt in einem lauffähigen Zustand. Zum Beispiel gibt es fertige Container, um einen Spark-Master oder -Worker zu starten. Damit ist der Aufbau eines Spark-Clusters wesentlich einfacher und reproduzierbarer als die Installation in mehreren virtuellen Maschinen. Zusätzlich gibt es Projekte, wie Kubernetes[4] oder Docker Swarm[5], die es erlauben Container auf einem verteilten Cluster auszuführen. Diese werden dann über das Cluster verteilt und über ein virtuelles Netzwerk verbunden. So wird auch die horizontale Skalierung der verfügbaren Hardware vereinfacht.

Das Ziel ist es, den gesamten Data Lake in Containern bereit zu stellen. Dabei treten allerdings ein paar Hürden auf, da die verwendeten Rahmenwerke nicht alle auf die Verwendung in Containern ausgelegt sind. Als erstes muss darauf geachtet werden, dass Container von sich aus keine Persistenz besitzen. Jenen Daten beziehungsweise Ordnern im Container, die nicht bei einem Neustart verloren gehen dürfen, müssen Volumes[6] zugeordnet werden. Außerdem müssen alle Container im selben Netzwerk sein, damit diese untereinander kommunizieren können. Da die IP-Adressen, welche die Container in diesen Netzwerken bekommen, nicht außerhalb des Netzwerks erreichbar sind, müssen sowohl das HDFS als auch die SparkSession so konfiguriert werden, dass sie die Host-Namen der Name- und DataNodes verwenden. Die Host-Namen müssen dann auf die IP-Adresse des Servers aufgelöst werden, auf dem die Container laufen. Um die

[3] https://docker.com
[4] https://kubernetes.io/
[5] https://docs.docker.com/engine/swarm/
[6] https://docs.docker.com/storage/volumes/

Services in den Containern von außen erreichen zu können, müssen die verwendeten Ports freigegeben werden. Dafür können Ports des Host-Systems zu Ports innerhalb des Containers zugeordnet werden. Dabei darf ein Host-Port immer nur einmal verwendet werden.

Die Bereitstellung der Services in Docker erfordert den Bau eigener Container. Als Basis wird das Docker-Image von Python verwendet. Für jeden Service müssen alle Python-Quell-Dateien kopiert werden. Bei der Ausführung ist es noch notwendig einen Ordner mit Konfigurationsdateien als Volume in den Container einzubinden.

5.8.1 Verwendete Docker Container

In **??** ist ein Beispiel zu sehen, welche Container für ein Data-Lake-System benötigt werden, wie es das Ergebnis dieser Arbeit ist. Jeder Block repräsentiert einen Container. Beschrieben werden hier nur die verwendeten Docker-Images und die freigegebenen Ports. Bei den Ports wird die Notation von Docker benutzt, bei der erst der Host-Port und als zweites der Ziel-Port im Container angegeben wird. Alle Container liegen in dem gleichen Docker-Netzwerk. Die grau hinterlegten Gruppen dienen nur der Verdeutlichung der Cluster. Die Anzahl der Spark-Worker, DataNodes oder Kafka-Broker kann je nach Hardware-Ressourcen angepasst werden. Alle Container bei denen der Docker-Image-Name mit „datalake/“ beginnt sind etxra für den Data Lake erstellt. Das Docker-Image „datalake/spark“ basiert auf „bitnami/spark“ und wurde angepasst, die gleiche Python-Version zu verwenden, die auch bei den Services zum Einsatz kommt.

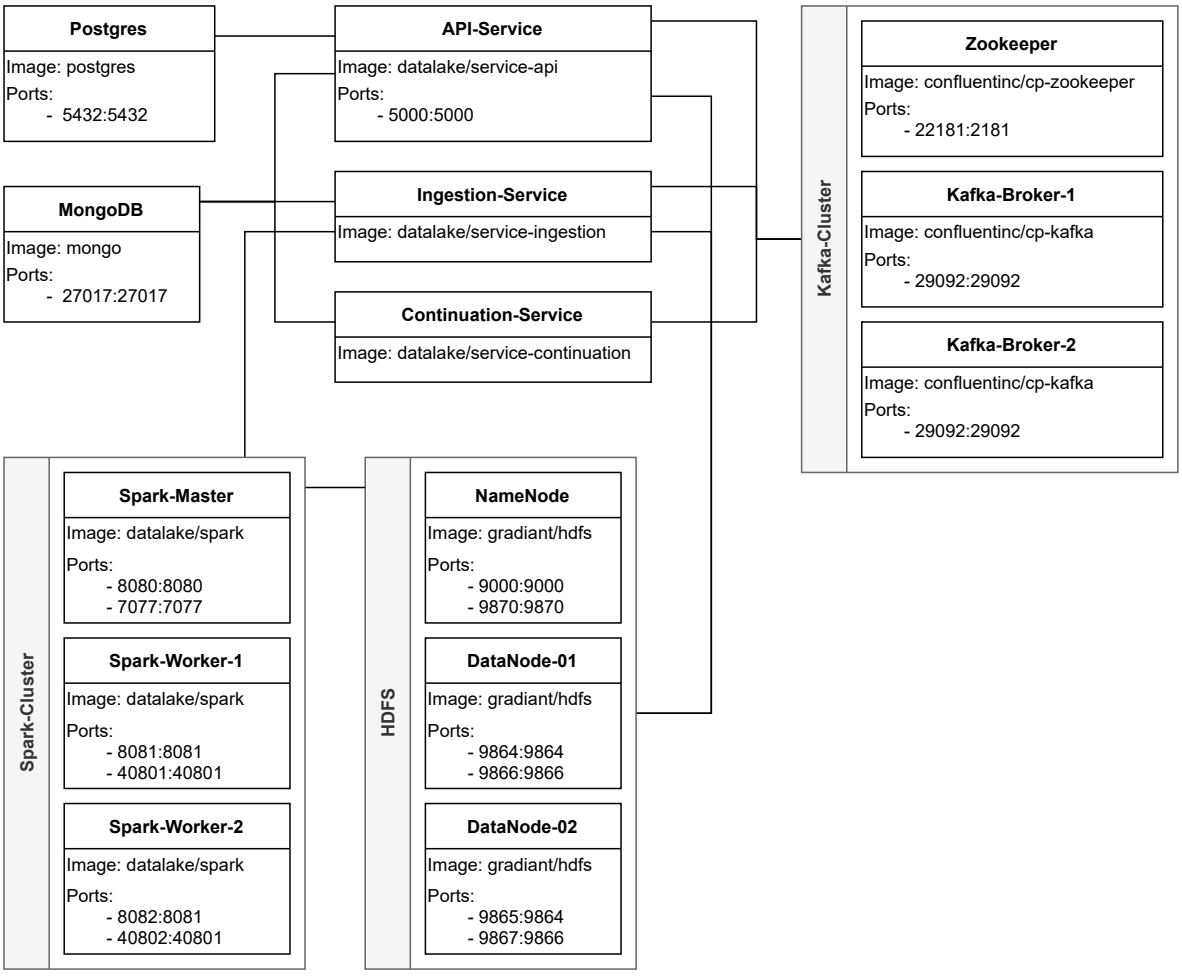


Abbildung 5.5: Docker-Container für den Data Lake

Kapitel 6

Evaluierung

Die Evaluierung der Ingestion-Schnittstelle besteht aus zwei Teilen. Im ersten Schritt wird geprüft, ob alle Anforderungen aus **??** erfüllt wurden. Dabei wird unterschieden in Anforderungen, deren Erfüllung durch die Betrachtung der Umsetzung bereits festgestellt werden kann und solche, bei denen ein funktionaler Test notwendig ist. Im zweiten Teil werden in einem Benchmark verschiedene Parameter gemessen, um einen Überblick zu geben, wie sich das System bei Verwendung verhält.

6.1 Erfüllung der Anforderungen

Die ersten Anforderungen die geprüft werden können, sind ANF$_1$2bis**??**.$DerenErfllungistdurchdieen$
$Schnittstelle, wiedurchdieAnforderunggefordert.EswerdenMetadatenwhrendderIngestionerfasstunddiedr$

Die Erfüllung der restlichen Anforderungen muss durch Tests überprüft werden. Hierzu werden Tests verwendet, die das komplette System im Zusammenhang überprüfen. Hier bedeutet das, dass mehrere Ingestions ausgeführt werden, die alle Funktionen abdecken. Mit den folgenden Schritten kann die korrekte Funktion einer Ingestion getestet werden:

1. Erstellung einer Datenquelle, zum Beispiel JSON-Dateien oder Postgres-Tabellen, die in der Ingestion verwendet wird.

2. Erstellung einer DatasourceDefinition für die Datenquelle.

3. Vergleichen der DatasourceDefinition aus der Datenbank des Systems mit einer vordefinierten Soll-Definition.

4. Starten der Ingestion.

5. Vergleichen der Daten im System nach der Ingestion mit einem vordefinierten Soll-Datensatz

Mit diesem Vorgehen kann auch die Aktualisierung von Daten getestet werden. Dazu muss es zweimal hintereinander ausgeführt werden. Im ersten Durchlauf werden die initialen Daten geladen und im zweiten wird dann entweder die gleiche Datenquelle mit veränderten Daten oder Änderungsdaten aus einer Update-Quelle geladen.

6.1.1 Durchgeführte Tests

Nachfolgend werden die durchgeführten Tests geschildert. Die ersten Tests sollen zeigen, dass sowohl das Laden von veränderten Daten als auch von Änderungsdaten für strukturierte und unstrukturierte Daten funktioniert. Damit wird auch indirekt die korrekte Ingestion überprüft.

Ein verwendeter Datensatz muss aus einer Mindestanzahl an Einträgen bestehen, die alle möglichen Operation bei einer Änderung abdecken. Der erste Eintrag bleibt in den Update-Daten unverändert, der zweite wird gelöscht und der dritte wird in einem Feld verändert. Zusätzlich muss noch ein weiterer Eintrag hinzugefügt werden. Damit Änderungen berechnet und eingepflegt werden können muss ein Feld festgelegt werden, dass als Id verwendet wird. Hier muss auch der Fall geprüft werden, dass die Id sich in einem verschachtelten Feld befindet.

Für den Test von strukturierten Daten wird eine Postgres Datenbank und für semistrukturierte JSON-Dateien verwendet. Damit werden der Upload von Dateien und das Laden von Daten aus einer Datenquelle abgedeckt. Für beide wird sowohl die Ingestion von veränderten Daten der gleichen Quelle als auch Änderungsdaten ausgeführt. Dabei werden jeweils eine Delta-Tabelle und Parquet als Speicherziel verwendet. Das Speichern in einem externen Ziel wird hier nicht getestet, da die Versionierung nur für interne Speicher unterstützt wird. Der

erfolgreiche Abschluss der Tests hat gezeigt, dass diese Ingestion-Abläufe bei veränderten Daten funktionieren:

- Speichern des neuen Stands einer Datenquelle in Parquet,
- Einpflegen von Änderungsdaten aus einer externen Quelle in die Bestandsdaten,
- Berechnen von Änderungsdaten aus einem neuem Stand einer Datenquelle und Speichern dieser in einer Delta-Tabelle und
- Einpflegen von Änderungsdaten aus einer externen Quelle in eine Delta-Tabelle.

Bei den Tests für weitere Quellen genügt es nur noch eine Ingestion auszuführen. Hier werden die Ingestion aus einem Kafka-Datenstrom und aus einer REST-Api überprüft. Die Ingestion des Datenstroms verwendet dabei ein After-Load-Plugin, um die tatsächlichen Daten aus der Nachricht zu extrahieren. Das Load-Plugin wird bei der Ingestion aus einer REST-API mit geprüft. Auch diese Tests sind erfolgreich durchgelaufen.

6.2 Benchmarks

Benchmarks sind ein allgemeines Mittel, um Systeme zu bewerten. Dabei werden festgelegte Operationen ausgeführt und bestimmte Parameter gemessen, um Aussagen über ein System treffen zu können. Im Big-Data-Bereich wird meistens die Menge der Daten, die verarbeitet werden können, bewertet. Hier sind Beispiele der Bigdatabench (**bigdatabench**) oder der TPCx-BB[1] und TCPx-HS[2]. Diese messen verschiedene Verarbeitungsoperationen mit großen Datenmengen auf einem bestehenden System. Teilweise werden zusammen mit der Datenmenge auch Kosten oder Stromverbrauch gemessen. Neben diesen Benchmarks kann auch die Qualität des Ergebnisses einer bestimmten Verarbeitungspipeline gemessen werden.

[1] http://tpc.org/tpcx-bb/default5.asp
[2] http://tpc.org/tpch/default5.asp

Für die Bewertung der Ingestion-Schnittstelle sind solche Benchmarks nicht geeignet, da die Geschwindigkeit oder Menge der Datenverarbeitung von Cluster zu Cluster unterschiedlich ist, die Schnittstelle aber für verschiedene Cluster eingesetzt werden soll. Hier bieten sich relative Vergleiche als eine bessere Lösung an. Dazu werden verschieden große Datensätze unter den gleichen Bedingungen in das System geladen und die Dauer der Ingestion und der verbrauchte Festplattenplatz gemessen.

6.2.1 Benchmark-Vorgehen

Ein wichtiger und ausschlaggebender Punkt der Ingestion ist die Versionierung und Verarbeitung von Änderungsdaten. Daraus folgt, dass hier für die Benchmarks das Verhalten beim Laden von Updates interessant ist. Dafür werden die vier Ingestion-Abläufe aus **??**, Updates in Parquet, Änderungsdaten in Parquet, Updates in eine Delta-Tabelle und Änderungsdaten in eine Delta-Tabelle mit schrittweise erhöhten Datenmengen durchlaufen. Zusätzlich werden sowohl strukturierte als auch semistrukturierte Daten verwendet.

Die Beispieldatensätze wurden eigens für die Benchmarks generiert, um die genaue Anzahl und den Inhalt bestimmen zu können. So bleiben die Daten auch zwischen den Strukturen vergleichbar. Zur Generierung wurde das Tool Synth[3] verwendet. Bei diesem wird in einer JSON-Datei ein Schema definiert. In dem Schema kann für ein Feld aus verschiedenen Generatoren gewählt werden, die zufällige oder feste Werte in den Daten erzeugen. Es ist auch in der Lage realitätsnahe Daten wie Namen oder E-Mail-Adressen zu generieren. Als Speicherziel können entweder JSON-Dateien oder einige Datenbanken gewählt werden.

Bei der Datengenerierung wird zuerst ein initialer Datensatz erstellt. Für diesen werden dann eine bestimmte Anzahl an Änderungen, Löschungen und Einfügungen erzeugt, die einmal in einer Kopie des Datensatzes als neuer Datensatz angewendet werden und einmal in einem Datensatz in Form von Änderungsdaten gespeichert werden. Die Menge der Daten wird durch vier Parameter gesteuert. Der erste ist die Anzahl an Zeilen im initialen Datensatz. Die anderen drei legen die Änderungen, Löschungen und Einfügungen fest. Sie werden in Prozent angegeben und werden auf Basis der Anzahl an Zeilen berechnet.

[3] https://www.getsynth.com/

Für die Durchführung eines Benchmarks werden dann die folgenden Schritte ausgeführt. Dabei sind fast alle Schritte gleich. Nur Schritt vier und fünf unterscheiden sich bei der Verwendung einer veränderten Tabelle oder von Änderungsdaten. Für die Benchmark-Messungen kann die Dauer aus den Ingestion-Events berechnet werden, da diese die Start- und Endzeit enthalten. Die Größe der Dateien wird über eine WebHDFS-Abfrage der Eigenschaften des entsprechenden Ordners geschehen, da sowohl Parquet als auch die Delta-Tabelle in HDFS gespeichert werden.

1. Erstellen einer DatasourceDefinition für die initialen Daten. Bei Postgres wird die Tabelle angegeben und bei JSON werden die entsprechenden Dateien hochgeladen.
2. Führe die erste Ingestion mit der Datenquelle aus.
3. Überprüfe, ob die Anzahl der Einträge in den geladenen Daten korrekt ist und erfasse Benchmark-Parameter für die initialen Daten.
4. Erstelle DatasourceDefinition für Änderungsdaten oder ändere die bestehende DatasourceDefinition, die veränderten Daten zu benutzen. Damit bei mehrfachen Benchmarks nicht immer wieder neu die Änderungen gemacht werden müssen, wird das dadurch simuliert, dass die Datenquelle der DatasourceDefinition angepasst wird. Für Postgres wird hier einfach die Tabelle geändert und für JSON werden die Dateien mit den neuen Daten hochgeladen und die alten nicht in die Liste der Quelldateien übernommen.
5. Führe entweder eine erneute Ingestion der Datenquelle aus oder führe die erste Ingestion für die Änderungsdaten aus.
6. Überprüfe, ob die Anzahl der Daten korrekt ist und erfasse Benchmark-Parameter für die veränderten Daten.

Die durchgeführten Benchmarks bilden zwei verschiedene Szenarien ab. Der erste Fall ist eine Datenquelle in der häufig Änderungen vorkommen. Hier werden 80% Änderungen, 20% Löschungen und 40% Einfügungen verwendet. Im zweiten Fall geht es darum, dass hauptsächlich neue Daten eingefügt werden und Änderungen und Löschungen werden selten gemacht. Dafür werden 5% Änderungen, 5% Löschungen und 80% Einfügungen angewendet. Die Benchmarks

für ein Speicherziel, also Parquet oder Delta-Tabelle, werden immer zusammen ausgeführt. Das heißt, dass die Ingestion von aktualisierten Daten und von Änderungsdaten direkt hintereinander geschieht. Dadurch kann eine weitere Überprüfung mit durchgeführt werden. Da das Speicherziel gleich ist muss für beide Durchläufe auch die Speichergröße der initialen Daten gleich sein.

Als Ziele der Datengenerierung und somit auch als Datenquellen werden wie bei den Tests Postgres und JSON-Dateien verwendet. Die Daten selbst bestehen aus jeweils zehn Feldern, wovon eines eine fortlaufende Nummer als Id ist, eines ein Datum enthält und die anderen mit einem zufälligen 32-Zeichen langen Text befüllt werden. Bei den semistrukturierten Daten werden außerdem die Felder ineinander verschachtelt. Die Anzahl der Einträge wird in Zehner-Potenzen erhöht. Dabei der werden die Faktoren 1, $2,5$, 5 und $7,5$ mit den Potenzen 10^3 bis 10^6 verwendet.

6.2.2 Benchmark-Ergebnisse

Die Benchmarks konnten nur auf einem lokalen Spark-Cluster mit einem Worker ausgeführt werden. Diesem standen 8 Kerne und 20 Gigabyte Arbeitsspeicher zur Verfügung. Es wurde auch probiert diesen Worker in zwei, mit jeweils der Hälfte der Ressourcen, aufzuteilen. Bei dieser Konfiguration konnten die Benchmarks aber nicht für eine Zeilenanzahl ab 1.000.000 erfolgreich beendet werden, da die Ressourcen nicht ausreichend waren. Aufgrund des verwendeten Spark-Clusters hat die Betrachtung der absoluten Zahlen keine große Aussagekraft. Diese ändern sich mit dem Spark-Cluster und sind nur sinnvoll, wenn der Benchmark auf dem System ausgeführt wird, das auch beim Betrieb zum Einsatz kommen soll. Eine der Kern-Funktionen ist die Versionierung. Daher werden die Benchmark-Ergebnisse mit dem Ziel verglichen, Aussagen über diese Funktion zu treffen. Dazu wird jeweils ein Faktor für die zwei Szenarien und die Quelle

wie folgt berechnet:

p_t : Zeit beim Speichern in Parquet, ohne Versionierung
d_t : Zeit beim Speichern in Delta-Tabelle, mit Versionierung
$f_t = d_t / p_t$

p_s : Speicherverbrauch in Parquet, ohne Versionierung
d_s : Speicherverbrauch in Delta-Tabelle, mit Versionierung
$f_s = d_s / p_s$

Diese Werte werden für die zweite Ingestion für beide Fälle mit jeweils aktualisierten Daten oder Änderungsdaten berechnet. Da die initialen Daten in beiden Fällen gleich sind, werden die Faktoren hierfür nur einmal berechnet.

Der Wert der Faktoren kann folgendermaßen interpretiert werden:

$f_t > 1$: Die Ingestion mit Versionierung ist langsamer als ohne
$f_t = 1$: Beide Ingestions sind gleich schnell
$f_t < 1$: Die Ingestion mit Versionierung ist schneller als ohne

$f_s > 1$: Die Ingestion mit Versionierung verbraucht mehr Speicherplatz
$f_s = 1$: Beide Ingestions verbrauchen gleich viel Speicher
$f_s < 1$: Die Ingestion mit Versionierung verbraucht weniger Speicherplatz

In den gezeigten Diagrammen werden nur die Ergebnisse ab 100.000 Zeilen gezeigt, da die Unterschiede vorher zu klein sind, um sie erkennbar darzustellen. Alle Ergebnisse können in den Tabellen in **??** nachgelesen werden. Die Diagramme sind jeweils so aufgebaut, dass eines die Werte der initialen Ingestion, eines die Werte der Ingestion aus einer aktualisierten Quelle und eines die Werte der Ingestion einer Update-Quelle zeigt.

Abbildung 6.1: Dauer erste Ingestion

Abbildung 6.2: Dauer Ingestion aus aktualisierten Daten

Dauer der Ingestion

Allgemein ist zu sehen, dass die Ingestion mit einer Versionierung immer länger dauert als ohne. Bei der ersten Ingestion (**??**) und der aus einer Update-Quelle (**??**) ist die Differenz annähernd konstant. Sie kann damit erklärt werden, dass in den Delta-Tabellen neben den Daten selbst noch extra Informationen zur Versionierung gespeichert werden. Bei der Ingestion aus einer aktualisierten Datenquelle (**??**) ist die Differenz jedoch deutlich höher. Hierfür ist die Berechnung der Änderungsdaten verantwortlich, die für die Versionierung erforderlich sind.

Ein weiterer Punkt, der bei allen Ingestions zu beobachten ist, ist, dass die Ingestion von semistrukturierten Daten schneller ist, als die von strukturierten Daten. Das liegt an der bereits angesprochenen Eigenschaft von verschachtelten Daten in Spark. Beide Datensätze haben gleich viele Felder, da aber bei verschachtelten Daten nur die oberste Ebene als Spalte betrachtet wird, haben diese effektiv weniger Spalten. Da Parquet, und damit auch in Delta-Tabellen, für jede Spalte zusätzliche Informationen enthält, müssen bei semistrukturierten Daten Informationen gespeichert werden.

Auch zwischen den zwei Szenarien für die Mengen der Änderungen sind Unterschiede zu erkennen. Im zweiten Szenario ist die Ingestion aus einer aktualisierten Datenquelle wesentlich langsamer als im Ersten. Hier liegt der Grund in der Menge der eingefügten Daten. Im ersten Szenario sind dies nur 40% aber im zweiten 80%. Das Ergebnis der Join-Operation bei der Berechnung von Änderungsdaten ist also im zweiten Szenario größer als im ersten, was zu einer längeren Berechnungszeit führt. Im Gegensatz dazu, ist dies bei einer Ingestion aus einer Update-Quelle genau anders, da insgesamt im ersten Szenario mehr Änderungen an dem Datensatz gemacht werden.

Abbildung 6.3: Dauer Ingestion aus Update-Quelle

Abbildung 6.4: Speicherverbrauch nach der ersten Ingestion

Abbildung 6.5: Speicherverbrauch nach der Ingestion aus aktualisierten Daten

Speicherverbrauch

Für die initial Ingestion **??** ist kein Unterschied im Speicherverbrauch zu erkennen. Bei Betrachtung der genauen Werte ist zu sehen, dass dieser minimal vorhanden ist. Das liegt daran, dass Delta-Tabellen die Daten auch in Parquet speichern und nach der ersten Ingestion noch keine Versionierungsinformationen vorhanden sind. Diese treten erst bei einer weiteren Ingestion auf.

Bei einer Ingestion von aktualisierten Daten (**??**) und aus einer Update-Quelle (**??**) ist der Speicherverbrauch gleich. Beide Quellen enthalten die gleichen Änderungen, weshalb auch das gespeicherte Ergebnis gleich sein muss. Der Unterschied zwischen den Szenarien liegt darin, dass im zweiten Szenario mehr Daten eingefügt und weniger gelöscht werden, so dass am Ende mehr Daten im Datensatz enthalten sind.

Auch beim Speicherverbrauch kommt die Eigenschaft der verschachtelten Daten in Spark zum Tragen. Man sieht, dass weniger Spalteninformationen gespeichert werden müssen, was den Verbrauch niedriger hält. Daraus lässt sich aber auch schließen, dass ohne weitere Verarbeitung weniger Informationen über Änderungen vorhanden sind. Das heißt, dass bei Änderungen in einem verschachtelten Feld nur erkannt wird, dass sich das Feld auf der obersten Ebene darüber verändert hat. Die genauen Änderungen müssen, durch zum Beispiel einen Vergleich der Versionen, nachträglich berechnet werden.

Abbildung 6.6: Speicherverbrauch nach der Ingestion aus Update-Quelle

Kapitel 7

Zusammenfassung und Ausblick

Die Aufgabenstellung dieser Arbeit war die Entwicklung einer Ingestion-Schnittstelle für ein Data-Lake-System, dass im Kontext des HIT-Instituts an der Hochschule Niederrhein angewendet werden soll. An diesem Institut werden Daten aus verschiedenen Quellen erfasst. Es gibt klassische Datenbank-Systeme, Dateien oder REST-APIs. Der Data Lake soll die zentrale Speicherlösung sein und Verarbeitungsprozesse unterstützen. Dazu ist es neben der Unterstützung verschiedener Datenquellen auch nötig die Daten zu versionieren. Dadurch verringert sich der Aufwand bei der weiteren Verarbeitung, da nur Änderungen betrachtet werden müssen.

In der vorliegenden Arbeit wurde eine generische Ingestion-Schnittstelle mit Versionierung für Data-Lake-Systeme entwickelt und mit funktionalen Tests und Benchmarks evaluiert. Dafür wurde neben der Entwicklung der bestehende Prototyp des Data Lake auf eine Microservice-Architektur umgebaut. Es wurden außerdem existierende Data-Lake-Systeme betrachtet und bewertet, ob diese als Lösung für die Problemstellung dienen könnten. Daraus ergab sich, dass eine Eigenentwicklung notwendig ist.

Die Ingestion-Schnittstelle besteht aus drei Microservices. Der Ingestion-Service verwaltet das Laden von Daten in das System. Dabei erlaubt die Verwendung von Spark und einem Plugin-System das Laden von Daten aus jeder Quelle. Die kontinuierliche Ausführung von Ingestions wird durch den Continuation-

Service sichergestellt. Benutzer können über den API-Service mit der Ingestion-Schnittstelle interagieren. Dieser besteht aus der Integration der neuen API in den Server des Prototyps. Dadurch bleibt das System mit der Verwendung der alten API kompatibel. Aus der Eingabe, die ein Benutzer an die API sendet, wird ein generelles Metadatenmodell aufgebaut.

Das Metadatenmodell zur Definition einer Datenquelle enthält alle Informationen über die Datenquelle, verwendete Plugins und zugehörige Ingestions. Über dieses Modell werden die Informationen zu den Ingestions zwischen den Microservices ausgetauscht.

Neben der Ingestion-Schnittstelle besteht der Data Lake aus weiteren Komponenten. Ein Kafka-Cluster wird für die Kommunikation im Data-Lake-System eingesetzt. Diese kann auch für die Anbindung von Datenströmen verwendet werden. Die Daten, die in den Data Lake geladen wurden, werden in einem HDFS gespeichert. Daten mit Versionierung werden in einer Delta-Tabelle des Delta Lake abgelegt. Für unversionierte Daten wird entweder das Parquet- oder, bei unstrukturierten Daten, das Ursprungsformat der Daten benutzt. Für die Verwaltung von Metadaten kommt eine MongoDB-Datenbank zum Einsatz. Die Datenverarbeitung geschieht über ein Spark-Cluster.

Die Evaluierung der Ingestion-Schnittstelle wurde in einem durch die gegebene Hardware möglichen Rahmen durchgeführt. Hierfür stand kein leistungsstarkes Cluster zur Verfügung, um absolute Aussagen über die Leistung der Ingestion-Schnittstelle treffen zu können. Daher ist lediglich ein relativer Vergleich ausgeführt worden. Dabei wurden Benchmarks für verschiedene Datenmengen mit strukturierten und semistrukturierten Daten in zwei Szenarien ausgeführt. Die Szenarien bilden dabei eine Situation, in der viele Änderungen an Daten geschehen und eine Situation, in der hauptsächlich Daten hinzugefügt werden ab.

Durch die Benchmarks und Tests wurde gezeigt, dass mit dieser Arbeit die Entwicklung der Ingestion-Schnittstelle noch nicht vollständig abgeschlossen ist. Die implementierte Berechnung von Änderungsdaten ist in bestimmten Fällen sehr langsam. Auch die Darstellung von Änderungen in verschachtelten Daten könnte durch ein anderes Vorgehen optimiert werden. Somit ist die Versionierung in der Ingestion-Schnittstelle ein Thema, dass in einer anschließenden Arbeit genauer betrachtet werden kann.

Weiterhin sind die Metadaten, die während der Ingestion gesammelt werden, nicht ausreichend. Um eine Aussagekräftige Suche, Exploration und Analyse der Daten durchführen zu können, müssen direkt nach dem Laden weitere Metadaten extrahiert werden. Diese könnten zum Beispiel Informationen über den Inhalt oder die Struktur der Daten beinhalten. Auch das ist ein großes Thema, mit dem sich während der Weiterentwicklung befasst werden kann.

Anhang A

Benchmark Ergebnisse

Nachfolgend sind alle Ergebnisse der Benchmarks aufgelistet. Zeiten sind dabei immer im Format Minuten:Sekunden. In den Spaltenüberschriften werden die drei Buchstaben mit folgenden Bedeutungen verwendet:

I : Initiale Ingestion

U : Ingestion mit aktualisierten Daten

C : Ingestion mit Änderungsdaten

Zeilen	Speicher I	Zeit I	Speicher U	Zeit U	Speicher C	Zeit C
1,0e+3	0,96 MB	00:07	0,83 MB	00:07	0,84 MB	00:10
2,5e+3	2,39 MB	00:07	2,11 MB	00:07	2,11 MB	00:10
5,0e+3	4,75 MB	00:07	4,22 MB	00:07	4,22 MB	00:10
7,5e+3	7,12 MB	00:08	6,29 MB	00:08	6,30 MB	00:10
1,0e+4	9,48 MB	00:08	8,47 MB	00:08	8,47 MB	00:10
2,5e+4	23,66 MB	00:08	21,14 MB	00:08	24,45 MB	00:10
5,0e+4	47,30 MB	00:09	52,77 MB	00:08	52,30 MB	00:11
7,5e+4	70,94 MB	00:09	79,15 MB	00:09	79,23 MB	00:12
1,0e+5	94,57 MB	00:09	105,38 MB	00:09	105,70 MB	00:13
2,5e+5	236,39 MB	00:11	264,49 MB	00:12	264,71 MB	00:16
5,0e+5	474,70 MB	00:15	531,68 MB	00:16	529,78 MB	00:20
7,5e+5	711,06 MB	00:18	796,23 MB	00:20	795,45 MB	00:25
1,0e+6	949,38 MB	00:21	1.060,58 MB	00:23	1.060,46 MB	00:29
2,5e+6	2.373,41 MB	00:40	2.651,87 MB	00:48	2.657,99 MB	01:08
5,0e+6	4.748,75 MB	01:16	5.305,38 MB	01:28	5.318,29 MB	01:57
7,5e+6	7.124,09 MB	01:47	7.956,28 MB	02:12	7.977,94 MB	02:58

Tabelle A.1: Strukturierte Daten - Fall 1 - Ohne Versionierung

Zeilen	Speicher I	Zeit I	Speicher U	Zeit U	Speicher C	Zeit C
1,0e+3	0,96 MB	00:13	4,19 MB	00:22	4,19 MB	00:20
2,5e+3	2,39 MB	00:13	7,22 MB	00:22	7,22 MB	00:20
5,0e+3	4,76 MB	00:13	12,28 MB	00:23	12,28 MB	00:20
7,5e+3	7,12 MB	00:13	17,32 MB	00:23	17,32 MB	00:21
1,0e+4	9,48 MB	00:14	22,35 MB	00:22	22,35 MB	00:20
2,5e+4	23,67 MB	00:13	52,49 MB	00:23	52,49 MB	00:20
5,0e+4	47,30 MB	00:13	102,70 MB	00:23	102,70 MB	00:21
7,5e+4	70,94 MB	00:13	152,90 MB	00:25	152,90 MB	00:21
1,0e+5	94,57 MB	00:14	203,10 MB	00:26	203,10 MB	00:22
2,5e+5	236,39 MB	00:16	504,28 MB	00:30	504,28 MB	00:24
5,0e+5	474,70 MB	00:19	1.008,09 MB	00:39	1.008,09 MB	00:30
7,5e+5	711,06 MB	00:23	1.509,56 MB	00:49	1.509,56 MB	00:34
1,0e+6	949,38 MB	00:26	2.012,54 MB	00:55	2.012,54 MB	00:41
2,5e+6	2.373,41 MB	00:45	5.024,57 MB	01:47	5.024,57 MB	01:20
5,0e+6	4.748,75 MB	01:18	10.046,59 MB	03:33	10.046,59 MB	02:17
7,5e+6	7.124,09 MB	01:51	15.058,13 MB	05:09	15.058,13 MB	03:32

Tabelle A.2: Strukturierte Daten - Fall 1 - Mit Versionierung

Zeilen	Speicher I	Zeit I	Speicher U	Zeit U	Speicher C	Zeit C
1,0e+3	0,65 MB	00:08	7,94 MB	00:07	7,82 MB	00:10
2,5e+3	1,61 MB	00:07	19,50 MB	00:07	19,34 MB	00:10
5,0e+3	3,21 MB	00:07	38,67 MB	00:07	38,42 MB	00:11
7,5e+3	4,80 MB	00:08	57,79 MB	00:07	57,50 MB	00:10
1,0e+4	6,39 MB	00:08	76,87 MB	00:07	76,58 MB	00:10
2,5e+4	15,92 MB	00:08	191,32 MB	00:07	191,43 MB	00:11
5,0e+4	31,81 MB	00:08	382,01 MB	00:07	382,95 MB	00:11
7,5e+4	47,73 MB	00:08	573,05 MB	00:07	573,80 MB	00:12
1,0e+5	63,63 MB	00:08	763,73 MB	00:08	766,76 MB	00:12
2,5e+5	159,07 MB	00:09	1.909,18 MB	00:10	1.915,76 MB	00:14
5,0e+5	318,15 MB	00:12	3.818,39 MB	00:12	3.832,67 MB	00:17
7,5e+5	477,25 MB	00:13	5.727,59 MB	00:14	5.738,92 MB	00:19
1,0e+6	636,31 MB	00:15	7.636,83 MB	00:16	7.645,30 MB	00:22
2,5e+6	1.590,86 MB	00:22	19.092,19 MB	00:25	19.023,51 MB	00:37
5,0e+6	3.181,51 MB	00:35	38.183,02 MB	00:42	38.129,25 MB	01:11
7,5e+6	4.772,36 MB	00:49	57.275,36 MB	01:00	57.418,35 MB	01:42

Tabelle A.3: Semistrukturierte Daten - Fall 1 - Ohne Versionierung

Zeilen	Speicher I	Zeit I	Speicher U	Zeit U	Speicher C	Zeit C
1,0e+3	0,65 MB	00:14	3,28 MB	00:23	3,28 MB	00:19
2,5e+3	1,61 MB	00:12	5,43 MB	00:21	5,43 MB	00:20
5,0e+3	3,21 MB	00:13	9,00 MB	00:22	9,00 MB	00:20
7,5e+3	4,80 MB	00:12	12,56 MB	00:21	12,56 MB	00:19
1,0e+4	6,39 MB	00:13	16,09 MB	00:22	16,10 MB	00:20
2,5e+4	15,92 MB	00:13	37,27 MB	00:23	37,28 MB	00:20
5,0e+4	31,81 MB	00:13	72,58 MB	00:24	72,60 MB	00:21
7,5e+4	47,74 MB	00:13	107,90 MB	00:24	107,99 MB	00:21
1,0e+5	63,63 MB	00:13	143,32 MB	00:24	143,32 MB	00:21
2,5e+5	159,08 MB	00:14	354,70 MB	00:28	354,70 MB	00:24
5,0e+5	318,16 MB	00:16	706,04 MB	00:31	706,05 MB	00:28
7,5e+5	477,25 MB	00:17	1.055,49 MB	00:34	1.055,50 MB	00:31
1,0e+6	636,32 MB	00:19	1.404,25 MB	00:39	1.404,25 MB	00:34
2,5e+6	1.590,87 MB	00:27	3.496,49 MB	01:02	3.498,78 MB	00:55
5,0e+6	3.181,53 MB	00:39	6.900,47 MB	01:41	6.888,11 MB	01:23
7,5e+6	4.772,39 MB	00:53	10.359,32 MB	02:33	10.361,91 MB	02:01

Tabelle A.4: Semistrukturierte Daten - Fall 1 - Mit Versionierung

Zeilen	Speicher I	Zeit I	Speicher U	Zeit U	Speicher C	Zeit C
1,0e+3	0,96 MB	00:07	1,02 MB	00:07	1,02 MB	00:10
2,5e+3	2,39 MB	00:07	2,57 MB	00:07	2,57 MB	00:10
5,0e+3	4,75 MB	00:07	5,15 MB	00:07	5,15 MB	00:09
7,5e+3	7,12 MB	00:08	7,71 MB	00:08	7,71 MB	00:10
1,0e+4	9,48 MB	00:07	10,33 MB	00:07	10,33 MB	00:10
2,5e+4	23,66 MB	00:08	25,94 MB	00:08	34,81 MB	00:11
5,0e+4	47,30 MB	00:08	82,51 MB	00:09	78,24 MB	00:11
7,5e+4	70,94 MB	00:09	123,75 MB	00:09	122,05 MB	00:12
1,0e+5	94,57 MB	00:09	164,99 MB	00:10	162,62 MB	00:12
2,5e+5	236,39 MB	00:11	414,83 MB	00:12	413,71 MB	00:14
5,0e+5	474,70 MB	00:14	827,02 MB	00:19	827,06 MB	00:17
7,5e+5	711,06 MB	00:17	1.240,49 MB	00:24	1.240,20 MB	00:21
1,0e+6	949,38 MB	00:21	1.658,10 MB	00:30	1.653,52 MB	00:30
2,5e+6	2.373,41 MB	00:39	4.142,78 MB	01:04	4.136,67 MB	00:48
5,0e+6	4.748,75 MB	01:13	8.284,29 MB	02:02	8.276,20 MB	01:35
7,5e+6	7.124,09 MB	01:45	12.430,15 MB	03:08	12.458,08 MB	02:18

Tabelle A.5: Strukturierte Daten - Fall 2 - Ohne Versionierung

Zeilen	Speicher I	Zeit I	Speicher U	Zeit U	Speicher C	Zeit C
1,0e+3	0,96 MB	00:14	4,75 MB	00:21	4,75 MB	00:19
2,5e+3	2,39 MB	00:12	8,72 MB	00:21	8,72 MB	00:18
5,0e+3	4,76 MB	00:13	15,29 MB	00:22	15,29 MB	00:19
7,5e+3	7,12 MB	00:13	21,82 MB	00:22	21,82 MB	00:19
1,0e+4	9,48 MB	00:12	28,36 MB	00:20	28,36 MB	00:19
2,5e+4	23,67 MB	00:13	67,52 MB	00:22	67,52 MB	00:19
5,0e+4	47,30 MB	00:14	132,74 MB	00:24	132,74 MB	00:21
7,5e+4	70,94 MB	00:14	197,88 MB	00:25	197,88 MB	00:21
1,0e+5	94,57 MB	00:14	263,10 MB	00:25	263,10 MB	00:21
2,5e+5	236,39 MB	00:16	654,40 MB	00:32	654,40 MB	00:24
5,0e+5	474,70 MB	00:19	1.307,04 MB	00:42	1.307,04 MB	00:26
7,5e+5	711,06 MB	00:23	1.956,73 MB	00:55	1.956,73 MB	00:34
1,0e+6	949,38 MB	00:26	2.608,57 MB	01:05	2.608,57 MB	00:38
2,5e+6	2.373,41 MB	00:44	6.512,17 MB	02:20	6.512,17 MB	01:02
5,0e+6	4.748,75 MB	01:19	13.035,43 MB	04:35	13.035,43 MB	01:51
7,5e+6	7.124,09 MB	01:50	19.542,27 MB	10:17	19.542,27 MB	02:39

Tabelle A.6: Strukturierte Daten - Fall 2 - Mit Versionierung

Zeilen	Speicher I	Zeit I	Speicher U	Zeit U	Speicher C	Zeit C
1,0e+3	0,65 MB	00:10	1,14 MB	00:07	1,13 MB	00:10
2,5e+3	1,61 MB	00:07	2,83 MB	00:07	2,80 MB	00:10
5,0e+3	3,21 MB	00:07	5,62 MB	00:07	5,58 MB	00:11
7,5e+3	4,80 MB	00:07	8,40 MB	00:07	8,36 MB	00:10
1,0e+4	6,39 MB	00:07	11,18 MB	00:07	11,15 MB	00:10
2,5e+4	15,92 MB	00:07	27,87 MB	00:08	27,89 MB	00:11
5,0e+4	31,81 MB	00:08	55,68 MB	00:08	55,83 MB	00:11
7,5e+4	47,73 MB	00:08	83,55 MB	00:08	83,61 MB	00:11
1,0e+5	63,63 MB	00:08	111,36 MB	00:08	111,65 MB	00:12
2,5e+5	159,07 MB	00:09	278,39 MB	00:10	279,37 MB	00:14
5,0e+5	318,15 MB	00:11	556,80 MB	00:13	558,88 MB	00:17
7,5e+5	477,25 MB	00:12	835,20 MB	00:15	837,07 MB	00:18
1,0e+6	636,31 MB	00:13	1.113,59 MB	00:18	1.115,78 MB	00:21
2,5e+6	1.590,86 MB	00:22	2.784,02 MB	00:32	2.781,98 MB	00:36
5,0e+6	3.181,51 MB	00:39	5.567,79 MB	01:00	5.576,58 MB	01:09
7,5e+6	4.772,36 MB	00:52	8.351,79 MB	01:25	8.412,88 MB	01:36

Tabelle A.7: Semistrukturierte Daten - Fall 2 - Ohne Versionierung

Zeilen	Speicher I	Zeit I	Speicher U	Zeit U	Speicher C	Zeit C
1,0e+3	0,65 MB	00:15	3,64 MB	00:22	3,64 MB	00:19
2,5e+3	1,61 MB	00:12	6,34 MB	00:21	6,34 MB	00:20
5,0e+3	3,21 MB	00:12	10,80 MB	00:22	10,80 MB	00:19
7,5e+3	4,80 MB	00:13	15,23 MB	00:23	15,24 MB	00:20
1,0e+4	6,39 MB	00:12	19,65 MB	00:22	19,67 MB	00:19
2,5e+4	15,92 MB	00:13	46,13 MB	00:22	46,18 MB	00:20
5,0e+4	31,81 MB	00:13	90,38 MB	00:24	90,49 MB	00:21
7,5e+4	47,74 MB	00:13	134,39 MB	00:24	134,62 MB	00:21
1,0e+5	63,63 MB	00:13	178,73 MB	00:24	178,73 MB	00:21
2,5e+5	159,08 MB	00:14	443,04 MB	00:28	443,05 MB	00:23
5,0e+5	318,16 MB	00:17	881,61 MB	00:35	881,76 MB	00:28
7,5e+5	477,25 MB	00:17	1.316,34 MB	00:38	1.316,35 MB	00:29
1,0e+6	636,32 MB	00:18	1.752,75 MB	00:43	1.754,19 MB	00:35
2,5e+6	1.590,87 MB	00:26	4.343,15 MB	01:11	4.377,65 MB	00:52
5,0e+6	3.181,53 MB	00:40	8.573,33 MB	02:02	8.637,57 MB	01:19
7,5e+6	4.772,39 MB	00:50	13.028,96 MB	03:05	13.029,04 MB	01:45

Tabelle A.8: Semistrukturierte Daten - Fall 2 - Mit Versionierung

Abbildungsverzeichnis